Mern Stack Web Development
UNIT 2

1.Explain Events in Reacts with Example

A)

Events in Java script

Mouse events:

onclick :

The event occurs when the user clicks on an element

oncontextmenu :

User right-clicks on an element to open a context menu

ondblclick :

The user double-clicks on an element

onmousedown :

User presses a mouse button over an element

onmouseenter :

The pointer moves onto an element

onmouseleave :

Pointer moves out of an element

onmousemove:

The pointer is moving while it is over an element

onmouseover :

When the pointer is moved onto an element or one of its children

onmouseout :

User moves the mouse pointer out of an element or one of its children

onmouseup :

The user releases a mouse button while over an element

Keyboard Events

onkeydown :

When the user is pressing a key down

onkeypress :

The moment the user starts pressing a key

onkeyup :

The user releases a key Frame

onabort :

The loading of a media is aborted

onbeforeunload :

Event occurs before the document is about to be unloaded

onerror :

An error occurs while loading an external

onhashchange :

There have been changes to the anchor part of a URL onload When an object has loaded

onpagehide:

The user navigates away from a webpage

onpageshow :

When the user navigates to a webpage

onresize :

The document view is resized

onscroll :

An element's scrollbar is being scrolled

onunload :

Event occurs when a page has unloaded

Form Events

onblur :

When an element loses focus

onchange :

The content of a form element changes (for , and)

onfocus :

An element gets focus

onfocusin :

When an element is about to get focus

onfocusout :

The element is about to lose focus

oninput :

User input on an element

oninvalid :

An element is invalid

onreset :

A form is reset

onsearch :

The user writes something in a search field (for)

onselect :

The user selects some text (for and)

onsubmit :

A form is submitted

React events are written in camelCase syntax:

onClick instead of onclick.

React event handlers are written inside curly braces: onClick={shoot} instead of onclick="shoot()".

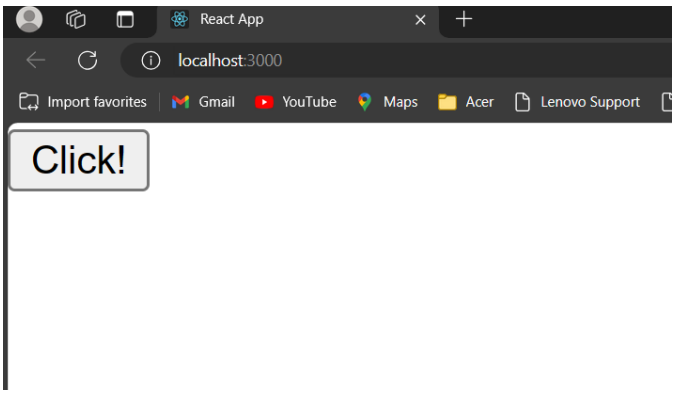
Syntax for react:

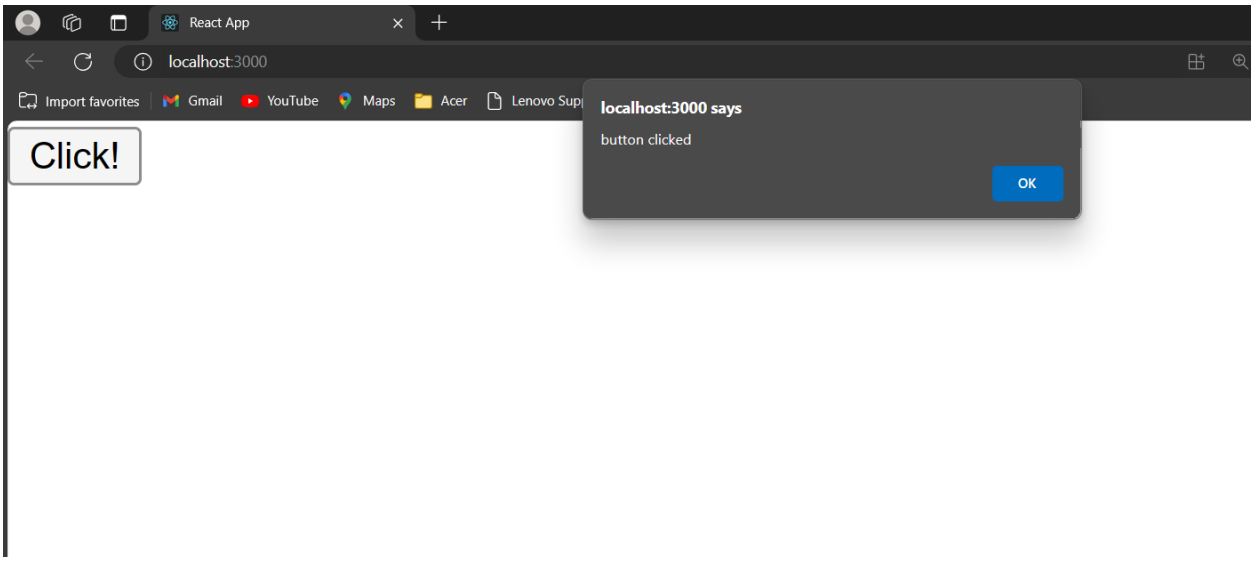
<button onClick={shoot}> Take the shot!</button>

Example Program for onClick() :

```
function App()
{
        function displayMessage()
        {
                alert("button clicked")
        }
        return
        (
        <div className=”App””>
                <button onClick={displayMessage}>Click</button>
        <div>
) }

export default App;
```

Output :

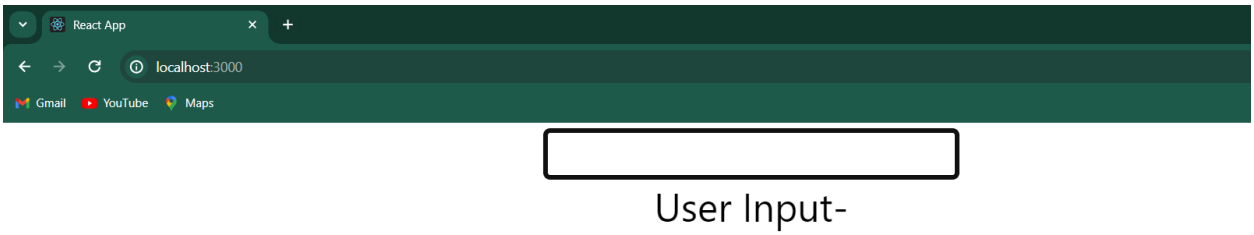


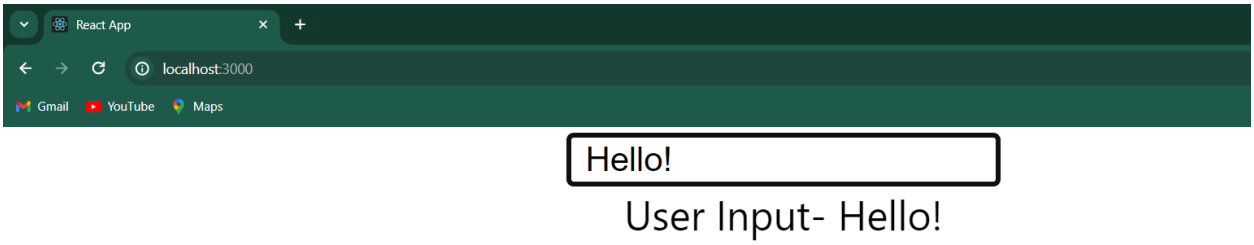


Example program for onChange() Event :

```
import React,{useState} from "react";
function App() {
  const[user,setUser]=useState("");
  const handler=(e)=>{
  setUser(e.target.value)
  }
  return(
    <div>
    <center>
    <input type="text" placeholder="username" value={user} onChange={handler}
/><br />User Input- {user}
    </center>
    </div>
  )
}
export default App;
```

Output :


React App
localhost:3000
Gmail
YouTube
Maps
User Input-


React App
localhost:3000
Gmail
YouTube
Maps
Hello!
User Input- Hello!

2.Explain onClick and onChange events with Example

A)

In React, there are several events available to interact with user actions. Some of the most commonly used events include:

1. onClick: Fires when an element is clicked.

2. onChange: Fires when the value of an input element changes (e.g., input, textarea, select).

available in standard HTML.Now, let's explain onClick and onChange events with examples:

1. onClick:

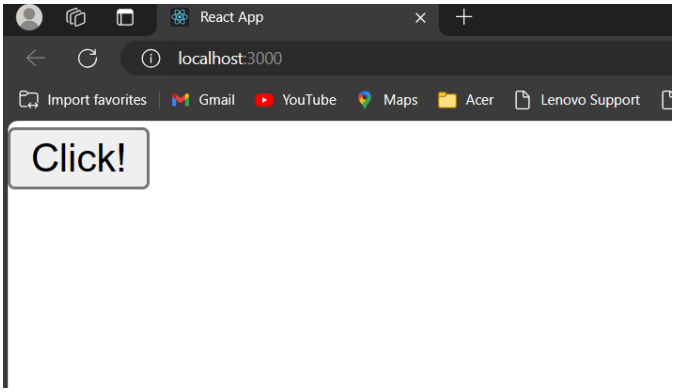
The onClick event is used to handle click events on elements like buttons, links, etc. It is triggered when the user clicks on the element.

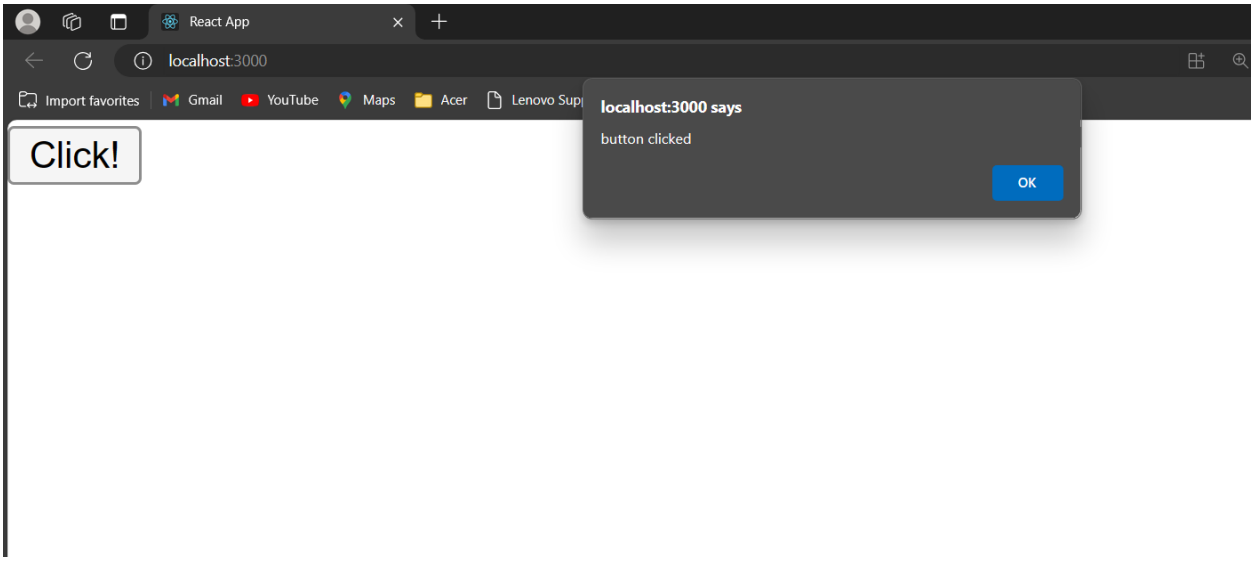
Example:

javascript

```
import React from 'react';

function ClickExample() {

// Function to handle click event

const handleClick = () => {

alert('Button clicked!');

};

return (

<div>

<button onClick={handleClick}>Click me</button>

</div>

);

}export default ClickExample;
```

Output :





In this example, when the button is clicked, the handleClick function is called, which displays an alert saying "Button clicked!".

2. onChange:

The onChange event is commonly used with form elements like input, textarea, and select to detect when their values change.

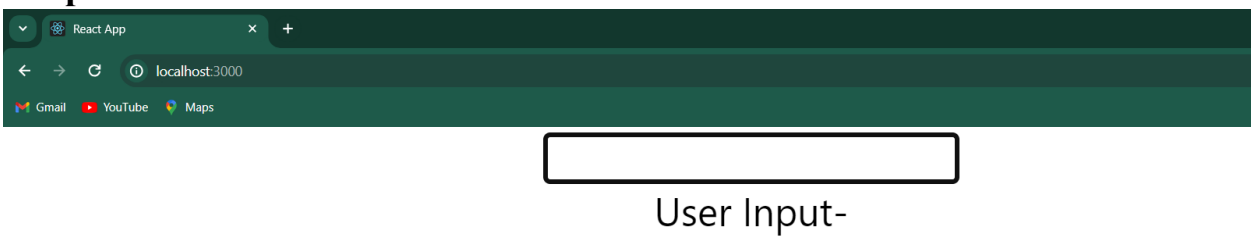
Example:

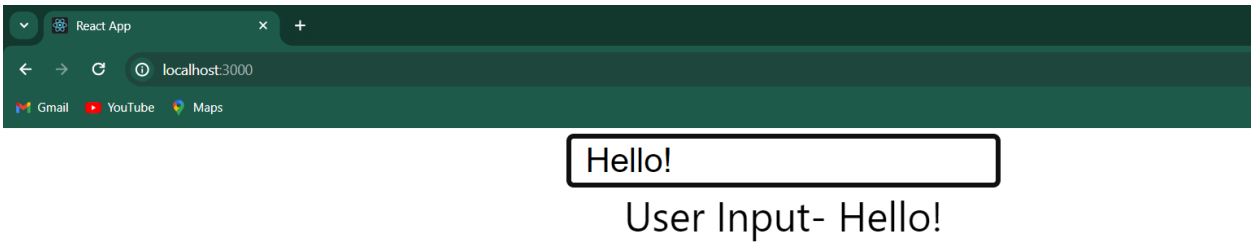
javascript

import React, { useState } from 'react';

```
function InputExample() {
// Initialize state for input value
const [value, setValue] = useState('');
// Function to handle input change
const handleChange = (event) => {
setValue(event.target.value);
};
return (
<div>
<input type="text" value={value} onChange={handleChange}  placeholder="Type something..." />
<p>You typed: {value}</p>
</div>
);
}
export default InputExample;
```

Output :





In this example, the handleChange function is called whenever the value of the input changes. It updates the value state with the new input value. The current value of the input is displayed below the input field.

These examples illustrate how to use onClick and onChange events in React to interact with user actions.

3.Explain Synthetic Events with Example

A)

Synthetic events are an abstraction layer on top of native browser events in React. They are cross-browser compatible and provide a consistent interface for handling events. When an event occurs in a React component, React first wraps it in a SyntheticEvent object before passing it to the event handler. This allows React to optimize event handling and provide additional features like event pooling and event delegation.

Example Program for it :

MyComponent.js

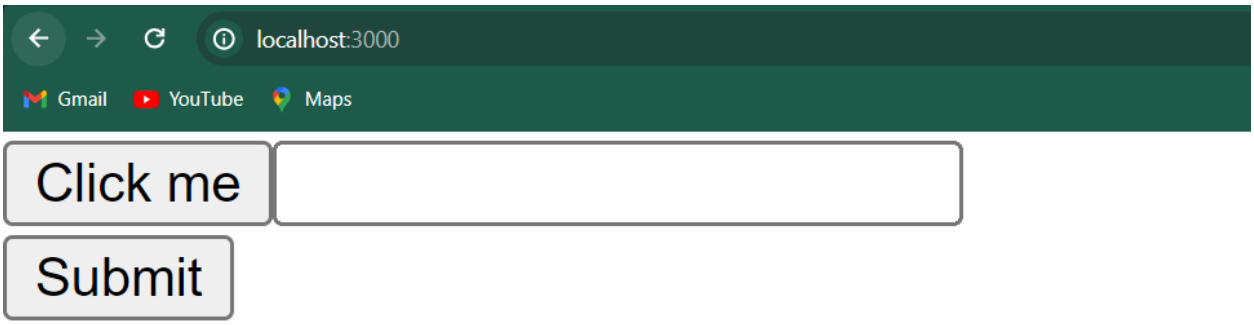
```
import React from 'react';

class MyComponent extends React.Component {
  handleClick = (e) => {
    e.preventDefault();
    console.log('Button clicked!');
  };
```

```
  handleChange = (e) => {
    console.log('Input value:', e.target.value);
  };
  handleSubmit = (e) => {
    e.preventDefault();
    console.log('Form submitted!');
  };
  render() {
    return (
      <div>
        {/* Handling click event */}
        <button onClick={this.handleClick}>Click me</button>

        {/* Handling change event */}
        <input type="text" onChange={this.handleChange} />

        {/* Handling form submission */}
        <form onSubmit={this.handleSubmit}>
          <input type="submit" value="Submit" />
        </form>
      </div>
    );
  }
}
export default MyComponent;
```

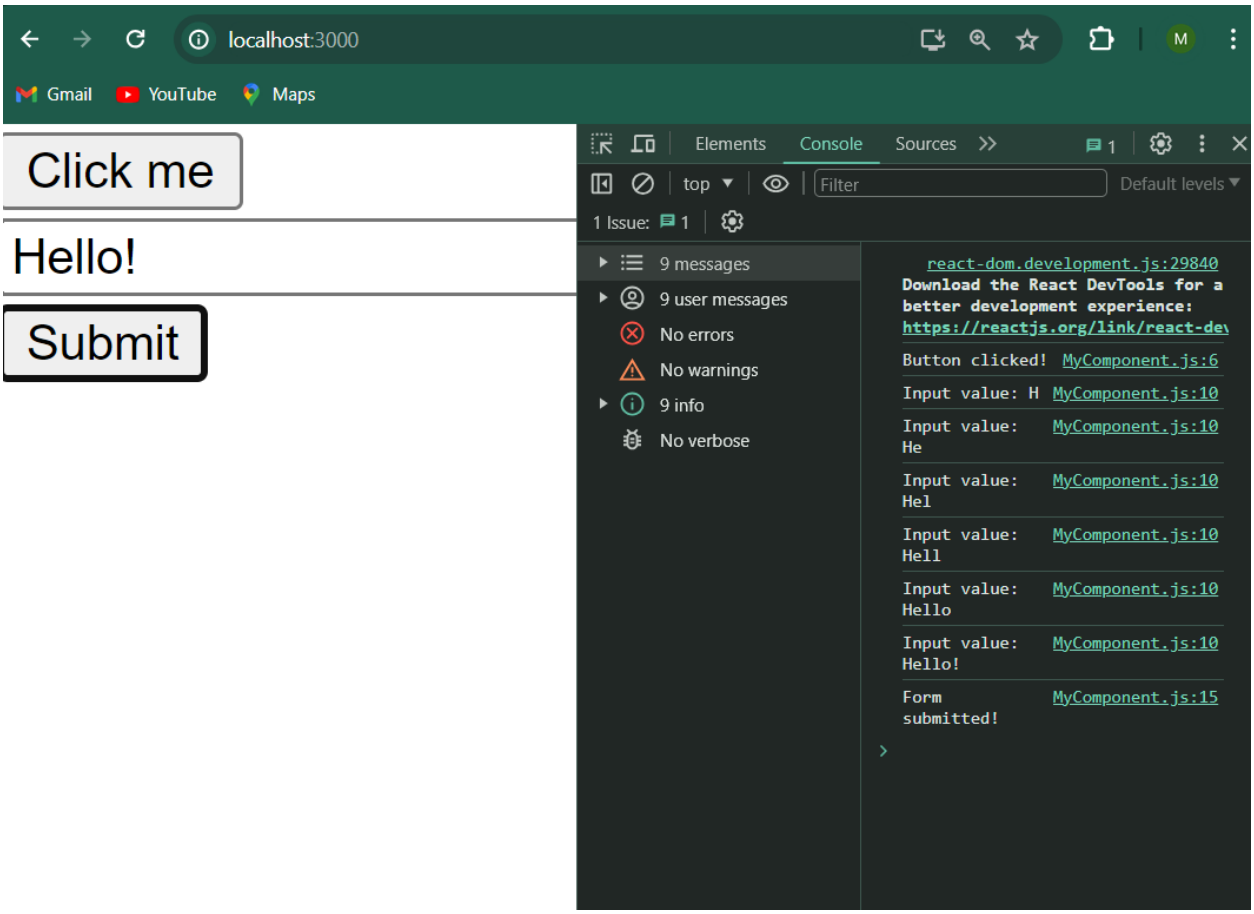
App.js :

```
import React,{useState} from "react";
import MyComponent from "./MyComponent";
function App() {
  return(
    <MyComponent/>
  )
}
export default App;
```

Output :



After clicking the button or submitting the form it will be shown in the console.



4.Explain about Conditional Rendering. Explain various ways of Conditional Rending with example.

A)

Conditional rendering in React refers to the ability to display different UI elements or components based on certain conditions or state. This allows you to dynamically change what is rendered to the user, providing a more interactive and personalized user experience.

There are several ways to achieve conditional rendering in React:

1. Using if Statements:

You can use regular JavaScript if statements within the render() method of a class

component to conditionally render components.javascript

```
import React, { Component } from 'react';
class ConditionalRenderingExample extends Component {
render() {
if (this.props.isLoggedIn) {
return <p>Welcome, User!</p>;
} else {
return <p>Please log in.</p>;
}
}
}
export default ConditionalRenderingExample;''
```

Output :

Please log in.

2. Using Ternary Operator:

Ternary operator (? :) is a concise way to conditionally render components.

Javascript example :

```
import React from 'react';
function ConditionalRenderingExample({ isLoggedIn }) {
return (
<div>
{isLoggedIn ? <p>Welcome, User!</p> : <p>Please log in.</p>}
</div>
);
}
export default ConditionalRenderingExample;
```

Output :

Please log in.

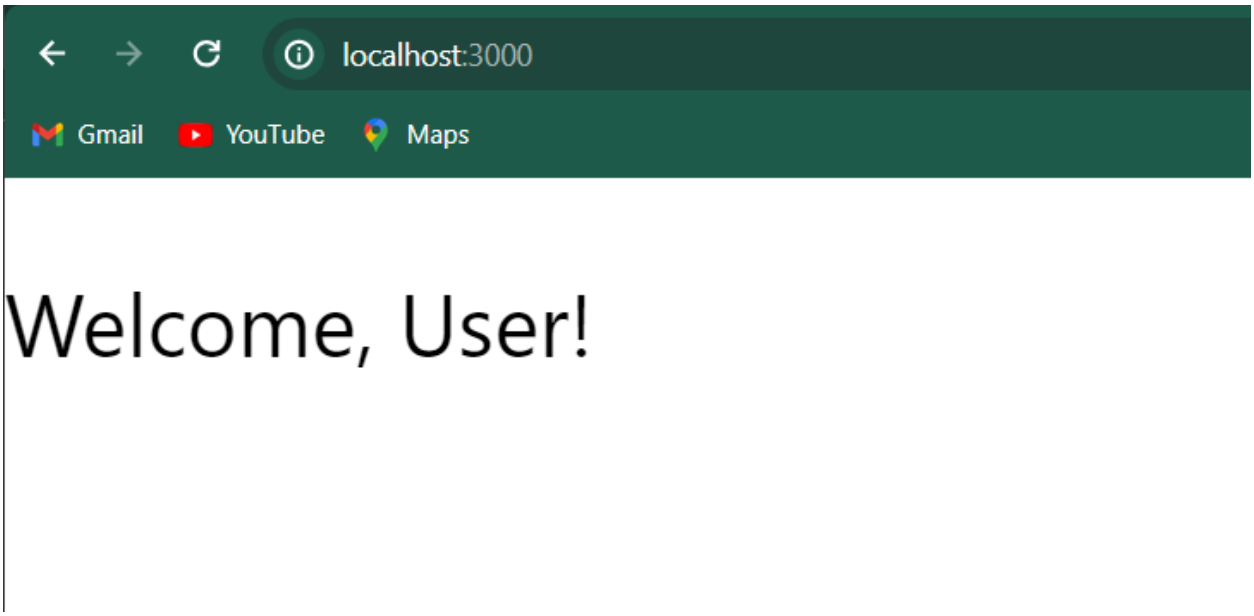
3. Using Logical && Operator:

The && operator can be used for simple conditional rendering. If the condition before && evaluates to true, the element after && is rendered.

Javascript example

```
import React from 'react';

function ConditionalRenderingExample() {

    const isLoggedIn = true;

return (

<div>

{isLoggedIn && <p>Welcome, User!</p>}

</div>);

}

export default ConditionalRenderingExample;
```

Output :



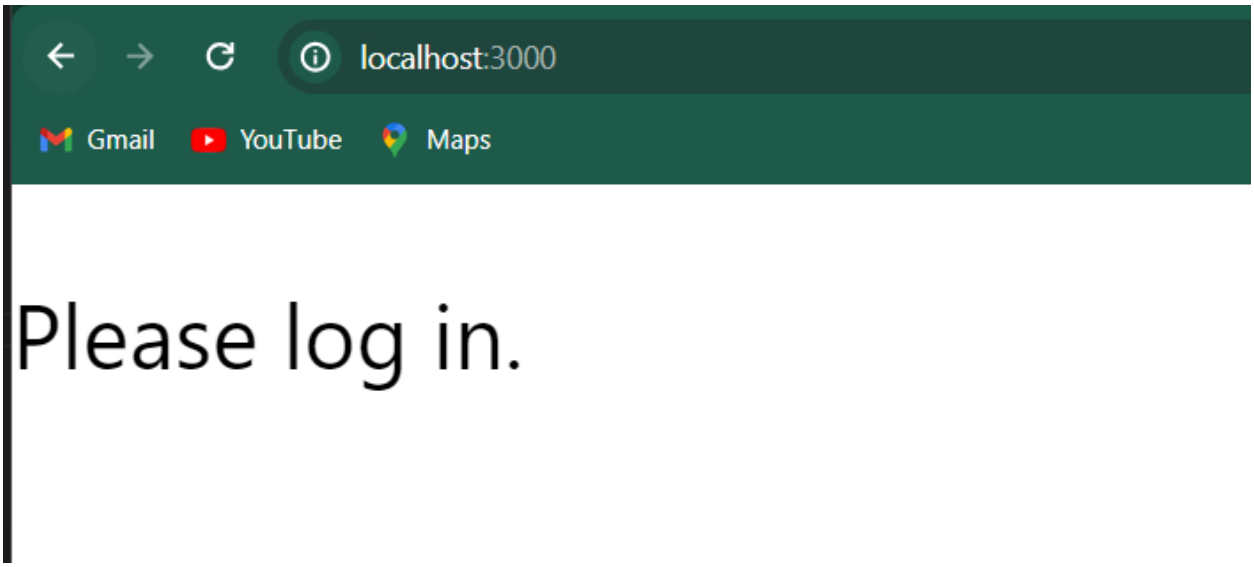
4. Using Switch Statement:

You can use a switch statement for more complex conditional rendering scenarios.

Javascript.

```
import React from 'react';
function ConditionalRenderingExample({ userType }) {
switch (userType) {
case 'admin':
return <p>Welcome, Admin!</p>;
case 'user':
return <p>Welcome, User!</p>;
default:
return <p>Please log in.</p>;
}
}
export default ConditionalRenderingExample;
```

Output :



5.Using If-Else in JSX (Not Recommended):

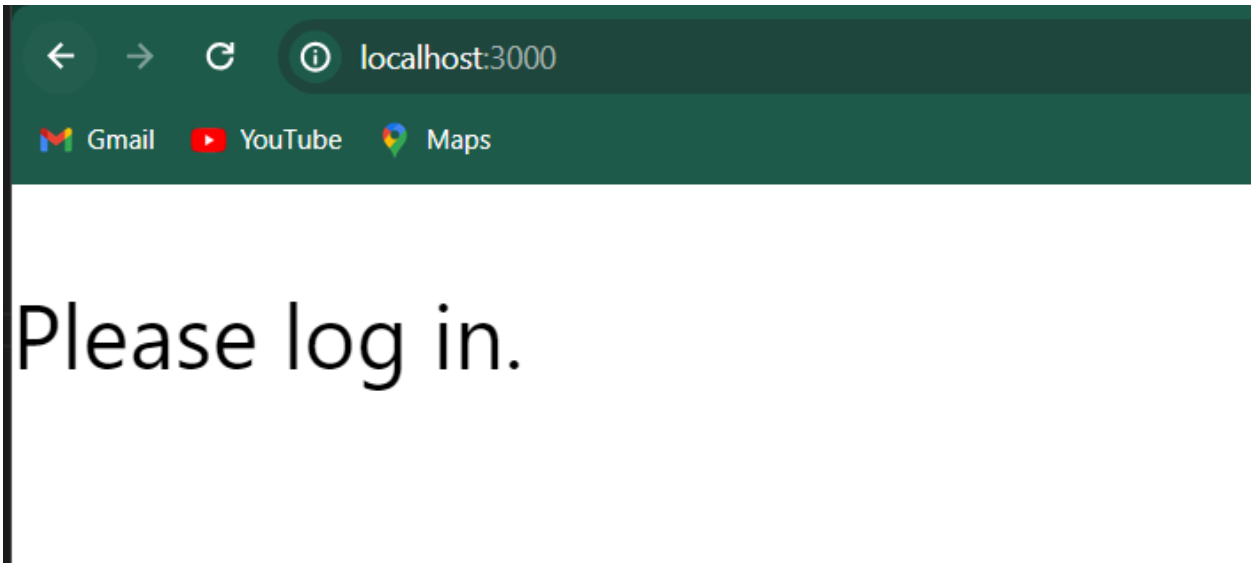
Though it's not commonly used, you can also use if-else statements directly within JSX by

enclosing them in curly braces {}.

Javascript example :

```
import React from 'react';

function ConditionalRenderingExample({ isLoggedIn }) {

if (isLoggedIn) {

return <p>Welcome, User!</p>;

} else {

return <p>Please log in.</p>;

}

}

export default ConditionalRenderingExample;
```

Output :



Each of these methods has its use case, and the choice depends on the complexity and readability of your code. Generally, using ternary operator or logical && operator is recommended for simple conditional rendering, while switch statements or if statements may be more suitable for complex scenarios.

5.Provide examples of using the logical && operator for simple and concise conditional rendering

A)

In React, the logical AND (`&&`) operator can be used for simple and concise conditional rendering. It allows you to conditionally render a component or an element based on a boolean expression without the need for an `if` statement or ternary operator.

Here are examples demonstrating the usage of the `&&` operator for conditional rendering:

Simple Conditional Rendering :

```
import React from "react";

function ConditionalRenderingExample({IsLoggedIn}){
   IsLoggedIn = true;
   return(
      <div>
         {IsLoggedIn&&<h1>Welcome</h1>}
      </div>
   )
}
export default ConditionalRenderingExample;
```

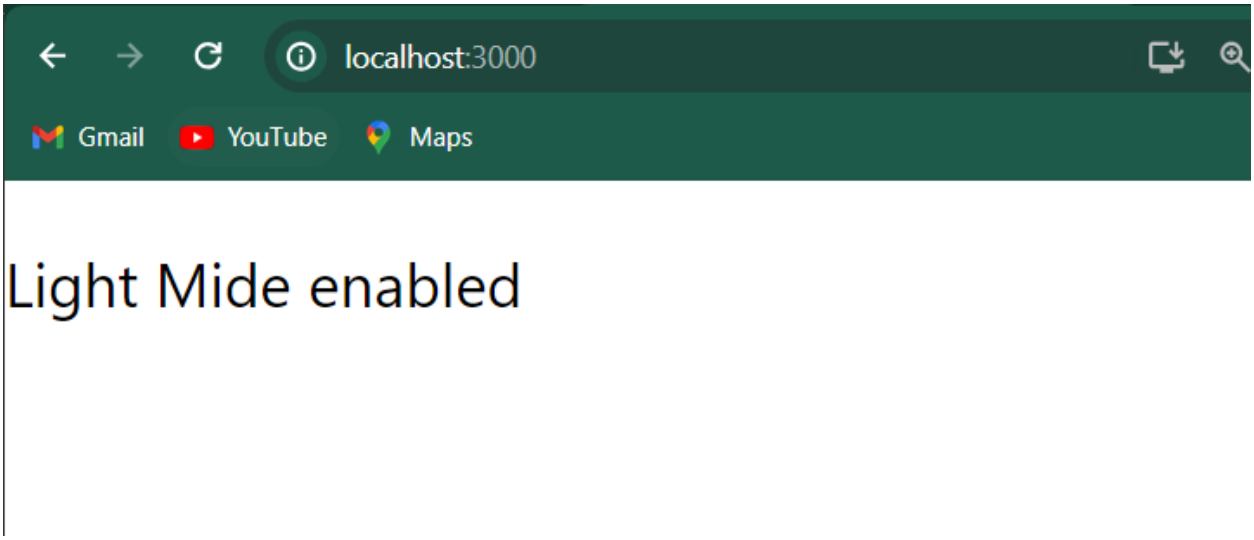
Output :

Welcome

Concise Conditional Rendering :

```
import React from "react";
function ConditionalRenderingExample({isDarkMode}){
   return(
      <div>
         {isDarkMode && <p> Dark Mode enabled</p>}
         {!isDarkMode &&<p>Light Mide enabled</p>}
      </div>
   )
}
export default ConditionalRenderingExample;
```

Output :



In this example, based on the value of the `isDarkMode` prop, either "Dark Mode Enabled" or "Light Mode Enabled" will be rendered. The `&&` operator is used to conditionally render each paragraph based on the value of `isDarkMode`.

These examples demonstrate how to use the `&&` operator for simple and concise conditional rendering in React. It's a clean and efficient way to conditionally render components or elements based on boolean expressions.

6)Explain the different ways to Style components in ReactJS.

A. In React, styling can be applied in several ways, each with its own advantages and use cases. Here are the most common methods:

• Inline Styles
• CSS-in-JS Libraries
• CSS Stylesheets
• CSS Modules

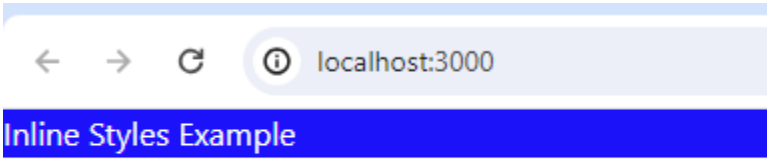
Inline Styles: You can apply styles directly to React elements using the style attribute. This involves passing a JavaScript object where the keys are CSS properties and the values are their corresponding values. Inline styles are scoped to the component and are useful for dynamic styling based on props or state.

Example:-

File Name:App.js

```
const App = () => {
  const dynamicColor = 'blue’;
  const styles = {
    backgroundColor: dynamicColor,
    fontSize: '16px',color : ‘white’
 };
return <div style={styles}>Inline Styles Example</div>;
};
```

OutPut:



CSS-in-JS Libraries: There are libraries like Styled Components, Emotion, and Aphrodite that allow you to write CSS directly in your JavaScript code.

These libraries often offer features like scoped styles, dynamic styling, and theming

File Name:App.js

```
import styled from 'styled-components';
const StyledDiv = styled.div`
  background-color: blue;
  font-size: 40px;
  color: white;
  padding: 25px 50px 75px 100px;
  height: 200px;
  width: 50%;
  border: 2px solid red;
```

```
  outline-style: dotted;
  outline-color: green;
`;
const App = () => {
return<StyledDiv>Styled using Styled Components</StyledDiv>;
 };
```

OutPut:

CSS Stylesheets:

write styling in a separate file for your React application, and save the file with a .css extension. Now, you can import this file in your application.

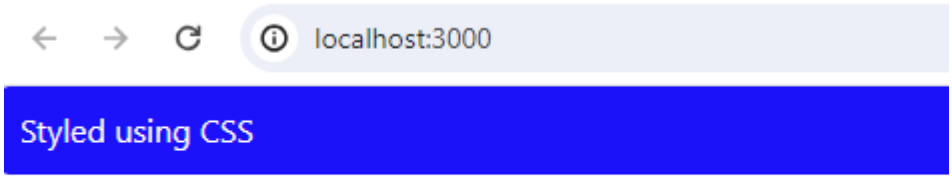
File Name:App.js

```
import './styles.css'; const App = () => {

return<div className="styledDiv">Styled using CSS</div>;

};
```

File Name:styles.css

```
.styledDiv {
background-color: blue;
font-size: 16px;
padding: 10px;
border-radius: 5px;
color: white;
}
```

OutPut:



CSS Modules:

CSS Modules allow you to write CSS files where class names are scoped locally to the component. This prevents class name clashes and makes it easier to maintain styles for individual components.

Example:-

File Name:App.js

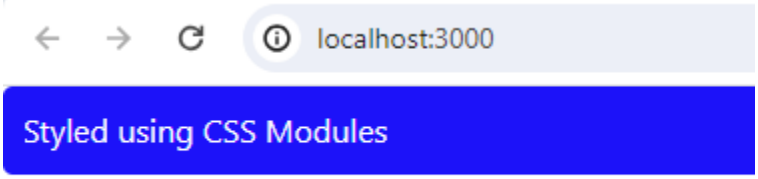
```
import styles from './Component.module.css';
const App = () => {
return <div className = {styles.styledDiv} >Styled using CSS Modules</div>;
};
```

File Name:

Component.module.css

```
.styledDiv {
background-color: blue;
font-size: 16px;
padding: 10px;
border-radius: 5px;
color: white;
}
```

Output:



7)Explain inline Styles in React with Example.

B. Inline styles: These are styles applied directly to a React component for a specific element. They are applied using the key called ‘style’ within the tag. Inline styles focus on the component and are useful for dynamically styling based on props and states.These inline styles are used for specific cases to keep React components clean and maintainable.

Syntax for inline style:

<element style={{style_name1: val1, style_name2: val2, …}} />

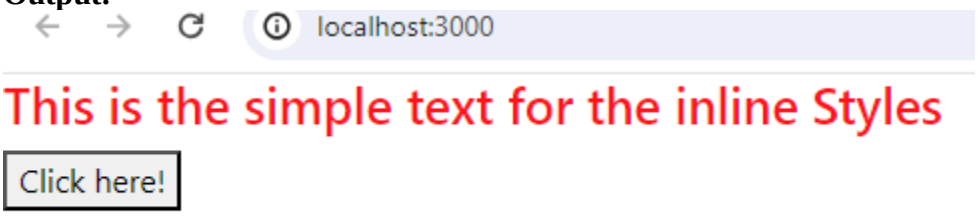
In the above, the "element" is generally known as the HTML tag that needs to be styled, and "style_name1" is for naming the style, such as color, background, padding, size, and others. Whereas "val1" provides the value, like color:red, background:blue, padding:20px 30px 40px, and others.
Since we are using React, we use {{….}} for a JavaScript object containing CSS property-value pairs. We can add more property-value pairs separated by commas to style multiple properties.

Inline styles can make code harder to read and maintain, especially for complex styles. The styles are calculated based on component props or states and applied directly.

Example code:
File Name:App.js

```
Function App() {
Const dynamicColor = 'blue';
Const styles ={
backrgroundColor:dynamicColor,
fontSize:20,
Color:'red'
}
return(
<>
<h1 style={styles}>This is the simple text for the inline Styles</h1>
<button onClick={()=>document.write('button is clicked!')} > Click here!</button>
</>
)
}export default App;
```

Output:



The above example illustrates inline styles within the function App, where variables for inline styles like dynamicColor and fontSize are declared. These are then used within the return tag, enclosed in a fragment <></>, along with an <h1> tag containing the text.

8)What is a React Router? Why is React Router used in React applications?

C. Routing is a process in which a user is directed to different pages based on their action or request. ReactJS Router is mainly used for developing Single Page Web Applications. React Router is used to define multiple routes in the application. When a user types a specific URL into the browser, and if this URL path matches any 'route' inside the router file, the user will be redirected to that particular route.

React Router is a standard library system built on top of the React and used to create routing in the React application using React Router Package. It provides the synchronous URL on the browser with data that will be displayed on the web page. It maintains the standard structure and behavior of the application and is mainly used for developing single page web applications.

Why React Router?

React Router is a JavaScript framework that lets us handle client and server-side routing in React applications. It enables the creation of single-page web or mobile apps that allow navigating without refreshing the page. It also allows us to use browser history features while preserving the right application view.

React Router plays an important role to display multiple views in a single page application. Without React Router, it is not possible to display multiple views in React applications. Most of the social media websites like Facebook, Instagram uses React Router for rendering multiple views.

React contains three different packages for routing. These are:

- **react-router:** It provides the core routing components and functions for the React Router applications.
- **react-router-native:** It is used for mobile applications.
- **react-router-dom:** It is used for web applications design.

Steps to Use React Router

Step 1: Initialize a react project. Check this post for setting up React app

Step 2: Install react-router in your application write the following command in your terminal

npm i react-router-dom

Step 3: Importing React Router

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

React Router Components

The Main Components of React Router are:

BrowserRouter: BrowserRouter is a router implementation that uses the HTML5 history API(pushState, replaceState, and the popstate event) to keep your UI in sync with the URL. It is the parent component that is used to store all of the other components.

Routes: It's a new component introduced in the v6 and an upgrade of the component. The main advantages of Routes over Switch are:

- Relative s and s
- Routes are chosen based on the best match instead of being traversed in order.

Route: Route is the conditionally show component that renders some UI when its path matches the current URL.

Link: The link component is used to create links to different routes and implement navigation around the application. It works like an HTML anchor tag.

Step to run the application: Open the terminal and type the following command.
npm start

Output: Open the browser and our project is shown in the URL **http://localhost:3000/.** Now, you can click on the links and navigate to different components. React Router keeps your application UI in sync with the URL.

9)Set up routing using React Router to create a simple multi-page application. Create routes for different pages (e.g., Home, About, Contact) and navigate between them.

A) React Router, is essential tool for building single-page applications (SPAs).

- As there is no inbuilt routing in React, the React JS Router enables routing support in React and navigation to different components in multi-page applications.
- It renders components for corresponding routes and assigned URLs.

React Router is a standard library for routing in React. It enables the navigation among views of various components in a React Application, allows changing the browser URL, and keeps the UI in sync with the URL. The application will contain three components the home component, the About component, and the contact component. We will use React Router to navigate between these components.

Program:

File Name: App.js

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

const App = () => {
  return (
```

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

export default App;
```

File Name: Home.js:

```
import React from "react";

function Home() {
  return (
    <div>
      <h1>Welcome to Home Component with Routing</h1>
      <h2>This is Home Page</h2>
    </div>
  );
}

export default Home;
```

File Name: Contact.js:

```
import React from "react";

function ContactUs() {
  return (
<div className={styles.styledDiv}>
            <h1>Welcome to Contact us Component with Routing</h1>
            <h2>This is Contact us Page </h2>
        </div>
  );
}
```

export default ContactUs;

FileName: Contact.module.css

```
.styledDiv {
    background-color: red;
    font-size: 16px;
    padding: 25px 50px 75px 100px;
    width: 50%;
    border-radius: 5px;
    color: green;
  }
```

File Name: About.js

```
import React from "react";
import './About.css';
function About() {
return(
        <div className={styles.styledDiv}>
            <h1>Welcome to About Component with Routing</h1>
            <h2>This is About Page </h2>
        </div>

  );
} export default About;
```

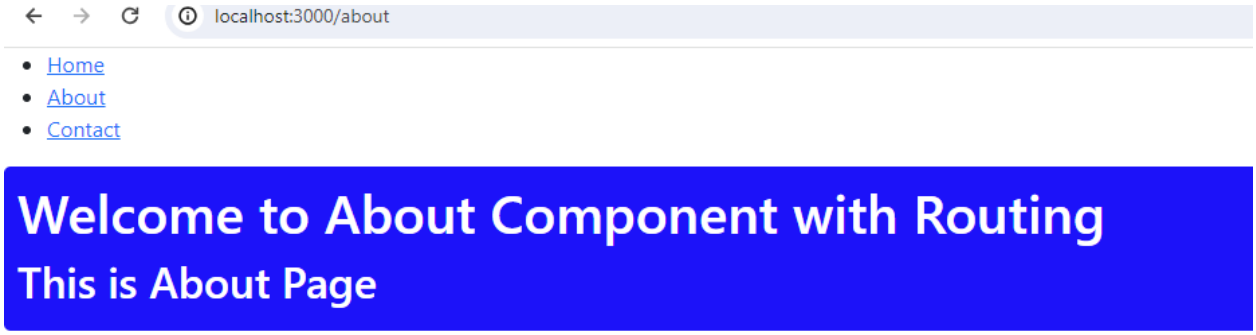
FileName: About.module.css

```
.styledDiv {
    background-color: blue;
    font-size: 16px;
    padding: 10px;
    border-radius: 5px;
    color: white;
  }
```

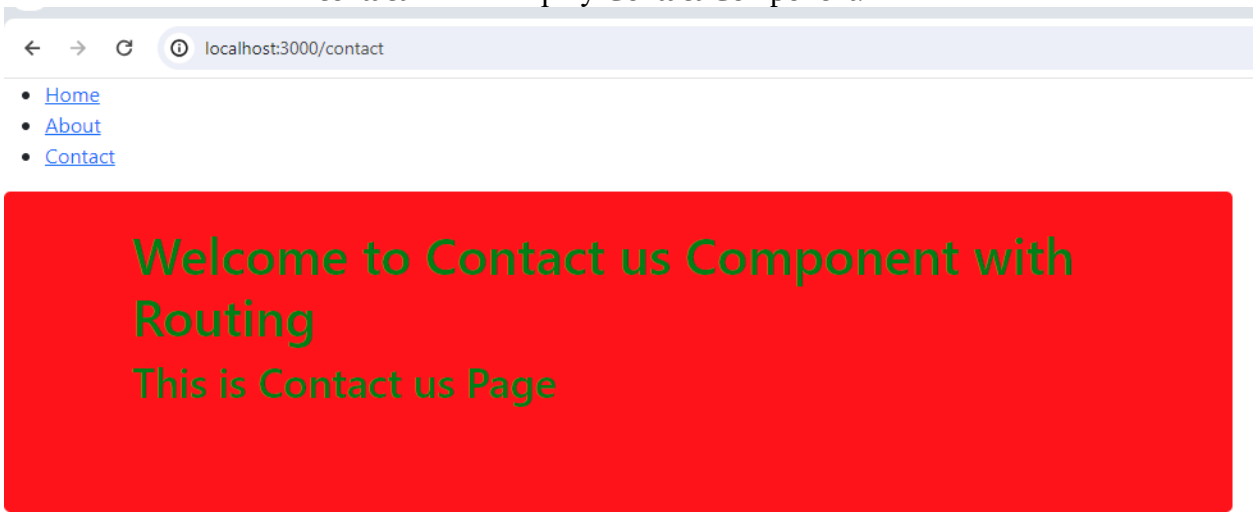
OutPut:

- ✓ When URL is '/' it will display Home Component.

✓ When URL is '/about' it will display About Component.



✓ When URL is '/contact' it will display Contact Component.



10)Create a Card component that displays content inside a styled card. Use the Card component to display StudentList.

Card

Introduction

Cards contain content and actions about a single subject. React cards have become an essential part of modern web development. They are a great way to present information in an organized and

visually appealing manner. React cards are versatile components that can be used in a variety of contexts, from e-commerce sites to social media feeds.
React cards are reusable components that display content in a structured and organized manner. They are used to showcase information in a visually appealing and user-friendly way.
React cards provide a modular and reusable way of displaying information on a webpage. They also make it easy to style and customize the layout and content of each card.
Cards are surfaces that display content and actions on a single topic. The Material UI Card component includes several complementary utility components to handle various use cases:

- Card: a surface-level container for grouping related components.
- Card Content: the wrapper for the Card content.
- Card Header: an optional wrapper for the Card header.
- Card Media: an optional container for displaying background images and gradient layers behind the Card Content.
- Card Actions: an optional wrapper that groups a set of buttons.
- Card Action Area: an optional wrapper that allows users to interact with the specified area of the Card.

Create React cards:

1. **First, import the necessary React and CSS modules:**

```
import React from "react";

import "./Card.css";
```

2. **Create a new component for the card:**

```
function Card(props) {

  return <div className="card">{props.children}</div>;

}
```

3. Define the CSS styles for the card in a separate CSS file (in this example, named Card.css):

```
.card {
  border: 1px solid #ddd;
  border-radius: 8px;
  padding: 20px;
  margin-bottom: 20px;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}
```

4.Finally, use the new Card component in your application:

```
function App() {
  return (
    <div className="App">
      <Card>
        <h2>Title</h2>
        <p>Description</p>
      </Card>
    </div>
  );
}
```

Example:

FileName:card.js

```
import React from 'react';
import './card.css'; // Import your CSS file for styling

// Card component
const Card = ({ children }) => {
  return (
    <div className="card">
      {children}
    </div>
  );
};

// StudentList component
const StudentList = () => {
  // Sample student data
  const students = [
    { id: 1, name: 'John Doe', age: 20, grade: 'A' },
    { id: 2, name: 'Jane Smith', age: 21, grade: 'B' },
    { id: 3, name: 'Alex Johnson', age: 19, grade: 'C' },
  ];

  return (
    <Card>
      <h2>Student List</h2>
      <ul>
        {students.map(student => (
          <li key={student.id}>
            <strong>Name:</strong> {student.name}<br />
            <strong>Age:</strong> {student.age}<br />
            <strong>Grade:</strong> {student.grade}
          </li>
        ))}
      </ul>
```

```
        </Card>
      );
    };

    export default StudentList;
```

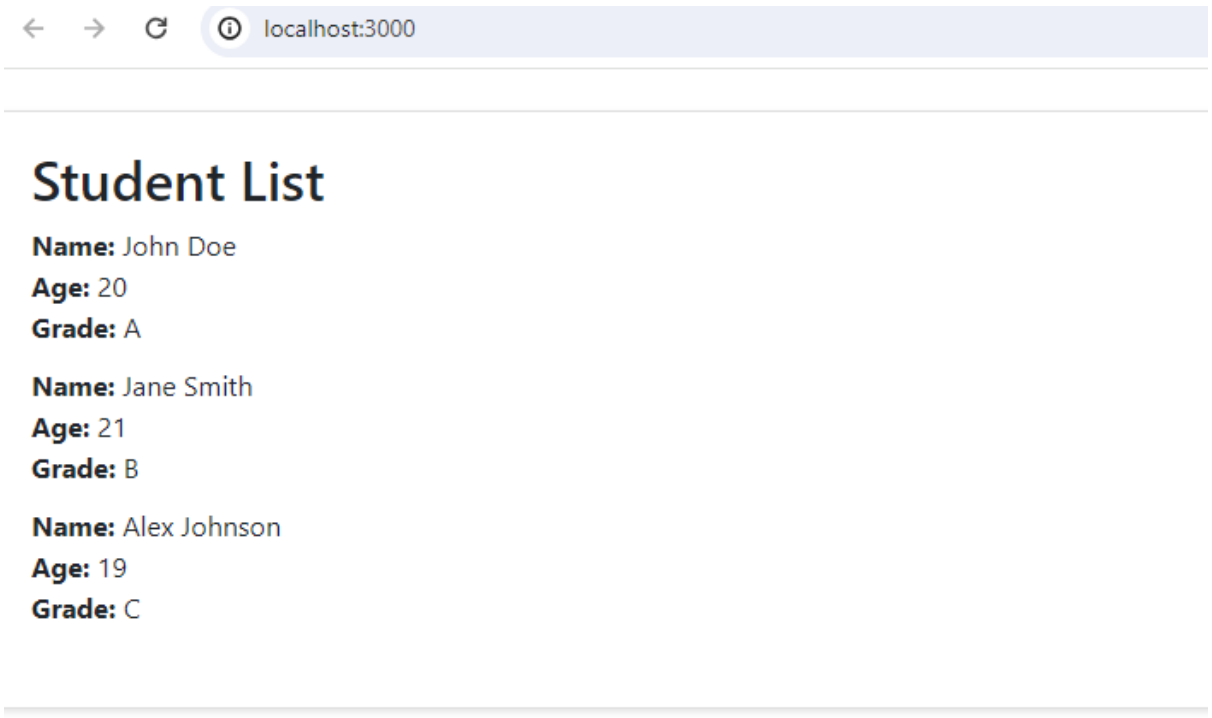
File Name:card.css

```
.card {
    border: 1px solid #ddd;
    border-radius: 8px;
    padding: 20px;
    margin-bottom: 20px;
    box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);}

  .card h2 {
    margin-top: 0;

  }

  .card ul {
    list-style: none;
    padding: 0;
  }

  .card li {
    margin-bottom: 10px;
  }
```

File Name:App.js

```
import StudentList from "./card";
function App()  {

       return  (<>

                    <StudentList />
                     </>)
};
export default App;
```

OutPut:



MERN STACK WEB DEVELOPMENT

UNIT-3

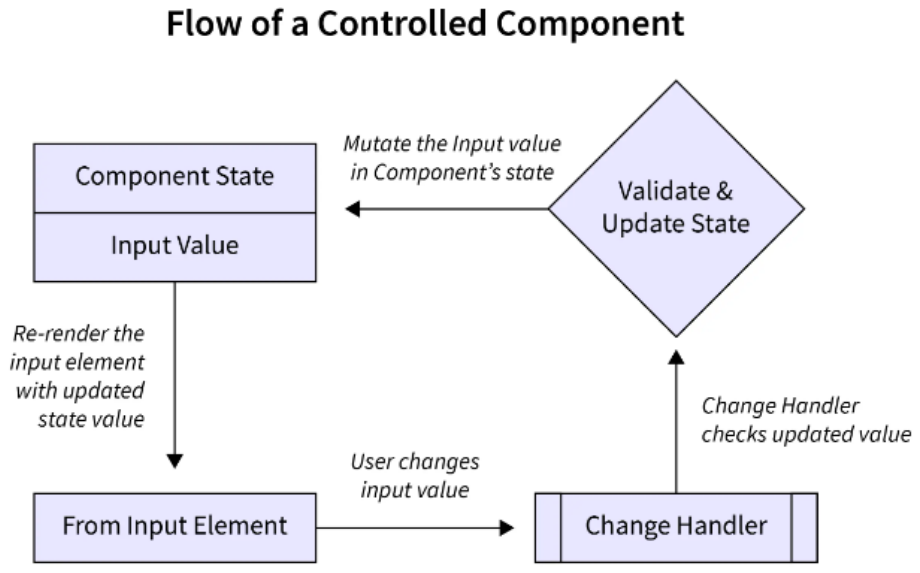
1) Explain about Controlled Components in ReactJS with an example.

Ans:

- A component in react is referred to as controlled when we let react control the component for us.
- It means that the component controls the input form, and all of its changes are completely driven by event handlers like setState(). Also, the component controls the render and keeps the data of form in the component state.
- The controlled component react takes all the values using props and reports the changes by using the callbacks.
- When we link the component state with the HTML elements, we are following "a single source of truth" as described by the documentation of the react.
- This means that react tracks all the changes with the help of the internal state and re-renders the component whenever there is a change.

Advantages:

- The instant validation check is one of the major benefits of using the controlled component over the uncontrolled component in react.
- we can access the input value at every time with the help of react state.



Ex:

```
import React, { useState } from 'react';
function ControlledInput() {
  const [value, setValue] = useState('')
  const handleChange = (event) => {
    setValue(event.target.value);
  };
  return (
    <input
      type="text"
      value={value}
      onChange={handleChange}
    />
  );
}
export default ControlledInput;
```

App.js:

```
Import ControlledInput from ‘./ControlledInput’;
function App () {
<div>
</ControlledInput>
</div>
export default App;
```

Code Explanation:

- In this example, the input's value is controlled by the value state variable.
- The handleChange function updates the value state variable whenever the user types into the input. This is a simple example of a controlled component in React.

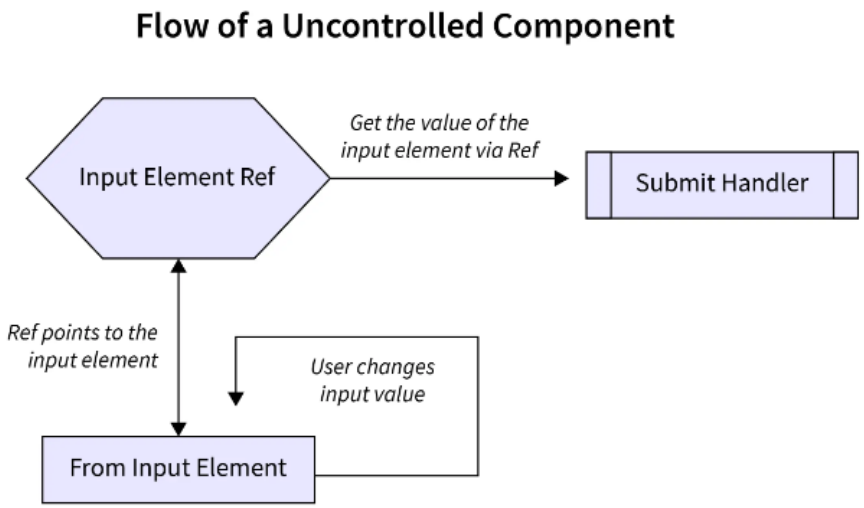
2) Explain about Uncontrolled Components in ReactJS with an example.

Ans:

- Uncontrolled component react does not use state. Thus uncontrolled components do not depend on any state of input elements or any event handler.
- This type of component does not care about real-time input changes.
- The data from the input fields are not stored in the react state, but the data is stored in the DOM itself.
- Instead of using any event handler setState() to update the value according to the changes by the user, we are provided with a ref to retrieve values from the DOM.
- Refs in react environment work like a pointer that provides us access to the DOM nodes

Advantages:

- Simplicity and Quick Prototyping.
- Reduced React Re-renders.
- Traditional HTML Form Behaviour.



Ex:

```
import React, { useRef } from 'react';
function UncontrolledInput() {
  const inputRef = useRef(null);
  const handleClick = () => {
    console.log(inputRef.current.value);
  };
  return (
    <>
      <input
        type="text"
        ref={inputRef}
      />
      <button      onClick={handleClick} >       Log Value      </button>
    </>
  );
}
export default UncontrolledInput;
```

App.js:

```
Import UncontrolledInput from ‘./UncontrolledInput’;
function App () {
<div>
</UncontrolledInput>
</div>
export default App;
```

Code explanation:

- In this example, the useRef hook is used to create a reference to the input element.
- The handleClick function logs the value of the input to the console when the button is clicked.

3) Briefly explain about how forms are created in ReactJS with an example.

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. Forms can perform many tasks that depend on the nature of your business requirements and logic such as authentication of the user, adding user, searching, filtering, booking, ordering, etc. A form can contain text fields, buttons, checkbox, radio button, etc.

Creating Form

React offers a stateful, reactive approach to build a form. The component rather than the DOM usually handles the React form. In React, the form is usually implemented by using controlled components.

There are mainly two types of form input in React.

1. Uncontrolled component
2. Controlled component

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. In other words, there is no need to write an event handler for every state update. You can use a ref to access the input field value of the form from the DOM.

Example

In this example, the code accepts a field **username** and **company name** in an uncontrolled component.

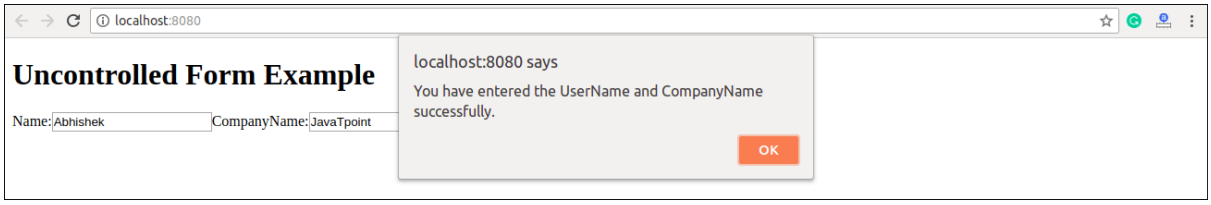
```
import React, { Component } from 'react';
class App extends React.Component {
 constructor(props) {
    super(props);
    this.updateSubmit = this.updateSubmit.bind(this);
    this.input = React.createRef();
 }
 updateSubmit(event) {
   alert('You have entered the UserName and CompanyName successfully.');
    event.preventDefault();
```

```
    }
    render() {
      return (
        <form onSubmit={this.updateSubmit}>
          <h1>Uncontrolled Form Example</h1>
        <input type="text" ref={this.input} />
          <label>Name:
         </label>
          <label>
              CompanyName:
              <input type="text" ref={this.input} />
          </label>
          <input type="submit" value="Submit" />
        </form>
      );
    }
}
    export default App;
```

Output:

When you execute the above code, you will see the following screen.

After filling the data in the field, you get the message that can be seen in the below screen.



Controlled Component

In HTML, form elements typically maintain their own state and update it according to the user input. In the controlled component, the input form element is handled by the component rather than the DOM. Here, the mutable state is kept in the state property and will be updated only with **setState()** method.

Controlled components have functions that govern the data passing into them on every **onChange event**, rather than grabbing the data only once, e.g., when you click a **submit button**. This data is then saved to state and updated with setState() method. This makes component have better control over the form elements and data.

A controlled component takes its current value through **props** and notifies the changes through **callbacks** like an onChange event. A parent component "controls" this changes by handling the callback and managing its own state and then passing the new values as props to the controlled component. It is also called as a "dumb component."

Example

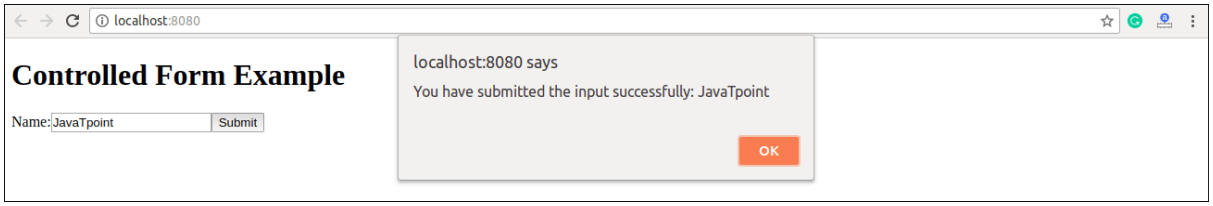
```
import React, { Component } from 'react';
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) {
    this.setState({value: event.target.value});
  }
  handleSubmit(event) {
    alert('You have submitted the input successfully: ' + this.state.value);
    event.preventDefault();
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <h1>Controlled Form Example</h1>
        <label>
```

```
            Name:
            <input type="text" value={this.state.value} onChange={this.handleChange
} />
          </label>
          <input type="submit" value="Submit" />
        </form>
      );
  }
}
export default App;
```

Output:

When you execute the above code, you will see the following screen.

After filling the data in the field, you get the message that can be seen in the below screen.



Q4) What is the purpose of forms in web development, and how are they used in React applications?

Ans:

A very common feature of web applications is the ability to collect and process user input through forms. Forms are essential since they allow users to input data and interact with a website.

Forms in web development serve the purpose of collecting and processing user input. They are crucial for creating interactive web applications where users can submit data, such as login information, registration details, search queries, or feedback. Forms allow users to interact with web pages dynamically, enabling actions like submitting data, performing searches, or making purchases.

In React applications, forms are used similarly to how they are used in traditional web development, but with a few differences due to React's component-based architecture and its approach to managing state. Here's how forms are typically used in React applications:

1. Component-Based Structure: React allows developers to create reusable components, and forms are often encapsulated within their own components. This modular approach makes it easier to manage form logic and state.

2. Controlled Components: In React, form inputs are often implemented as controlled components, where the value of the input field is controlled by the React component's state. This means that the input value is set by the component's state and any changes to the input value are handled by updating the state. This allows React to have full control over the form inputs, making it easier to validate and manipulate user input.

3. State Management: React components maintain their own state, so form data is typically stored in the component's state. As users interact with the form inputs, the component's state is updated accordingly. This allows React to re-render the component with the updated state, providing a responsive user interface.

4. Event Handling: React uses synthetic events to handle user interactions with form elements. Event handlers, such as `onChange` for input changes and `onSubmit` for form submissions, are attached to form elements to handle user actions. When an event occurs, React invokes the corresponding event handler function, allowing developers to update the component's state or perform other actions as needed.

5. Validation: Form validation in React can be implemented using various techniques, such as conditional rendering based on the validity of input values, custom validation functions, or third-party libraries. Validating form input helps ensure that users submit correct and valid data.

Overall, forms play a crucial role in React applications for capturing user input and enabling interaction, and React provides developers with the tools and patterns to effectively manage forms within their applications.

A very common feature of web applications is the ability to collect and process user input through forms. Forms are essential since they allow users to input data and interact with a website.

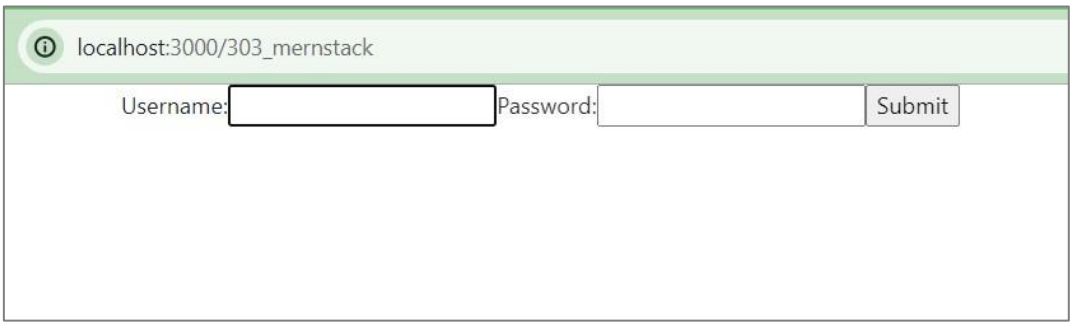
Ex:

Input:

```
import React from "react";
```

```
function App () {
  return (
    <form>
      <label>
        Username:
        <input type="text" name="username" />
      </label>
      <label>
        Password:
        <input type="password" name="password" />
      </label>
      <input type="submit" value="Submit" />
    </form>
  );}
```

Output:



5. Create a React App that implements a registration form component in React. The form should include fields for the user to enter their name, email, and password

Ans:

1. **Create a new file:** Create a new file in your project directory and name it **Registration.js.**
2. **Import React and useState:** At the top of the Registration.js file, import the React library and the useState hook from it with the following line of code:
   import React, { useState } from 'react';
3. **Create the RegistrationForm function:** Create a new function named RegistrationForm, initialize a state variable form with the useState hook. This state will hold the form field values

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

function RegistrationForm() {
    const [form, setForm] = useState({ name: '', email: '', password: '' });

    const handleChange = (e) => {
        setForm({
            ...form,
            [e.target.name]: e.target.value
        });
    };

    const handleSubmit = (e) => {
        e.preventDefault();
        console.log(form);
    };

    const formStyle = {
        display: 'flex',
        flexDirection: 'column',
        justifyContent: 'center',
        alignItems: 'center',
        height: '100vh'
    };

    return (
        <form onSubmit={handleSubmit} style={formStyle}>
            <label>
                Name:
                <input type="text" name="name" value={form.name} onChange={handleChange}
required />
            </label>
            <label>
```

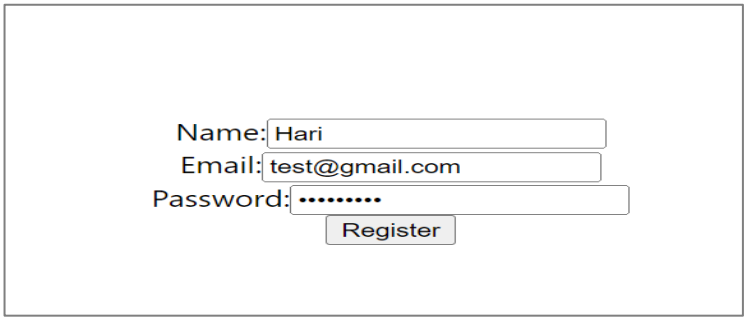
```
            Email:
            <input type="email" name="email" value={form.email}
onChange={handleChange} required />
          </label>
          <label>
            Password:
            <input type="password" name="password" value={form.password}
onChange={handleChange} required />
          </label>
          <input type="submit" value="Register" />
        </form>
    );
}

export default RegistrationForm;
```

4. **Import RegistrationForm in App.js:**

```
import logo from './logo.svg';
import './App.css';
import RegistrationForm from './components/Registration';



function App() {
  return (
    <div className="App">
      <RegistrationForm />
    </div>
  );
}

export default App;
```

5. **Run command :** npm start

Output:



Code Explanation:

- First line imports the React library and the useState hook from it. The useState hook is a function that lets you add React state to function components.
- we're using the useState hook to create a state variable form and a function setForm to update it. The initial state is an object with name, email, and password properties, all set to an empty string.
- handleChange is a function that gets called whenever the user types into any of the form fields. It updates the form state with the new value the user typed.
- handleSubmit function gets called when the user submits the form. It prevents the page from reloading (which is the default behavior when a form is submitted), and then logs the current state of the form to the console
- formStyle this is an object that contains CSS styles for the form. It's used to center the form on the screen and arrange the form fields vertically.

6. Develop a React App using functional component that implements creating a form component that allows users to submit data to an API endpoint as a POST request using React. The form should include Name and Email ID.

Ans:

Step-1: Creating a signup/register form component with name, email id and password.

Step-2: creating a json-server on local host to post the data which is collected from the form component.

Step-3: using the json-server api endpoint post the data to json-server

Step-1: FormComponent.js

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

```
import "./formcomponent.css";

const FormComponent = () => {
  const [formData, setFormData] = useState({
    password: "",
    username: "",
    email: "",
  });

  function handleSubmit(e) {
    e.preventDefault();
    console.log("Register Data");
    console.log(formData);

    fetch("http://localhost:3133/users", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(formData),
    })
      .then((response) => {
        if (response.ok) {
          alert("Registered Successfully");
        } else {
          throw new Error("Registration failed");
        }
      })
      .catch((error) => {
        console.error("Registration error:", error);
        alert("Registration failed. Please try again later.");
      });
  }
```

```
function handleChange(e) {
  const { name, value } = e.target;
  setFormData({ ...formData, [name]: value });
}

return (
  <div className="register-container">
    <h2 className="title">Register</h2>
    <form onSubmit={handleSubmit}>
      <div className="form-row">
        <div style={{ paddingBottom: "10px" }}>
          <input
            type="text"
            placeholder="Enter username"
            name="username"
            value={formData.username}
            onChange={handleChange}
            className="form-control input-field"
            required
          />
        </div>
        <div style={{ paddingBottom: "10px" }}>
          <input
            type="email"
            placeholder="Enter email"
            name="email"
            value={formData.email}
            onChange={handleChange}
            className="form-control input-field"
            required
          />
        </div>

        <div style={{ paddingBottom: "10px" }}>
```

```
          <input
            type="password"
            placeholder="Enter Password"
            name="password"
            value={formData.password}
            onChange={handleChange}
            className="form-control input-field"
            required
          />
        </div>

        <button type="submit" className="btn btn-success">
          Register
        </button>
      </div>
    </form>
  </div>
 );
};
export default FormComponent;
```

formcomponent.css:

```
.register-container {
  max-width: 400px;
  margin: 100px auto;
  padding: 20px;
  border: 1px solid #ccc;
  border-radius: 5px;
  background-color: #fff;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
  text-align: center;
}
```

```
.title {
  margin-bottom: 50px;
}
.input-field {
  margin-bottom: 20px;
}
```

Step-2: Setup json-server

To install json-server, run command:

"npm install json-server" on terminal

Creating a db.json file to save the data.

db.json:

```
{
  "users": [
    {
      "username": "",
      "password": "",
      "email": ""
    }
  ]
}
```

Step-3: using endpoint

To run the json-server, run the below command on terminal:

"npx json-server --watch db.json --port 3133"

On successfully setting up the server it will give a endpoint of

"http://localhost:3133/users".

Using above endpoint in formcomponent.js, fetch method api url we can post the data to api url.

Output

Fig-1: initial form component page

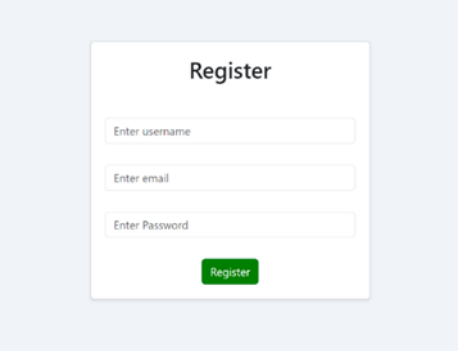


Fig-2: entering data into form

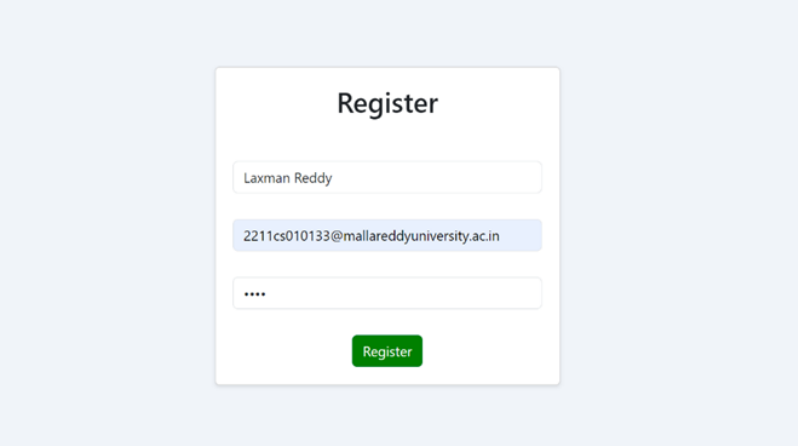


Fig-3: if the data enter successfully it will give an alert message of “Registered successfully”

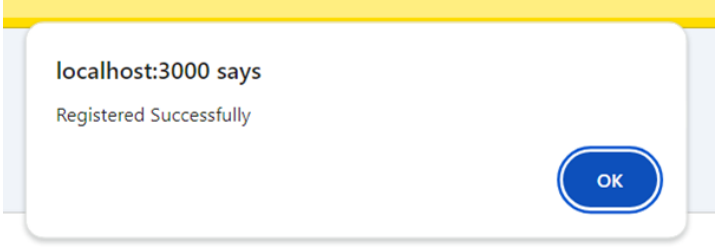


Fig-4: if any error occur while registration it will give an alert message of “Registered Failed. Please try again later.”

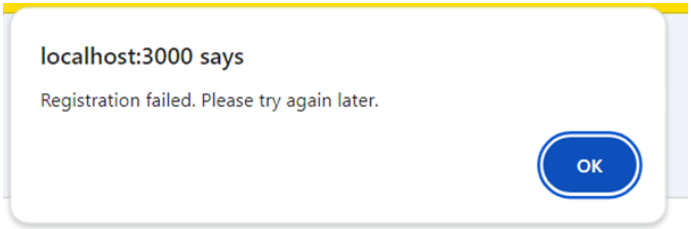


Fig-5: registered data is successfully stored in api url of

http://localhost:3133/users

7. Build a React component that async fetches data from a given API endpoint and renders it as a list. API endpoint returns an array of objects, display id and name from objects as a list.

API Endpoint: https://jsonplaceholder.typicode.com/users.

UserList.js:

```
import React, { useState, useEffect } from "react";
const UserList = () => {
  const [users, setUsers] = useState([]);
  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then((response) => response.json())
      .then((data) => {
        setUsers(data);
      })
      .catch((error) => {
        console.error("Error fetching data:", error);
      });
  }, []);
```

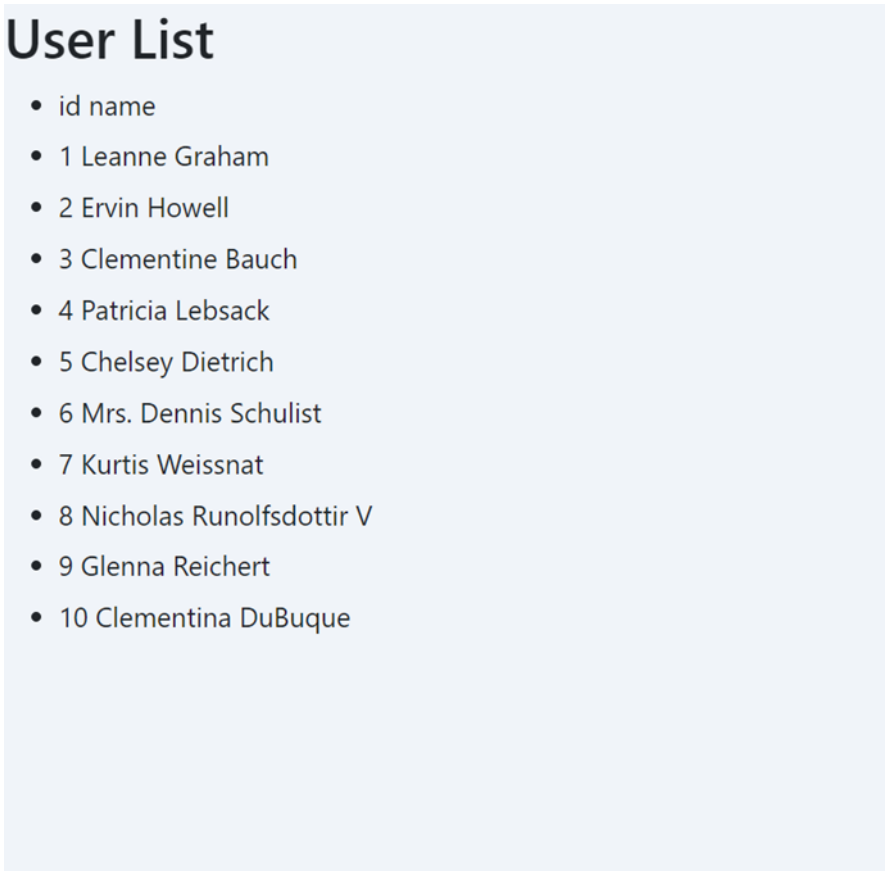
```
  return (
    <div>
      <h2>User List</h2>
      <ul>
        <li>id name</li>
        {users.map((user) => (
          <li key={user.id}>
            {user.id}
            {"              "}
            {user.name}
          </li>
        ))}
      </ul>
    </div>
  );
};

export default UserList;
```

APIEndpoint: https://jsonplaceholder.typicode.com/users contains:


```
[
  {
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
      "street": "Kulas Light",
      "suite": "Apt. 556",
      "city": "Gwenborough",
      "zipcode": "92998-3874",
      "geo": {
        "lat": "-37.3159",
        "lng": "81.1496"
      }
    },
    "phone": "1-770-736-8031 x56442",
    "website": "hildegard.org",
    "company": {
      "name": "Romaguera-Crona",
      "catchPhrase": "Multi-layered client-server neural-net",
      "bs": "harness real-time e-markets"
    }
  },
```


Each collection have above key value pairs

Output:



8) **Describe the process of form validation in React, including both client-side and server-side validation.**

Ans: Form validation in React typically involves both client-side and server-side validation to ensure data integrity and security.

Client-Side Validation:

- State Management: Use React state (or libraries like Redux) to store form input values and validation errors.
- Input Handling: Attach onChange handlers to input fields. These capture user input and update the state with the new values.
- Validation Logic: Implement validation logic within the handlers or separate validation functions. This logic checks the input against defined rules (e.g., required field, email format).
- Error State: If validation fails, update the state to include error messages for the specific fields.

Example:

```
const handleSubmit = (e) => {
    e.preventDefault();
    if(true){
      fetch("http://localhost:8080/user").then(res=>{
        return res.json();
      }).then((response)=>{
        response.map((user)=>{
          if(user.email===username){
            if(user.password === password){
            alert('Login Suceessfully');
              navigate('/VolunteerHome');
             }
          }
        })
      }).catch((err)=>{
        alert('Login Failed due to:'+err.message);
      })
  }
};
```

This code defines a function called handleSubmit that is likely meant to handle form submissions:

- e.preventDefault();: This line prevents the default behavior of form submission, which would typically cause the page to reload.
- if(true) { ... }: This if statement doesn't actually conditionally execute any code because it always evaluates to true. It seems like a placeholder for where you might put more complex logic in the future.
- Inside the if block, there's a fetch request to "http://localhost:8080/user". This fetches data from the specified URL, presumably retrieving user information.
- The fetch request is followed by a .then() block, which processes the response from the server. The response is expected to be in JSON format, so the first .then() block converts it to JSON.

- Another .then() block handles the JSON response data. It appears to iterate over each user in the response using the map function.
- Inside the map function, it checks if the user's email matches the username variable provided elsewhere in the code. If it does, it then checks if the user's password matches the password variable provided elsewhere in the code.
- If both the email and password match, it displays an alert saying "Login Successfully" and navigates to the "/Home" route. This likely means that the login was successful and redirects the user to a home page or dashboard for volunteers.
- If there is any error during the fetch request or processing of the response, it catches the error in a .catch() block and displays an alert with the error message.

Server-side Validation:

- Form Submission: When the user submits the form, the data is sent to the server (backend).
- Server-side Validation: The server performs its own validation checks on the received data. This can involve more complex rules or database checks not feasible on the client-side.

Ex:

```
var express=require('express');
var mongoose=require('mongoose');
var app=express();
const PORT=8000;
const URI="mongodb://localhost:27017/users";
mongoose.connect(URI)
.then(()=>{
   console.log("Database connected Successfully....");
})
.catch((error)=>{
   console.log("Failed to connect mongoDB", error);
})
app.post('/login', async (req, res) => {
   const { username, password } = req.body;
   try {
      const user = await users.findOne({ username: username });
```

```
        console.log("Trying to Login with username:", user);
        if (user) {
            if (user.password === password) {
                res.json("Success");
                console.log("User Logined Successfully....")
            } else {
                res.json("Password is incorrect...");
            }
        } else {
            res.json("No record exists for this username...");
        }
    } catch (error) {
        console.error("Error:", error.message);
        res.status(500).json("An error occurred while processing your request...");
    }
});

app.listen(PORT,(error)=>{
    if(error){
        console.log("Failed to connect server");
    }
    else{
        console.log(`Server started and Server running on ${PORT}`)
    }
})
```

- We import necessary modules: express for creating the server, mongoose for MongoDB interaction
- We create an Express app instance and define the port number and MongoDB URI
- We establish a connection to MongoDB using Mongoose. This connects to the database specified by the URI.
- We define a user schema with username and password fields. Then, we create a Mongoose model named User based on this schema.
- This defines a POST route /login for user authentication.

- It asynchronously tries to find a user with the provided username.
- If the user exists, it compare the provided password with the password stored in the database.
- It sends appropriate responses based on the authentication result.
- We start the Express server on the specified port.

9. What are controlled components in React forms, and why are they preferred over uncontrolled components?

Ans:

In React, there are two ways of handling form data:

1. Controlled Components
2. Uncontrolled Components

Controlled Components in React

- In React, a controlled component is a component where form elements derive their value from a React state.
- When a component is controlled, the value of form elements is stored in a state, and any changes made to the value are immediately reflected in the state.
- To create a controlled component, you need to use the value prop to set the value of form elements and the onChange event to handle changes made to the value.
- To create a controlled component, you need to use the value prop to set the value of form elements and the onChange event to handle changes made to the value.

Ex:

```
import {useState} from 'react';
 export default function  ControlledComponent()  {
      const  [inputValue, setInputValue] =  useState('');

      const  handleChange = (event) => {
            setInputValue(event.target.value);
      };
return  (
<form>
      <label>Input Value:
      <input  type="text"  value={inputValue} onChange={handleChange} />
      </label>
      <p>Input Value: {inputValue}</p>
</div>
)};
```

Output

UnControlled Components in React

Uncontrolled components in React refer to form elements whose state is not managed by React. Instead, their state is handled by the browser's DOM.

Controlled components are preferred over uncontrolled components in React for several reasons:

1. **Single Source of Truth:** Controlled components allow you to keep all form data in React state. This creates a single source of truth for your application's data, making it easier to manage and synchronize across components. In contrast, uncontrolled components maintain their own internal state, leading to potential data inconsistencies and harder-to-manage state.

2. **Immutable Data:** React state is immutable, meaning it cannot be directly modified. With controlled components, any changes to form data are handled through React state, ensuring that your data remains immutable and making it easier to track changes, especially in larger applications.

3. **Easier to Manage:** Controlled components provide a clear and explicit way to manage form data flow. You can easily see how data is passed between the form elements and React state, making your code more predictable and easier to debug.

4. **Testing:** Controlled components are easier to test compared to uncontrolled components. Since form data is stored in React state, you can easily write unit tests to simulate user interactions and validate how your components respond to different input values.

5. **Validation and Error Handling:** Controlled components make it straightforward to implement validation and error handling logic. You can validate the input value as the user types and provide immediate feedback if the input is invalid, ensuring a better user experience.

10) Discuss the use of the useState hook in managing form state in React.

Ans:

In React, useState is a special function that lets you add state to functional components. It provides a way to declare and manage state variables directly within a function component. It should be noted that one use of useState() can only be used to declare one state variable. It was introduced in version 16.8.

Importing the useState Hook

To import the useState hook, write the following code at the top level of your component

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

Structure of React useState hook

This hook takes some initial state and returns two value. The first value contains the state and the second value is a function that updates the state. The value passed in useState will be treated as the default value.

Syntax:

```
const [var, setVar] = useState(0)
```

Internal working of useState hook

- useState() creates a new cell in the functional component's memory object.
- New state values are stored in this cell during renders.
- The stack pointer points to the latest cell after each render.
- Deliberate user refresh triggers stack dump and fresh allocation.
    - The memory cell preserves state between renders, ensuring persistence.

Handling input forms with useState:

Handling input forms with useState in React involves creating state variables to store the values of input fields and updating them as the user interacts with the form.

- **State Variables**: Define state variables to hold the values of input fields. Each input field typically has its own state variable.
- **Binding Input Values**: Bind the value of each input field to its corresponding state variable. This ensures that the input field displays the current value stored in the state.
- **Event Handlers**: Create event handler functions to update the state variables as the user enters or modifies input. These functions typically listen for events like onChange for text inputs or onClick for buttons.
- **Updating State**: When the user interacts with the input fields, the event handlers update the state variables using setState, triggering a re-render to reflect the changes in the UI.

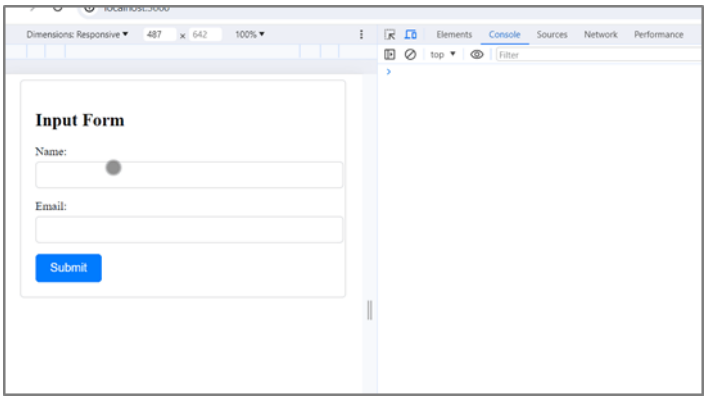
- **Submitting the Form**: When the user submits the form, you can access the values stored in the state variables to perform further actions, such as validation or sending data to a server.

Below is an example of handling input forms with useState():

```
import React, {useState} from 'react';
import './App.css';
const FormExample = () => {
    const [name, setName] = useState('');
    const [email, setEmail] = useState('');
    // Event handlers to update state variables
    const handleNameChange = (event) => {
        setName(event.target.value);
    };
    const handleEmailChange = (event) => {
        setEmail(event.target.value);
    };
const handleSubmit = (event) => {
        // Prevent default form submission
        event.preventDefault();
        console.log('Name:', name);
        console.log('Email:', email);
};
return (
        <div className="form-container">
            <h2>Input Form</h2>
            <form onSubmit={handleSubmit}>
                <div className="form-group">
                    <label>Name:</label>
                    <input
                        type="text"
                        value={name}
                        onChange={handleNameChange}
```

```
                    />
                </div>
                <div className="form-group">
                    <label>Email:</label>
                    <input
                        type="email"
                        value={email}
                        onChange={handleEmailChange}
                    />
                </div>
                <button type="submit">Submit</button>
            </form>
        </div>
    );
};
    export default FormExample;
```

Output:



MERN STACK WEB DEVELOPMENT

UNIT - 4 NOTES

1. Explain about RESTful API, and how do you handle HTTP methods in ExpressJS.

Ans:

Representational State Transfer (REST) is an architectural style that defines a set of constraints to be used for creating web services. REST API is a way of accessing web services in a simple and flexible way without having any processing. REST technology is generally preferred to the more robust Simple Object Access Protocol (SOAP) technology because REST uses less bandwidth, simple and flexible making it more suitable for internet usage. It's used to fetch or give some information from a web service. All communication done via REST API uses only HTTP request.

HTTP Methods: RESTful APIs use standard HTTP methods to perform CRUD (Create, Read, Update, Delete) operations on resources:

GET: The GET method requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect.

POST: The POST method requests that the server accept the data enclosed in the request as a new object/entity of the resource identified by the URI.

PUT: The PUT method requests that the server accept the data enclosed in the request as a modification to existing object identified by the URI. If it does not exist then the PUT method should create one.

DELETE: The DELETE method requests that the server delete the specified resource.

GET Method

The get method is a kind of HTTP request that a client makes to request some data from the server. Suppose we are creating a weather application. In order to get data about the weather, the client will make a get request from the server to get data.The get method comes with two parameters: the first is the URL, which means the URL for which the method is listening, and the second is the callback function, where that particular callback function takes two default arguments: one is a request made by the client, and the other is a response that will be sent to the client.

Syntax: app.get("URL",(req,res)=>{})

POST Method

The post method is used to send data from the client side to the server side that will be further stored in the database. Suppose in an application, the user registers with their credentials and details to receive those details in the backend through the request body, and storing that data in the database will be achieved under the post method. Similarly, the post method contains two parameters: the first is the URL, which means the URL for which the method is listening, and the second is the callback function, where that particular callback function takes two default arguments: one is a request made by the client, and the other is a response that will be sent to the client. The post-request body contains the data sent by the client, and this data can only be accessed after parsing it into JSON.

Syntax: app.post("URL",(req,res)=>{})

PUT Method

The put method is used to update the data that is already present in the database. Suppose in an application you are updating your name, so to update that name, you need to call a PUT method request from the client side to the server side, and that data will again be sent in the body of the request, and once the data is received on the server, the data will be updated in the database. Similarly, the put method contains two parameters: the first is the URL, which means the URL for which the method is listening, and the second is the callback function, where that particular callback function takes two default arguments: one is a request made by the client, and the other is a response that will be sent to the client. Again the data wants to client update will comes under body section of request.

Syntax: app.put("URL",(req,res)=>{})

DELETE Method

Delete is one of the most commonly used HTTP methods; it is used to delete particular data from a database. Suppose you are using an application consisting of notes. When you click on delete any note, it will make an HTTP delete request to the server and pass the ID of that particular note in the body of the request. Once that ID is received by the server, it will delete that particular note from Database. Similarly, the delete method contains two parameters: the first is the URL, which means the URL for which the method is listening, and the second is the callback function, where that particular callback function takes two default arguments: one is a request made by the client, and the other is a response that will be sent to the client. Again the ID will be comes under body section of request.

Syntax: app.delete("URL",(req,res)=>{})

Example: Implementation of above HTTP methods.

```
const express = require('express');
const app = express();
const PORT = 3000;
app.use(express.json());
app.get("/", (req, res) => {
        console.log("GET Request Successfull!");
        res.send("Get Req Successfully initiated");
})
app.put("/put", (req, res) => {
        console.log("PUT REQUEST SUCCESSFUL");
        console.log(req.body);
res.send(`Data Update Request Recieved`);
})
app.post("/post", (req, res) => {
        console.log("POST REQUEST SUCCESSFUL");
        console.log(req.body);
        res.send(`Data POSt Request Recieved`);
})
app.delete("/delete", (req, res) => {
        console.log("DELETE REQUEST SUCCESSFUL");
        console.log(req.body);
        res.send(`Data DELETE Request Recieved`);
})
app.listen(PORT, () => {
        console.log(`Server established at ${PORT}`);
})
```

Output:

1. **GET Request (/):**
   - Output: "GET Request Successful!" in server console.
   - Response: "Get Req Successfully initiated" sent to the client.

2. **PUT Request (/put):**

- Output: "PUT REQUEST SUCCESSFUL" followed by the body of the PUT request in server console.
- Response: "Data Update Request Received" sent to the client.

3. **POST Request (/post):**
   - Output: "POST REQUEST SUCCESSFUL" followed by the body of the POST request in server console.
   - Response: "Data POST Request Received" sent to the client.
4. **DELETE Request (/delete):**
   - Output: "DELETE REQUEST SUCCESSFUL" followed by the body of the DELETE request in server console.
   - Response: "Data DELETE Request Received" sent to the client.

2. Create Node.js Express App and Create API methods to get data from MongoDB collection using both Get and Post Methods.

Ans:

```
const express = require("express");
const mongoose = require("mongoose");
const bodyParser = require("body-parser");
const app = express();
const PORT = 3000;
// Connect to MongoDB
mongoose.connect("mongodb://localhost:27017/my_database");
// Define a schema for the MongoDB collection in index.js only. you can create
Scheme in another file also
const dataSchema = new mongoose.Schema({
  name: String,
  rno: String,
  group: String,
});
// Create a model based on the schema
const DataModel = mongoose.model("Data", dataSchema);
// Middleware to parse JSON bodies
app.use(bodyParser.json());
// GET method to retrieve data from MongoDB
app.get("/data", async (req, res) => {
```

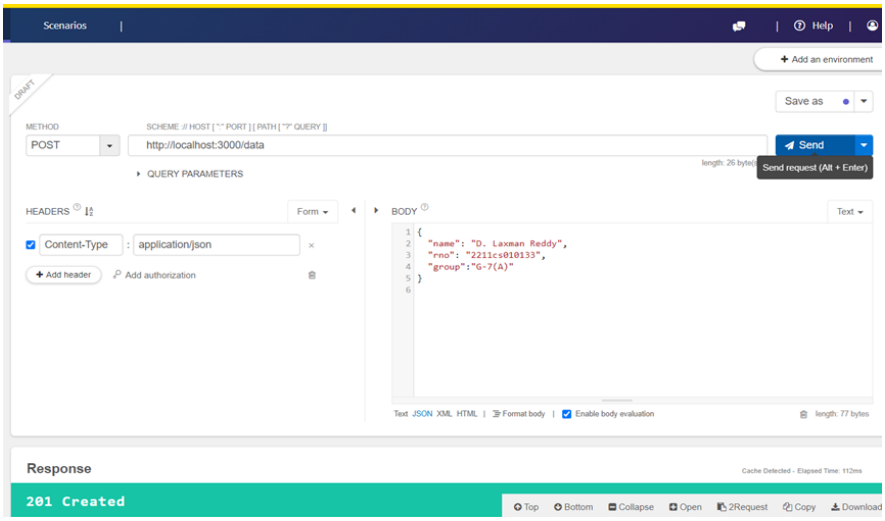
```
  try {
    const data = await DataModel.find();
    res.json(data);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});
// POST method to add data to MongoDB
app.post("/data", async (req, res) => {
  const data = new DataModel({
    name: req.body.name,
    rno: req.body.rno,
    group: req.body.group,
  });
try {
    const newData = await data.save();
    res.status(201).json(newData);
  } catch (error) {
    res.status(400).json({ message: error.message });
  }
});

// Start the server
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

start using below command:

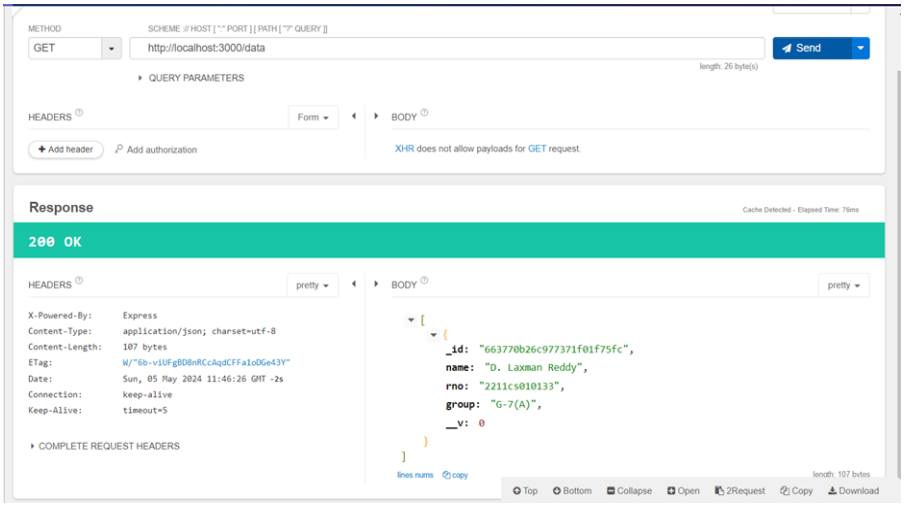
node index.js

output:

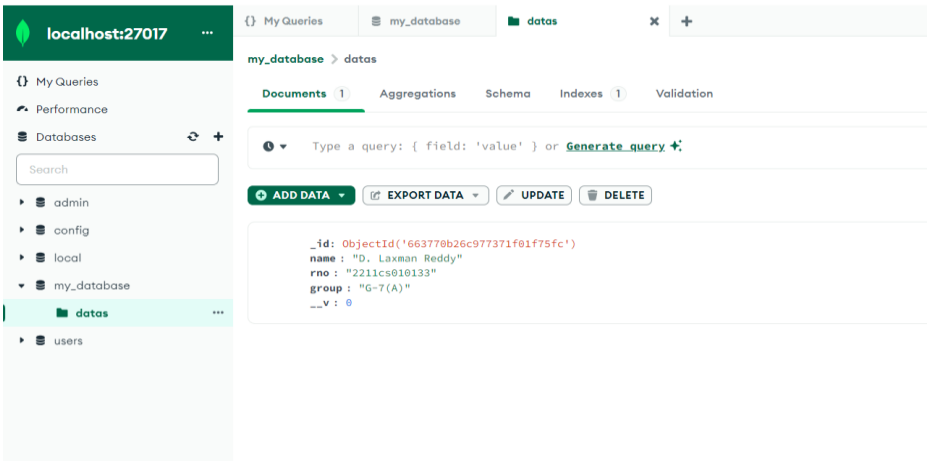
using api tester we post the data into the mongodb



using api tester we post the data into the mongodb



Using api tester get the data from mongodb



Data stored in mongodb

3. (a) What is the role of npm in Node.js development ?

Ans: npm (Node package manager) : npm is the default package manager for Node.js. We can use npm to install third-party libraries (packages) and also manage dependencies between them. In other words, npm is a package manager for Node.js projects made available for public use. Projects available on the npm registry are called "packages." npm allows us to use code written by others easily without the need to write them ourselves during development.

npm is primarily used for managing packages/modules in Node.js projects. It allows developers to easily install, update, and remove dependencies required for their projects. These dependencies are typically stored in the package.json file, which npm uses to track and manage project dependencies.npm allows developers to define custom scripts in the package.json file, which can be executed using the npm run command. This feature is useful for automating common development tasks such as building, testing, and deploying applications.npm allows developers to install packages globally, making them available across multiple projects. This is useful for installing command-line tools and utilities that are commonly used in development workflows.

Essential npm commands:

npm install : This command is used to install packages. You can either install packages globally or locally. When a package is installed globally, we can make use of the package's functionality from any directory in our computer. On the other hand, if we install a package locally, we can only make use of it in the directory where it was installed. So no other folder or file in our computer can use the package

Syntax: npm install -g [package name]

npm version : Shows you the current npm version installed on your computer.

Syntax: npm -v

npm init : The init command is used to initialize a project. When you run this command, it creates a package.json file.

Syntax: npm init -y

npm start: Used to start a package when required. **npm start**

npm publish: Used to publish an npm package to the npm registry. This is mostly used when you have created your own package.

Overall, npm plays a central role in the Node.js ecosystem by providing a convenient and efficient way to manage dependencies, share code, and automate development tasks.

(b). Explain about the Node.js modules.

Ans: In simple terms, a module is a piece of reusable JavaScript code. It could be a .js file or a directory containing .js files. We can export the content of these files and use them in other files. Modules help developers adhere to the DRY (Don't Repeat Yourself) principle in programming. They also help to break down complex logic into small, simple, and manageable chunks.

Types of Node Modules :

There are three main types of Node modules they include the following.

1. Built-in modules
2. Local modules
3. Third-party modules

1. Built-in modules:

Node.js is a light weight framework. The Built-in modules include bare minimum functionalities of Node.js. These Built-in modules are compiled into its binary distribution and load automatically when Node.js process starts. However, you need to import the core module first in order to use it in your application.

Core Modccule	Description
http	http module includes classes, methods and events to create Node.js http server.
url	url module includes methods for URL resolution and parsing.
path	path module includes methods to deal with file paths.
fs	fs module includes classes, methods, and events to work with file I/O.
util	util module includes utility functions useful for programmers.

In order to use Node.js core or NPM modules, you first need to import it using require() function as shown below.

```
var module = require('module_name');
```

The following example demonstrates how to use Node.js http module to create a web server.

```
const http = require('http');
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, World!\\n');
});
server.listen(3000)
```

In the above example, require() function returns an object because http module returns its functionality as an object, here we can then use its properties and methods using dot notation e.g. http.createServer().

2. Local Modules :

Local modules are modules created locally in your Node.js application. These modules include different functionalities of your application in separate files and folders. You can also package it and distribute it via NPM, so that Node.js community can use it.

For example, if you need to connect to MongoDB and fetch data then you can create a module for it, which can be reused in your application.

```
var log = {
info: function (info) {
console.log('Info: ' + info);
}, warning:function (warning){
console.log('Warning: ' + warning);
}, error:function (error) {
        console.log('Error: ' + error);
}
};
module.exports = log
```

In the above example of logging module, we have created an object with three functions - info(), warning() and error(). At the end, we have assigned this object to module.exports.

The module.exports in the above example exposes a log object as a module.

Loading Local Module :

```
Var myLogModule = require('./Log.js');
myLogModule.info('Node.js started');
```

In the above example, app.js is using log module. First, it loads the logging module using require() function and specified path where logging module is stored. Logging module is contained in Log.js file in the root folder. So, we have specified the path './Log.js' in the require() function. The '.' denotes a root folder.

The require() function returns a log object because logging module exposes an object in Log.js using module.exports. So now you can use logging module as an object and call any of its function using dot notation e.g myLogModule.info() or myLogModule.warning() or myLogModule.error()

3. **Third-Party Modules** :

In addition to core modules, developers can use third-party modules available on npm (Node Package Manager) to extend the functionality of their applications. Third-party modules can be installed using npm and imported into Node.js projects using require(). Some of the popular third-party modules are Mongoose, express, angular, and React.

Syntax : npm install <name-of-package>

Example :

npm install Express

npm install mongoose

For example, there's a package called “capitalize”. It performs functions like capitalizing the first letter of a word.

npm install capitalize

To use the installed package, you need to load it with the require function.

```
const capitalize = require('capitalize)
const capitalize = require('capitalize')
console.log(capitalize("hello")) // Hello
```

4. What is Express.js and why is it commonly used with Node.js?

Ans: Express.js is a widely-used web application framework for Node.js, renowned for its simplicity, flexibility, and robust feature set. It serves as a foundational tool for developers to create server-side applications, APIs, and web services with Node.js, offering a plethora of capabilities to streamline the development process.

4. Express.js is built on top of Node.js, leveraging its asynchronous, event-driven architecture.
5. It provides a higher-level abstraction for handling HTTP requests and responses, making it easier and more efficient to develop web applications in Node.js.

6. Express.js offers a minimalist, unopinionated design, allowing developers to structure their applications according to their preferences.
7. It has a vast ecosystem of middleware that can be easily integrated to add functionalities such as routing, authentication, and error handling.
8. Express.js is widely adopted in the Node.js community, making it well-documented and supported with a large number of resources and third-party modules available.
9. Express.js provides a flexible and extensible framework that allows developers to easily customize and extend the functionality of their applications.
10. It supports the use of middleware functions, which can be used to modify request and response objects, execute additional logic, and handle errors.
11. Express.js offers powerful routing capabilities, allowing developers to define routes for different HTTP methods and URL patterns.
12. It enables the creation of clean and organized route handlers, making it easy to manage complex application logic.
13. Express.js has a vibrant ecosystem of middleware modules that can be seamlessly integrated into applications.
14. Middleware functions can be used for tasks such as parsing request bodies, handling sessions, compressing responses, and implementing security features.
15. Express.js has a large and active community of developers who contribute to its development, share knowledge, and provide support.
16. The community-driven nature of Express.js ensures that it stays up-to-date with the latest web development trends and best practices
17. Express.js is known for its performance and scalability, making it suitable for building high-performance web applications and APIs.
18. It leverages the non-blocking I/O model of Node.js, allowing it to handle large numbers of concurrent requests efficiently.
19. Since Express.js is built on top of Node.js, it seamlessly integrates with other Node.js modules and libraries.
20. Express.js has a relatively low learning curve, making it accessible to developers of all skill levels.
21. Its intuitive API design and extensive documentation make it easy for developers to get started and build applications quickly.
22. Express.js integrates seamlessly with various template engines such as EJS, Handlebars, Pug (formerly Jade), allowing developers to generate dynamic HTML content efficiently.

23. This makes it convenient for building server-rendered web applications with dynamic views.
24. Express.js is well-suited for building RESTful APIs due to its lightweight and minimalist architecture.
25. It provides features such as routing, middleware, and JSON parsing out of the box, making it ideal for developing APIs that adhere to REST principles.
26. Express.js applications can be easily tested using popular testing frameworks and libraries such as Mocha, Chai, and Supertest.
27. It provides built-in support for middleware testing, making it convenient to write unit and integration tests for various components of the application.
28. Express.js simplifies session and cookie handling with middleware such as express-session and cookie-parser.
29. Developers can easily implement user authentication, session management, and cookie-based storage using these middleware modules.
30. While primarily designed for handling HTTP requests, Express.js can also be extended to support WebSocket communication using libraries like Socket.io.
31. This enables real-time bidirectional communication between the client and server, making it suitable for building interactive web applications and multiplayer games.
32. Express.js incorporates security best practices such as CSRF protection, XSS prevention, and content security policies.
33. It allows developers to implement authentication mechanisms, input validation, and data

5. What is MongoDB. What are the features of MongoDB?

A well-liked NoSQL database called MongoDB offers a scalable and adaptable way to store and retrieve data. You might need to establish a connection to a MongoDB database while using ReactJS to create a web application in order to get and modify data.

At the core of most large-scale web applications and services is a high-performance data storage solution. The backend data store is responsible for storing everything from user account information to shopping cart items to blog and comment data.

Features of MongoDB include the following:

1.Replication: A replica set is two or more MongoDB instances used to provide high availability. Replica sets are made of primary and secondary servers. The primary MongoDB server performs all the read and write operations, while the secondary replica keeps a copy of the data. If a primary replica fails, the secondary replica is then used

2.Scalability: MongoDB supports vertical and horizontal scaling. Vertical scaling works by adding more power to an existing machine, while horizontal scaling works by adding more machines to a user's resources.

3.Load balancing: MongoDB handles load balancing without the need for a separate, dedicated load balancer, through either vertical or horizontal scaling.

4.Schema-less: MongoDB is a schema-less database, which means the database can manage data without the need for a blueprint.

5.Document: Data in MongoDB is stored in documents with key-value pairs instead of rows and columns, which makes the data more flexible when compared to SQL databases.on to protect against common security threats.

6. Differentiate between MongoDB and RDBMS.

Ans:

1. A relational database management system (RDBMS) is a collection of programs and capabilities that let IT teams and others create, update, administer and otherwise interact with a relational database. RDBMS store data in the form of tables and rows. Although it is not necessary, RDBMS most commonly uses SQL.
2. One of the main differences between MongoDB and RDBMS is that RDBMS is a relational database while MongoDB is nonrelational. Likewise, while most RDBMS systems use SQL to manage stored data, MongoDB uses BSON for data storage -- a type of NoSQL database.
3. While RDBMS uses tables and rows, MongoDB uses documents and collections. In RDBMS a table -- the equivalent to a MongoDB collection -- stores data as columns and rows. Likewise, a row in RDBMS is the equivalent of a MongoDB document but stores data as structured data items in a table. A column denotes sets of data values, which is the equivalent to a field in MongoDB.

The table below summarizes the main differences between SQL and NoSQL databases.

	SQL Databases	NoSQL Databases
Data Storage Model	Tables with fixed rows and columns	Document: JSON documents, Key-value: key-value pairs, Wide-column: tables with rows and dynamic columns, Graph: nodes and edges
Development History	Developed in the 1970s with a focus on reducing data duplication	Developed in the late 2000s with a focus on scaling and allowing for rapid application change driven by agile and DevOps practices.
Examples	Oracle, MySQL, Microsoft SQL Server, and PostgreSQL	Document: MongoDB and CouchDB, Key-value: Redis and DynamoDB, Wide-column: Cassandra and HBase, Graph: Neo4j and Amazon Neptune
Primary Purpose	General purpose	Document: general purpose, Key-value: large amounts of data with simple lookup queries, Wide-column: large amounts of data with predictable query patterns, Graph: analyzing and traversing relationships between connected data
Schemas	Rigid	Flexible
Scaling	Vertical (scale-up with a larger server)	Horizontal (scale-out across commodity servers)
Multi-Record ACID Transactions	Supported	Most do not support multi-record ACID transactions. However, some — like MongoDB — do.
Joins	Typically required	Typically not required
Data to Object Mapping	Requires ORM (object-relational mapping)	Many do not require ORMs. MongoDB documents map directly to data structures in most popular programming languages.

MongoDB is an open-source document-oriented database used for high volume data storage. It falls under the classification of a NoSQL database. NoSQL tool means that it doesn't utilize the usual rows and columns. MongoDB uses BSON (document storage format) which is a binary style of JSON documents.

The following table shows the relationship of RDBMS terminology with MongoDB.

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by MongoDB itself)
Database Server and Client	
mysqld/Oracle	mongod
mysql/sqlplus	mongo

7. What are CRUD Operations in MongoDB.

Ans: MongoDB documents are modified by connecting to a server, querying the proper documents, and then changing the setting properties before sending the data back to the database to be updated. CRUD is data-oriented, and it's standardized according to HTTP action verbs.

When it comes to the individual CRUD operations:

1. The Create operation is used to insert new documents in the MongoDB database.
2. The Read operation is used to query a document in the database.
3. The Update operation is used to modify existing documents in the database.
4. The Delete operation is used to remove documents in the database.

Create Operations

For MongoDB CRUD, if the specified collection doesn't exist, the create operation will create the collection when it's executed. Create operations in MongoDB target a single collection, not multiple collections.

MongoDB provides two different create operations that you can use to insert documents into a collection:

- db.collection.insertOne()
- db.collection.insertMany()

insertOne()

As the namesake, insertOne() allows you to insert one document into the collection. For this example, we're going to work with a collection called RecordsDB. We can insert a single entry into our collection by calling the insertOne() method on RecordsDB. We then provide the information we want to insert in the form of key-value pairs, establishing the schema.

Example

```
db.RecordsDB.insertOne({
   name: "Marsh",
   age: "6 years",
   species: "Dog",
   ownerAddress: "380 W. Fir Ave",
   chipped: true
})
```

If the create operation is successful, a new document is created. The function will return an object where “acknowledged” is “true” and “insertID” is the newly created “ObjectId.”

Output

```
{
      "acknowledged" : true,
      "insertedId" : ObjectId("5fd989674e6b9ceb8665c57d")
}
```

insertMany()

It’s possible to insert multiple items at one time by calling the insertMany() method on the desired collection. In this case, we pass multiple items into our chosen collection (RecordsDB) and separate them by commas. Within the parentheses, we use brackets to indicate that we are passing in a list of multiple entries. This is commonly referred to as a nested method.

Example

```
db.RecordsDB.insertMany([{
   name: "Marsh",
   age: "6 years",
   species: "Dog",
   ownerAddress: "380 W. Fir Ave",
   chipped: true},
    {name: "Kitana",
    age: "4 years",
    species: "Cat",
    ownerAddress: "521 E. Cortland",
    chipped: true}])
```

Output

```
{
      "acknowledged" : true,
      "insertedIds" : [
            ObjectId("5fd98ea9ce6e8850d88270b4"),
            ObjectId("5fd98ea9ce6e8850d88270b5")
      ]
}
```

Read Operations

The read operations allow you to supply special query filters and criteria that let you specify which documents you want. The MongoDB documentation contains more information on the available query filters. Query modifiers may also be used to change how many results are returned.

MongoDB has two methods of reading documents from a collection:

- db.collection.find()
- db.collection.findOne()

find()

In order to get all the documents from a collection, we can simply use the find() method on our chosen collection. Executing just the find() method with no arguments will return all records currently in the collection.

Example

db.RecordsDB.find()

Output

{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }

{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years", "species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }

Here we can see that every record has an assigned “ObjectId” mapped to the “_id” key.

If you want to get more specific with a read operation and find a desired subsection of the records, you can use the previously mentioned filtering criteria to choose what results should be returned. One of the most common ways of filtering the results is to search by value.

Example

db.RecordsDB.find({"species":"Cat"})

{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }

findOne()

In order to get one document that satisfies the search criteria, we can simply use the findOne() method on our chosen collection. If multiple documents satisfy the query, this method returns the first document according to the natural order which reflects the order of documents on the disk. If no documents satisfy the search criteria, the function returns null. The function takes the following form of syntax.

Syntax

db.{collection}.findOne({query}, {projection})

Let's take the following collection—say, RecordsDB, as an example.

Output

{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }

{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years", "species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }

And, we run the following line of code:

Example:

db.RecordsDB.find({"age":"8 years"})

We would get the following result:

Output

{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }

Notice that even though two documents meet the search criteria, only the first document that matches the search condition is returned.

Update Operations

Like create operations, update operations operate on a single collection, and they are atomic at a single document level. An update operation takes filters and criteria to select the documents you want to update.

You should be careful when updating documents, as updates are permanent and can't be rolled back. This applies to delete operations as well.

For MongoDB CRUD, there are three different methods of updating documents:

- db.collection.updateOne()
- db.collection.updateMany()
- db.collection.replaceOne()

updateOne()

We can update a currently existing record and change a single document with an update operation. To do this, we use the updateOne() method on a chosen collection, which here is "RecordsDB." To update a document, we provide the method with two arguments: an update filter and an update action.

The update filter defines which items we want to update, and the update action defines how to update those items. We first pass in the update filter. Then, we use the “$set” key and provide the fields we want to update as a value. This method will update the first record that matches the provided filter.

Example

db.RecordsDB.updateOne({name: "Marsh"}, {$set:{ownerAddress: "451 W. Coffee St. A204"}})

output

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

updateMany()

updateMany() allows us to update multiple items by passing in a list of items, just as we did when inserting multiple items. This update operation uses the same syntax for updating a single document.

Example:

db.RecordsDB.updateMany({species:"Dog"}, {$set: {age: "5"}})

Output

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }

replaceOne()

The replaceOne() method is used to replace a single document in the specified collection. replaceOne() replaces the entire document, meaning fields in the old document not contained in the new will be lost.

Example

db.RecordsDB.replaceOne({name: "Kevin"}, {name: "Maki"})

Output

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

Delete Operations

Delete operations operate on a single collection, like update and create operations. Delete operations are also atomic for a single document. You can provide delete operations with filters and criteria in order to specify which documents you would like to delete from a collection. The filter options rely on the same syntax that read operations utilize.

MongoDB has two different methods of deleting records from a collection:

- db.collection.deleteOne()
- db.collection.deleteMany()

deleteOne()

DeleteOne() is used to remove a document from a specified collection on the MongoDB server. A filter criteria is used to specify the item to delete. It deletes the first record that matches the provided filter.

Example

db.RecordsDB.deleteOne({name:"Maki"})

Output

{ "acknowledged" : true, "deletedCount" : 1 }

deleteMany()

DeleteMany() is a method used to delete multiple documents from a desired collection with a single delete operation. A list is passed into the method and the individual items are defined with filter criteria as in deleteOne().

Example

db.RecordsDB.deleteMany({species:"Dog"})

Output

{ "acknowledged" : true, "deletedCount" : 2 }

8. a. What are MongoDB documents and collections.

b. What syntax is used to create and drop a collection in MongoDB?

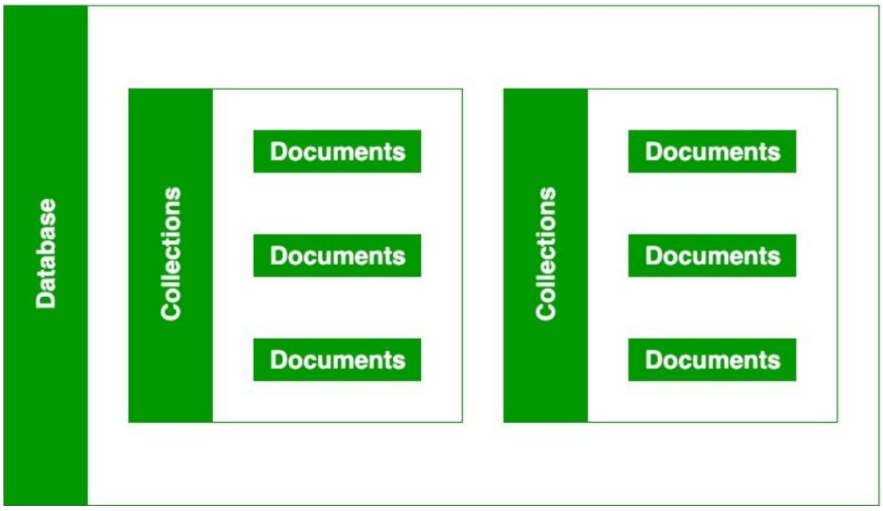
Ans)

MongoDB is a document-oriented NoSQL database system that provides high scalability, flexibility, and performance. Unlike standard relational databases, MongoDB stores data in a JSON document structure form. This makes it easy to operate with dynamic and unstructured data and MongoDB is an open-source and cross-platform database System.

Database

- Database is a container for collections.
- Each database gets its own set of files.
- A single MongoDB server can have multiple databases.

Databases, collections, documents are important parts of MongoDB without them you are not able to store data on the MongoDB server. A Database contains a collection, and a collection contains documents and the documents contain data, they are related to each other.



In MongoDB, a database contains the collections of documents. One can create multiple databases on the MongoDB server.

Documents:

- The document is the unit of storing data in a MongoDB database.
- document use JSON (JavaScript Object Notation, is a lightweight, thoroughly explorable format used to interchange data between various applications) style for storing data.
- A simple example of a JSON document is as follows: **Employee.json**

```
Result
-------------------------------------------------
{
"employees":
[       {
          "name":"Raj",
          "email":"raj@gmail.com",
          "age":32

},
       {
          "name":"Mohan",
          "email":"Mohan@yahoo.com",
          "age":21

         }
]
}
```

- Often, the term "object" is used to refer a document.
- Documents are analogous to the records of an RDBMS. Insert, update, and delete operations can be performed on a collection. The following table will help you to understand the concept more easily:

RDBMS	**MongoDB**
Table	Collection
Column	Key
Value	Value
Records / Rows	Document / Object

- The following table shows the various datatypes which may be used in MongoDB.

Data Types	**Description**
string	May be an empty string or a combination of characters.
integer	Digits.
boolean	Logical values True or False.
double	A type of floating point number.
null	Not zero, not empty.
array	A list of values.
object	An entity which can be used in programming. May be a value, variable, function, or data structure.

timestamp	A 64 bit value referring to a time and unique on a single "mongod" instance. The first 32 bit of this value refers to seconds since the UTC January 1, 1970. And last 32 bits refer to the incrementing ordinal for operations within a given second.
Internationalized Strings	UTF-8 for strings.
Object IDs	Every MongoDB object or document must have an Object ID which is unique. This is a BSON(Binary JavaScript Object Notation, which is the binary interpretation of JSON) object id, a 12-byte binary value which has a very rare chance of getting duplicated. This id consists of a 4-byte timestamp (seconds since epoch), a 3-byte machine id, a 2-byte process id, and a 3-byte counter.

Collections:

- A collection may store a number of documents. A collection is analogous to a table of an RDBMS.
- A collection may store documents those who are not same in structure. This is possible because MongoDB is a Schema-free database. In a relational database like MySQL, a schema defines the organization / structure of data in a database. MongoDB does not require such a set of formula defining structure of data. So, it is quite possible to store documents of varying structures in a collection. Practically, you don't need to define a column and it's datatype unlike in RDBMS, while working with MongoDB.
- In the following code, it is shown that two MongoDB documents, belongs to same collection, storing data of different structures.

```
1  {"tutorial" : "NoSQL"}
2                  {"topic_id" : 7}
```

- A collection is created, when the first document is inserted.
- Collection names must begin with letters or an underscore.
- A Collection name may contain numbers.
- You can't use "$" character within the name of a collection. "$" is reserved.
- A Collection name must not exceed 128 characters. It will be nice if you keep it within 80/90 characters.

- Using a "." (dot) notation, collections can be organized in named groups. For example, tutorials.php and tutorials.javascript both belong to tutorials. This mechanism is called as collection namespace which is for user primarily. Databases don't have much to do with it.
- Following is how to use it programmatically:

```
1  db.tutorials.php.findOne()
```

Creating a Collection:

To create a collection in MongoDB, you use the “db.createCollection()” method.

Syntax:

```
db.createCollection(
   "collectionName", // Name of the collection you want to create
   options // Optional parameter for additional configuration options
)
```

- **"collectionName":** This is a string parameter representing the name of the collection you want to create. You can choose any valid string as the name for your collection. For example, "users", "products", "orders", etc.
- **options (optional):** This parameter allows you to specify additional configuration options for the collection being created. These options can include settings related to storage engine, validation rules, indexing options, etc. If you don't need to specify any options, you can omit this parameter.

Dropping a Collection:

To drop (delete) a collection in MongoDB, we use the db.collectionName.drop() method.

Syntax:

```
db.collectionName.drop()
```

- **"collectionName":** This is the name of the collection you want to drop. After connecting to a specific database, you can reference collections within that database directly using the db object, followed by the name of the collection. For example, if you have a collection named "users", you would reference it as db.users.
- **.drop():** This method is called on the collection object and tells MongoDB to delete the entire collection along with all its documents and indexes.

9. What is routing in Express JS and how do you implement it?

Ans:

Express routing is about showing your app how to respond to different URLs. It involves associating HTTP methods with specific functions to handle requests. This helps organize and control the flow of your web application.

Routing in Express:

- Routing in Express is like showing your web app where to go.
- It's about deciding what your app should do when users go to various URLs.
- You get to set the actions for things like going to the homepage, submitting forms, or clicking links.
- Express makes it simple by letting you create rules that connect to specific parts of your code.

Steps to Implement Routing in Express:

Step 1: Initialising the Node App using the below command:

npm init -y

Step 2: Installing express in the app:

npm install express

Example: Below is the example of routing in express:

```
// server.js
const express = require("express");
const app = express();
const PORT = 3133;
// define route
app.get("/login", (req, res) => {
  res.send("<h1>welcome to login page</h1>");
});
app.get("/home", (req, res) => {
  res.send("<h1>welcome to home page</h1>");
});
app.listen(PORT, () => {
  console.log(`Server is listening at http://localhost:${PORT}`);
});
```

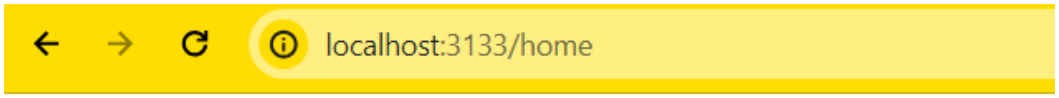
- Start using the following command

  node server.js

OUTPUT:

welcome to login page

Route Path “/login”



welcome to home page

Route path “/home”


```
Cannot GET /
```

We didn’t write the code for root path “/”

40	Develop a Node.js Express App and write CRUD methods for Students Collection in MongoDB database	8	5

Ans.

Below is a step-by-step guide on how to create a Node.js Express application with CRUD (Create, Read, Update, Delete) methods for a "Students" collection in a MongoDB database.

Prerequisites

1. Node.js and npm: Make sure you have Node.js and npm installed.

2. MongoDB: You need a running instance of MongoDB. You can use a local instance or a cloud-based one like MongoDB Atlas.

Step 1: Setup the Project

1. Initialize the Project

Open your terminal and create a new directory for your project. Then initialize it with npm:

mkdir student-crud-app

cd student-crud-app

npm init -y

2. Install Dependencies

Install the necessary npm packages:

npm install express mongoose body-parser

Step 2: Create the Express Server

1. Create the Entry Point

Create a file named `server.js`:

```
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');
const app = express();
// Middleware
app.use(bodyParser.json());
// MongoDB Connection
mongoose.connect('mongodb://localhost:27017/studentdb', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
```

```
});

const db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', () => {
  console.log('Connected to MongoDB');
});
// Routes
app.use('/api/students', require('./routes/students'));
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

Step 3: Create the Student Model

1. Create a Folder for Models:

Inside your project directory, create a folder named `models` and inside it, create a file named `Student.js`:

```
const mongoose = require('mongoose');
const studentSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
  age: {
    type: Number,
    required: true,
  },
  major: {
    type: String,
    required: true,
  },
});
module.exports = mongoose.model('Student', studentSchema);
```

Step 4: Create the Routes for CRUD Operations

1. Create a Folder for Routes

Inside your project directory, create a folder named `routes` and inside it, create a file named `students.js`:

```
const express = require('express');
const router = express.Router();
const Student = require('../models/Student');

// Create a new student
router.post('/', async (req, res) => {
  const { name, age, major } = req.body;
  const newStudent = new Student({ name, age, major });

  try {
    const savedStudent = await newStudent.save();
    res.status(201).json(savedStudent);
  } catch (error) {
    res.status(400).json({ message: error.message });
  }
});

// Get all students
router.get('/', async (req, res) => {
  try {
    const students = await Student.find();
    res.json(students);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

// Get a student by ID
router.get('/:id', async (req, res) => {
  try {
    const student = await Student.findById(req.params.id);
```

```
    if (student == null) {
      return res.status(404).json({ message: 'Student not found' });
    }
    res.json(student);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

// Update a student by ID
router.put('/:id', async (req, res) => {
  try {
    const { name, age, major } = req.body;
    const updatedStudent = await Student.findByIdAndUpdate(
      req.params.id,
      { name, age, major },
      { new: true, runValidators: true }
    );
    if (updatedStudent == null) {
      return res.status(404).json({ message: 'Student not found' });
    }
    res.json(updatedStudent);
  } catch (error) {
    res.status(400).json({ message: error.message });
  }
});

// Delete a student by ID
router.delete('/:id', async (req, res) => {
  try {
    const student = await Student.findByIdAndDelete(req.params.id);
    if (student == null) {
      return res.status(404).json({ message: 'Student not found' });
    }
    res.json({ message: 'Student deleted' });
```

```
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

module.exports = router;
```

Step 5: Run the Application

1. Start the Server

In your terminal, start the server:

node server.js

2. Test the Endpoints

Use a tool like Postman or curl to test the CRUD endpoints:

- Create a Student (POST): `http://localhost:3000/api/students`
- Get All Students (GET): `http://localhost:3000/api/students`
- Get a Student by ID (GET): `http://localhost:3000/api/students/:id`
- Update a Student by ID (PUT): `http://localhost:3000/api/students/:id`
- Delete a Student by ID (DELETE): `http://localhost:3000/api/students/:id`

41	How does React interact with Express in a MERN stack application?	8	5

Ans.

In a MERN (MongoDB, Express.js, React.js, Node.js) stack application, React and Express interact to create a full-fledged web application. Here's a brief overview of how they work together:

1. Client-side (React):

- React is responsible for handling the client-side rendering of the application. It creates the user interface and manages the interactions within the browser.

- React components are structured to represent different parts of the UI, and they can communicate with each other through props and state.

- When the user interacts with the application, such as submitting a form or clicking a button, React handles these events and updates the UI accordingly without reloading the entire page. It achieves this through its virtual DOM and reconciliation process.

2. Server-side (Express.js):

- Express.js is a web application framework for Node.js that handles the server-side logic and routing of the application.

- It defines routes that handle HTTP requests from the client and sends back appropriate responses. These routes can handle tasks like fetching data from a database, processing form submissions, or serving static files.

- Express middleware can be used for various purposes such as logging, authentication, and error handling. Middleware functions can intercept and process requests before they reach the route handlers.

3. Interaction between React and Express:

- React components often need to fetch data from the server to display dynamic content. This is typically done using HTTP requests, such as AJAX requests or using modern techniques like the Fetch API or Axios.

- When React components need to fetch data, they make HTTP requests to the Express server, usually to specific API endpoints defined in Express routes.

- Express routes then handle these requests, querying the database if necessary, and sending back the requested data in the response.

- React components receive the data from the server and update their state or props accordingly, triggering re-rendering of the UI to display the new information.

First, let's create a simple Express server with a couple of routes to handle API requests:

```
// server.js
const express = require('express');
const app = express();
const PORT = process.env.PORT || 5000;

// Example data
const items = [
  { id: 1, name: 'Item 1' },
  { id: 2, name: 'Item 2' },
  { id: 3, name: 'Item 3' },
];

// Route to get all items
app.get('/api/items', (req, res) => {
  res.json(items);
});

// Route to get a single item by ID
app.get('/api/items/:id', (req, res) => {
  const itemId = parseInt(req.params.id);
```

```
  const item = items.find(item => item.id === itemId);
  if (item) {
    res.json(item);
  } else {
    res.status(404).json({ message: 'Item not found' });
  }
});

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

Next, let's create a simple React component that fetches data from these API endpoints and displays it:

```
// App.js

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

function App() {
  const [items, setItems] = useState([]);

  useEffect(() => {
    // Fetch all items when component mounts
    axios.get('/api/items')
      .then(response => {
        setItems(response.data);
      })
      .catch(error => {
        console.error('Error fetching items:', error);
      });
  }, []); // Empty dependency array means this effect runs only once on component mount

  return (
    <div>
```

```
      <h1>Items</h1>
      <ul>
        {items.map(item => (
          <li key={item.id}>{item.name}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

In this example:

- Express server serves two routes: `/api/items` to get all items and `/api/items/:id` to get a single item by its ID.

- React component `App` fetches all items from `/api/items` endpoint using Axios when it mounts and displays them in a list.

To run this example, make sure you have Node.js and npm installed. Then, create a new directory, place these files in it, install the necessary dependencies (`express`, `axios`, etc.) using npm, and start the server using `node server.js`. Finally, run the React app using `npm start` in another terminal window.

42	What are the advantages of using Axios or Fetch for making API calls in React?	8	5

Ans.

When deciding between Axios and Fetch for making API calls in a React application, each has its own set of advantages. Here's a breakdown:

Axios

1. Simplified Syntax:

- Axios has a simpler and more concise syntax, making code easier to read and write. For example:

```
axios.get('/api/endpoint')
  .then(response => console.log(response))
  .catch(error => console.error(error));
```

2. Automatic JSON Transformation:

- Axios automatically transforms JSON data, eliminating the need to manually parse the response with response.json(), which is required when using Fetch.

3. Interceptors:

- Axios provides interceptors to modify requests or responses before they are handled by then or catch, which is useful for tasks like adding authorization headers or handling global error logging.

4. Request Cancellation:

- Axios supports canceling requests using the CancelToken feature, which can be useful for aborting ongoing requests when they are no longer needed.

5. Default Configurations:

- You can set default configurations like base URLs, headers, and timeouts globally, which helps avoid redundancy and keeps the codebase clean.

6. Browser Support:

- Axios has built-in support for older browsers and automatically handles polyfills, which might be required for Fetch.

7. Error Handling:

- Axios provides better error handling by throwing errors for HTTP status codes that are outside the range of 2xx. This makes it easier to handle different error scenarios.

Fetch

1. Native JavaScript API:

- Fetch is a built-in JavaScript API and does not require additional libraries, which can reduce the size of the application.

2. Modern and Flexible:

- Fetch follows a modern and flexible approach, utilizing Promises and allowing more fine-grained control over the request and response.

3. Stream Handling:

- Fetch allows direct access to response streams, which can be useful for handling large files or progressively loading content.

4. Wide Usage and Familiarity:

- Fetch is widely used and familiar to many developers, making it easier to find examples, documentation, and community support.

5. Low-Level Control:

- Fetch provides low-level control over requests and responses, making it suitable for handling complex scenarios that might require custom handling.

Summary

- Use Axios if you prefer a more feature-rich library with simplified syntax, built-in JSON handling, request interceptors, and better error handling.

- Use Fetch if you prefer a native, lightweight solution that is built into the browser and requires no additional dependencies, especially if you need fine-grained control over the network requests and responses.

In many cases, Axios is preferred for its convenience and additional features, while Fetch might be chosen for its native support and flexibility without extra dependencies. The choice depends on the specific needs and preferences of the project and development team.

43	Build a React App to fetch and display Students List from Node.js Express API.	8	5

Ans.

To build a React app that fetches and displays a list of students from a Node.js Express API, we need to follow these steps:

1. Set Up the Node.js Express API:

- Create a simple Express server to serve the students list.

2. Set Up the React App:

- Create a React app using create-react-app.

- Fetch the students list from the API and display it.

Step 1: Setting Up the Node.js Express API

1. Create a new directory for the API and initialize it with npm init -y.

2. Install the necessary dependencies:

bash

npm install express cors

3. Create the Express server in a file named server.js:

```
const express = require('express');
const cors = require('cors');

const app = express();
const port = 5000;

app.use(cors());

const students = [
  { id: 1, name: 'John Doe', age: 20 },
  { id: 2, name: 'Jane Smith', age: 22 },
  { id: 3, name: 'Alice Johnson', age: 19 },
];

app.get('/api/students', (req, res) => {
  res.json(students);
});

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

4. Run the server:

```
node server.js
```

Step 2: Setting Up the React App

1. Create a new React app using create-react-app:

```
npx create-react-app student-list
cd student-list
```

2. Install Axios for making API calls:

```
npm install axios
```

3. Update the App.js file to fetch and display the students list:

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

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

const App = () => {
  const [students, setStudents] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const fetchStudents = async () => {
      try {
        const response = await axios.get('http://localhost:5000/api/students');
        setStudents(response.data);
        setLoading(false);
      } catch (error) {
        console.error('Error fetching the students:', error);
        setLoading(false);
      }
    };

    fetchStudents();
  }, []);

  return (
    <div className="App">
      <h1>Student List</h1>
      {loading ? (
        <p>Loading...</p>
      ) : (
        <ul>
          {students.map(student => (
            <li key={student.id}>
              {student.name} - {student.age} years old
            </li>
          ))}
        </ul>
      )}
```

```
    </div>
  );
};

export default App;
```

4. Start the React app:

```
npm start
```

Summary

1. The Node.js Express server serves the list of students at the /api/students endpoint.

2. The React app fetches the students list from this endpoint using Axios and displays it.

By following these steps, you will have a simple React app that fetches and displays a list of students from a Node.js Express API.

44	How do you handle asynchronous operations when making API calls in React?	8	5

Ans.

Handling asynchronous operations in React, particularly when making API calls, involves using JavaScript's async/await syntax within the React component lifecycle methods or hooks. Here is a detailed approach:

Using Functional Components with Hooks

1. Using useEffect and useState Hooks:

- useEffect is used for side effects in functional components.
- useState is used to manage component state.

Here's a step-by-step example:

1. Set Up the React Component:

- Import the necessary hooks and libraries.
- Define state variables for storing the API data and loading state.
- Use useEffect to make the API call when the component mounts.

2. Handle API Call:

- Use async/await within the useEffect hook to fetch data.
- Handle loading state and errors appropriately.

Example Code:

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

const StudentsList = () => {
  const [students, setStudents] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchStudents = async () => {
      try {
        // Making the API call
        const response = await axios.get('http://localhost:5000/api/students');
        setStudents(response.data);
        setLoading(false);
      } catch (err) {
        console.error('Error fetching the students:', err);
        setError(err);
        setLoading(false);
      }
    };

    fetchStudents();
  }, []); // Empty dependency array means this useEffect runs once after the initial render

  if (loading) {
    return <div>Loading...</div>;
  }

  if (error) {
    return <div>Error fetching data</div>;
  }

  return (
```

```
    <div>
      <h1>Student List</h1>
      <ul>
        {students.map(student => (
          <li key={student.id}>
            {student.name} - {student.age} years old
          </li>
        ))}
      </ul>
    </div>
  );
};

export default StudentsList;
```

Key Points

1. State Management:

- students state variable holds the fetched data.
- loading state variable indicates whether the data is currently being fetched.
- error state variable stores any error that might occur during the fetch operation.

2. Effect Hook (useEffect):

- The useEffect hook with an empty dependency array ([]) ensures the fetch operation runs only once when the component mounts.
- Inside useEffect, the fetchStudents function is defined and immediately invoked.

3. Async/Await for API Call:

- async/await is used to handle the asynchronous API call in a cleaner way compared to using .then() and .catch().

4. Error Handling:

- Any errors during the API call are caught in the catch block, and the error state is updated accordingly.

5. Conditional Rendering:

- The component conditionally renders content based on the loading and error states to provide user feedback.

Benefits:

- Readability: Using async/await within useEffect makes the code more readable and easier to understand.

- Clean State Management: Managing different states (loading, error, data) helps in creating a responsive UI that provides feedback to the user during different stages of the data fetching process.

- Reusability: This pattern can be reused across different components that need to perform similar asynchronous operations.

45	What is CORS and how does it affect communication between React frontend and Express backend?	8	5

Ans.

What is CORS?

CORS (Cross-Origin Resource Sharing) is a security feature implemented by web browsers to prevent web pages from making requests to a different domain than the one that served the web page. This is a security measure to prevent malicious websites from accessing sensitive data on other sites without permission.

How CORS Works?

When a web page makes an HTTP request to a different domain (cross-origin request), the browser sends an HTTP request with an Origin header. The server can then decide whether to allow the request based on this header. If the server allows the request, it sends back specific CORS headers in its response, indicating that the request is permitted.

CORS Headers

- **Access-Control-Allow-Origin:** Specifies which origins are allowed to access the resource. It can be a specific domain, a list of domains, or * (which allows any domain).

- **Access-Control-Allow-Methods:** Specifies the HTTP methods that are allowed (e.g., GET, POST, PUT, DELETE).

- **Access-Control-Allow-Headers:** Specifies the headers that are allowed in the request.

- **Access-Control-Allow-Credentials:** Indicates whether the response to the request can be exposed when the credentials flag is true.

CORS in React Frontend and Express Backend

When a React frontend tries to communicate with an Express backend hosted on a different domain or port, CORS policy comes into play. For example, if your React app is running on http://localhost:3000 and your Express backend is running on http://localhost:5000, any request from React to Express is considered a cross-origin request.

How to Handle CORS in Express

To enable CORS in an Express application, you can use the cors middleware. Here's how to do it:

1. Install the CORS Middleware:

    npm install cors

2. Use the CORS Middleware in Your Express App:

```
const express = require('express');
const cors = require('cors');

const app = express();
const port = 5000;

// Use CORS middleware
app.use(cors());

const students = [
  { id: 1, name: 'John Doe', age: 20 },
  { id: 2, name: 'Jane Smith', age: 22 },
  { id: 3, name: 'Alice Johnson', age: 19 },
];

app.get('/api/students', (req, res) => {
  res.json(students);
});

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

Explanation

1. Install and Import cors: Install the cors package and import it into your Express application.

2. Apply cors Middleware: Use the cors middleware in your Express app to enable CORS for all routes. This allows your React frontend to make requests to your Express backend without running into CORS issues.

Customizing CORS

You can also configure CORS to be more restrictive by specifying the allowed origins, methods, and headers:

```
const corsOptions = {
  origin: 'http://localhost:3000',
  methods: 'GET,POST,PUT,DELETE',
  allowedHeaders: 'Content-Type,Authorization',
```

```
  credentials: true,
};

app.use(cors(corsOptions));
```

46	How do you handle error responses from API calls in React?	8	5

Ans.

Handling error responses from API calls in React involves several steps to ensure that errors are appropriately caught, handled, and communicated to the user. Here is a comprehensive approach using axios and functional components with hooks.

Steps to Handle Error Responses

1. Set Up State for Errors: Use the useState hook to manage an error state.

2. Make API Calls with axios: Use axios to make API calls and handle errors with try-catch blocks.

3. Display Errors to the User: Render error messages conditionally based on the error state.

Example Code

Here's an example that demonstrates these steps:

1. Install axios:

```
npm install axios
```

2. Set Up React Component:

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

const StudentsList = () => {
  const [students, setStudents] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchStudents = async () => {
      try {
        setLoading(true);
        const response = await axios.get('http://localhost:5000/api/students');
        setStudents(response.data);
        setLoading(false);
```

```
    } catch (err) {
      console.error('Error fetching the students:', err);
      setError(err.response ? err.response.data : 'Error fetching data');
      setLoading(false);
    }
  };

  fetchStudents();
 }, []); // Empty dependency array means this useEffect runs once after the initial render

 if (loading) {
  return <div>Loading...</div>;
 }

 if (error) {
  return <div>Error: {error}</div>;
 }

 return (
  <div>
   <h1>Student List</h1>
   <ul>
    {students.map(student => (
     <li key={student.id}>
      {student.name} - {student.age} years old
     </li>
    ))}
   </ul>
  </div>
 );
};

export default StudentsList;
```

Explanation

1. State Management:

- students: Holds the fetched data.

- loading: Indicates whether data is being fetched.

- error: Holds any error message encountered during the fetch operation.

2. Fetching Data:

- useEffect: Triggers the fetchStudents function once after the component mounts.

- fetchStudents: An asynchronous function to fetch data using axios.

- try-catch: Handles both successful and failed API calls. In the catch block, err.response checks for the server response; if it's not available, a generic error message is used.

3. Rendering Conditional Content:

- While loading is true, a loading message is displayed.

- If an error occurs, the error message is displayed.

- If data is successfully fetched, it is displayed in a list.

Best Practices

1. Error Logging:

- Log errors to the console for debugging during development. This can be expanded to log errors to an external monitoring service in production.

2. User-Friendly Error Messages:

- Customize error messages to be user-friendly and informative. Avoid showing raw error objects or technical details.

3. Graceful Error Handling:

- Ensure the UI can handle different types of errors gracefully, such as network errors, server errors, and validation errors.

4. Fallback UI:

- Consider implementing a fallback UI component that users can interact with if data cannot be fetched.

5. Retries:

- Implement retry logic for transient errors to improve robustness.

Summary

Handling error responses in React involves managing an error state, using try-catch blocks for API calls, and conditionally rendering error messages. This ensures a better user experience and makes the application more robust. By following the example and best practices provided, you can effectively handle errors in your React applications.

47	Can you describe the process of setting up user authentication in a MERN stack application?	8	5

Ans.

Certainly! Setting up user authentication in a MERN (MongoDB, Express.js, React.js, Node.js) stack application involves several steps. Here's a general outline of the process:

1. Install Necessary Packages:

- Install required packages for authentication. For example:

- Express.js: `express-session`, `passport`, `passport-local` (for local strategy), `bcryptjs` (for password hashing).

- React.js: `axios` (for making HTTP requests to the backend).

- You may also use additional packages or libraries depending on your specific requirements (e.g., JWT for token-based authentication).

2. Set Up MongoDB:

- Design your user schema and create a collection to store user data in MongoDB.

- Connect your Node.js application to MongoDB using a MongoDB driver like `mongoose`.

3. Set Up Express.js Server:

- Create routes for user authentication (e.g., registration, login, logout).

- Implement middleware for session management and authentication using `passport`.

- Configure passport strategies (e.g., local strategy, JWT strategy).

4. Implement User Registration:

- Create an endpoint to handle user registration.

- Validate user input (e.g., check for unique username or email).

- Hash the user's password using `bcryptjs` before storing it in the database.

5. Implement User Login:

- Create an endpoint to handle user login.

- Authenticate the user using `passport-local` or another strategy.

- Issue a session or JWT token upon successful authentication.

6. Implement User Logout:

- Create an endpoint to handle user logout.
- Destroy the user's session or revoke the JWT token.

7. Integrate Authentication with React.js:

- Create components for registration, login, and logout forms.
- Implement form submission logic using `axios` to communicate with the backend API.
- Store authentication tokens (session or JWT) in browser storage (e.g., localStorage or sessionStorage).
- Update the UI based on the user's authentication status (e.g., show different content for authenticated users).

8. Handle Protected Routes:

- Implement a mechanism to protect routes that require authentication.
- Create middleware to check if the user is authenticated before allowing access to protected routes.
- Redirect unauthenticated users to the login page or display an appropriate error message.

9. Optional Enhancements:

- Implement features like password reset, email verification, or social login (using OAuth).
- Add error handling and logging to improve security and user experience.

10. Testing and Deployment:

- Test the authentication flow thoroughly, including edge cases and error scenarios.
- Deploy your MERN stack application to a hosting provider (e.g., Heroku, AWS, DigitalOcean) and ensure that authentication works correctly in the production environment.

By following these steps, you can set up user authentication in your MERN stack application effectively. Remember to prioritize security best practices and consider the specific requirements of your application while implementing authentication.

48	What is JWT authentication and how is it implemented in Express for user authentication?	8	5

Ans.

JWT (JSON Web Token) authentication is a method of user authentication where JSON web tokens are used to securely transmit information between parties. JWTs are compact, URL-safe tokens that consist of three parts: a header, a payload, and a signature. They are commonly used for authentication and information exchange in web applications.

Here's a brief overview of how JWT authentication works and how it can be implemented in Express for user authentication:

1. JWT Workflow:

- Authentication: When a user successfully logs in, a JWT is generated on the server and sent back to the client.

- Authorization: The client includes the JWT in subsequent requests to the server. The server verifies the JWT to authenticate the user and authorize access to protected resources.

- Expiration: JWTs can optionally have an expiration time, after which they are no longer considered valid.

2. Implementation in Express:

- Installation: First, you need to install the necessary packages. Common choices include `jsonwebtoken` for creating and verifying JWTs and `express-jwt` for middleware to handle JWT authentication.

- Generation: When a user logs in, create a JWT containing relevant user information (e.g., user ID, username) and sign it using a secret key known only to the server.

- Sending JWT to Client: Send the JWT back to the client, typically in the response body or as a cookie.

- Verification Middleware: Create middleware to verify incoming JWTs on protected routes. This middleware will check if the JWT is valid and extract the decoded payload for further processing.

- Protected Routes: Apply the verification middleware to routes that require authentication. This ensures that only users with a valid JWT can access protected resources.

- Token Expiration: Optionally, you can include an expiration time in the JWT payload to enforce session limits. Clients will need to refresh their tokens periodically to maintain access.

Here's a basic example of JWT authentication implementation in Express:

```
const jwt = require('jsonwebtoken');
const expressJwt = require('express-jwt');
const express = require('express');
const app = express();

// Secret key for signing JWTs
const secretKey = 'your_secret_key';

// Middleware to verify JWT
app.use(expressJwt({ secret: secretKey }).unless({ path: ['/login'] }));

// Login route
app.post('/login', (req, res) => {
  // Assuming user authentication is successful
```

```
    const userId = 'user123'; // Retrieve user ID from database
    const token = jwt.sign({ userId }, secretKey);
    res.json({ token });
});

// Protected route
app.get('/protected', (req, res) => {
    // User is authenticated if the JWT verification middleware passes
    res.json({ message: 'Protected resource' });
});

// Error handling middleware for JWT authentication errors
app.use((err, req, res, next) => {
    if (err.name === 'UnauthorizedError') {
        res.status(401).json({ error: 'Unauthorized' });
    }
});

// Start server
app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

In this example, the `/login` route generates a JWT upon successful authentication, and the JWT verification middleware (`expressJwt`) is applied to all routes except `/login`. The `/protected` route is an example of a protected route that requires authentication.

49	How can you restrict access to certain routes or resources based on user authentication status in Express?	8	5

Ans.

In Express, you can restrict access to certain routes or resources based on user authentication status by implementing middleware functions. These middleware functions can be applied to specific routes or globally to your entire application. Here's how you can do it:

1. Authentication Middleware: Create a middleware function to check if the user is authenticated. This function can verify the presence and validity of a JWT, session cookie, or any other authentication mechanism you're using.

2. Apply Middleware: Apply the authentication middleware to the routes or resources that require authentication. You can apply it globally to all routes or selectively to specific routes.

3. Handle Unauthorized Access: If the user is not authenticated, you can handle unauthorized access by sending an appropriate HTTP response (e.g., 401 Unauthorized) or redirecting the user to a login page.

Here's a basic example of how you can implement authentication middleware in Express:

```
// Authentication middleware
const authenticateUser = (req, res, next) => {
  // Check if user is authenticated (e.g., by verifying JWT)
  const isAuthenticated = /* Logic to check authentication */;

  if (isAuthenticated) {
    // User is authenticated, proceed to next middleware
    next();
  } else {
    // User is not authenticated, send 401 Unauthorized response
    res.status(401).json({ error: 'Unauthorized' });
  }
};

// Apply authentication middleware globally to all routes
app.use(authenticateUser);

// Protected route
app.get('/protected', (req, res) => {
  // This route is only accessible to authenticated users
  res.json({ message: 'Protected resource' });
});

// Unprotected route
app.get('/public', (req, res) => {
```

```
  // This route is accessible to all users
  res.json({ message: 'Public resource' });
});
```

In this example, the `authenticateUser` middleware function checks if the user is authenticated. If the user is authenticated, it calls `next()` to proceed to the next middleware in the request-handling pipeline. If the user is not authenticated, it sends a 401 Unauthorized response.

The `authenticateUser` middleware is applied globally to all routes using `app.use(authenticateUser)`. This means that all routes defined after this middleware will require authentication. You can also apply the middleware selectively to specific routes by calling `app.use(authenticateUser)` before defining those routes.

50	Develop a React App with Sign-up & Login Components and implement Session Management using Cookies.	8	5

Ans.

Creating a React app with sign-up and login components and implementing session management using cookies involves several steps. Here's a step-by-step guide to get you started:

Step 1: Set up the React App

First, create a new React app using Create React App.

```
npx create-react-app react-auth-app
cd react-auth-app
```

Step 2: Install Dependencies

Install the necessary dependencies. We'll use axios for HTTP requests and js-cookie for handling cookies.

```
npm install axios js-cookie
```

Step 3: Create Sign-up and Login Components

Create two components, Signup.js and Login.js, for user registration and login.

```
//Signup.js
import React, { useState } from 'react';
import axios from 'axios';

const Signup = () => {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
```

```
  const handleSubmit = async (event) => {
    event.preventDefault();
    try {
      const response = await axios.post('/api/signup', { email, password });
      console.log(response.data);
    } catch (error) {
      console.error('Error signing up', error);
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <h2>Sign Up</h2>
      <input
        type="email"
        placeholder="Email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
      />
      <input
        type="password"
        placeholder="Password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
      />
      <button type="submit">Sign Up</button>
    </form>
  );
};

export default Signup;

//Login.js
import React, { useState } from 'react';
```

```
import axios from 'axios';
import Cookies from 'js-cookie';

const Login = () => {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');

  const handleSubmit = async (event) => {
    event.preventDefault();
    try {
      const response = await axios.post('/api/login', { email, password });
      Cookies.set('session_token', response.data.token, { expires: 1 });
      console.log('Logged in successfully');
    } catch (error) {
      console.error('Error logging in', error);
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <h2>Login</h2>
      <input
        type="email"
        placeholder="Email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
      />
      <input
        type="password"
        placeholder="Password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
      />
      <button type="submit">Login</button>
    </form>
```

```
  );
};

export default Login;
```

Step 4: Implement Session Management

Create a higher-order component (HOC) or a context to manage the user session.

```
//AuthContext.js
import React, { createContext, useState, useEffect } from 'react';
import Cookies from 'js-cookie';
import axios from 'axios';

export const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);

  useEffect(() => {
    const token = Cookies.get('session_token');
    if (token) {
      axios
        .get('/api/me', { headers: { Authorization: `Bearer ${token}` } })
        .then((response) => setUser(response.data))
        .catch(() => Cookies.remove('session_token'));
    }
  }, []);

  return (
    <AuthContext.Provider value={{ user, setUser }}>
      {children}
    </AuthContext.Provider>
  );
};
```

```
//ProtectedRoute.js
import React, { useContext } from 'react';
import { Route, Redirect } from 'react-router-dom';
import { AuthContext } from './AuthContext';

const ProtectedRoute = ({ component: Component, ...rest }) => {
  const { user } = useContext(AuthContext);
  return (
    <Route
      {...rest}
      render={(props) =>
        user ? <Component {...props} /> : <Redirect to="/login" />
      }
    />
  );
};

export default ProtectedRoute;
```

Step 5: Set Up Routes

Set up the routes for your app, including protected routes.

```
//App.js
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import { AuthProvider } from './AuthContext';
import Signup from './Signup';
import Login from './Login';
import ProtectedRoute from './ProtectedRoute';

const Dashboard = () => <h2>Dashboard (Protected)</h2>;

const App = () => {
  return (
    <AuthProvider>
```

```
    <Router>
      <Switch>
        <Route path="/signup" component={Signup} />
        <Route path="/login" component={Login} />
        <ProtectedRoute path="/dashboard" component={Dashboard} />
        <Route path="/" component={() => <h2>Home</h2>} />
      </Switch>
    </Router>
  </AuthProvider>
 );
};

export default App;
```

Step 6: Backend API

For the purposes of this example, we'll assume you have a backend API set up with the following endpoints:

- POST /api/signup: To create a new user.
- POST /api/login: To authenticate a user and return a token.
- GET /api/me: To get the authenticated user's details based on the token.

Step 7: Run the App

Run your React app.

```
npm start
```

This is a basic implementation. In a real-world scenario, you'd need to add error handling, form validation, and security measures such as HTTPS, secure cookies, and potentially more advanced session management strategies.