**Week-7 Hands-On(React)**

14.ReactJS-HOL:

Objectives:

**Explain the need and Benefits of React Context API**

The React Context API is designed to solve the problem of **prop drilling**, which occurs when data needs to be passed down through many levels of nested components, even if intermediate components don’t need that data. Instead of manually passing props through each component, Context allows you to share values like theme, authentication status, or user preferences directly across the component tree. The benefits of using the Context API include: making your code cleaner, improving maintainability, avoiding repetitive prop passing, and making it easier to manage global or shared state across the application without introducing complex state management libraries for small to medium projects.

**Working with createContext()**

To use the React Context API, you start by calling React.createContext(), which creates a Context object. This object has two main components: a **Provider** and a **Consumer**. The Provider component is used to wrap the part of your component tree that should have access to the context, and it accepts a value prop which holds the data you want to share. The Consumer component (or the useContext hook in functional components) is used inside any descendant to access that shared data.

**List the types of Router Components**

When working with **React Router**, there are several important router components that help define your application’s routing structure. The main types include:

* **BrowserRouter**: Uses the HTML5 history API to keep the UI in sync with the URL; suitable for most web apps.
* **HashRouter**: Uses the hash portion of the URL (example.com/#/about); useful for older browsers or static file hosting.
* **MemoryRouter**: Keeps the history of your “URL” in memory without affecting the actual browser URL; often used for testing or non-browser environments.  
  Additionally, there are other components like **Routes**, **Route**, **Link**, **NavLink**, and **Outlet** that help configure individual routes, create navigation links, and render nested routes.

**ThemeContext.js**

import React from 'react';

const ThemeContext = React.createContext('light');

export default ThemeContext;

**App.js**

import React, { useState } from 'react';

import EmployeeList from './EmployeeList';

import ThemeContext from './ThemeContext';

function App() {

const [theme, setTheme] = useState('light');

return (

<ThemeContext.Provider value={theme}>

<div>

<h1>Employee Management System</h1>

<button onClick={() => setTheme(theme === 'light' ? 'dark' : 'light')}>

Toggle Theme

</button>

<EmployeeList />

</div>

</ThemeContext.Provider>

);

}

export default App;

**EmployeeList.js**

import React from 'react';

import EmployeeCard from './EmployeeCard';

function EmployeeList() {

const employees = [

{ id: 1, name: 'John', role: 'Developer' },

{ id: 2, name: 'Jane', role: 'Designer' },

{ id: 3, name: 'Mike', role: 'Tester' }

];

return (

<div>

<h2>Employees</h2>

{employees.map(emp => (

<EmployeeCard key={emp.id} employee={emp} />

))}

</div>

);

}

export default EmployeeList;

**EmployeeCard.js**

import React, { useContext } from 'react';

import ThemeContext from './ThemeContext';

function EmployeeCard({ employee }) {

const theme = useContext(ThemeContext);

const buttonStyle = {

backgroundColor: theme === 'dark' ? '#333' : '#f4f4f4',

color: theme === 'dark' ? '#fff' : '#000',

padding: '5px 10px',

border: 'none',

marginTop: '5px',

cursor: 'pointer'

};

return (

<div style={{ marginBottom: '15px', padding: '10px', border: '1px solid #ccc' }}>

<p><strong>Name:</strong> {employee.name}</p>

<p><strong>Role:</strong> {employee.role}</p>

<button style={buttonStyle}>Message</button>

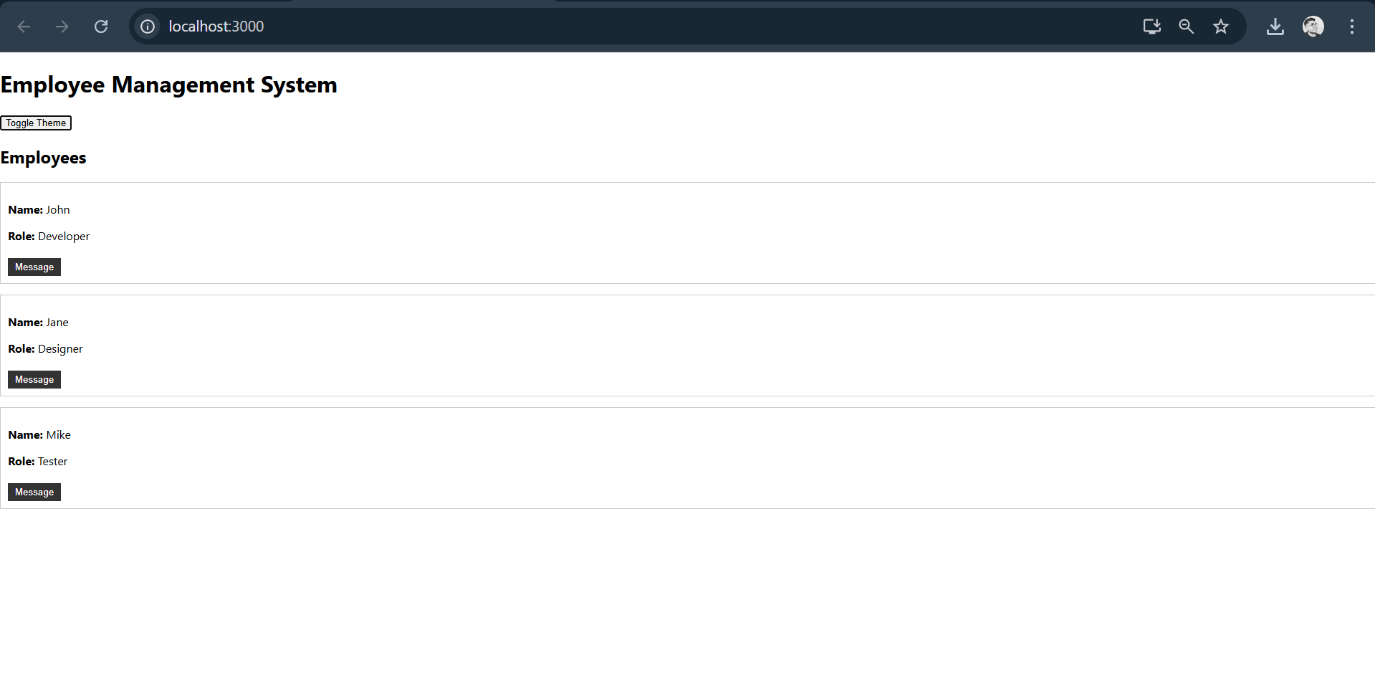
</div>

);

}

export default EmployeeCard;

Output:



15.ReactJS-HOL:

Objectives:

**Explain about React forms**

React forms are used to collect user input within React applications, just like traditional HTML forms. However, in React, forms are typically managed using **component state**, which keeps the form data in sync with what the user sees and what is stored internally. This approach makes it easier to validate data, dynamically change form behavior, and handle complex interactions, since you have direct control over the form’s values through React’s declarative model.

**Define controlled components**

In React, a controlled component is a form input element (such as <input>, <textarea>, or <select>) whose value is **controlled by React state**. This means the displayed value of the input is always driven by the state, and any user input triggers an event handler (usually onChange) that updates the state. This design makes it easier to manage and validate user input because React becomes the single source of truth for the form data.

**Explain about various input controls**

React supports the same input controls you’d find in regular HTML, such as textboxes (<input type="text">), password fields (<input type="password">), checkboxes, radio buttons, textareas, and dropdown menus (<select>). Each of these can be used as controlled components by setting their value (or checked for checkboxes) and providing an onChange handler to manage their state in the component.

**Explain about handling forms**

Handling forms in React involves managing form input values with component state and responding to user actions like typing or selecting. Typically, each input has an onChange event handler that updates the corresponding state variable. This ensures that the form’s displayed values always reflect the component’s state, enabling features like live validation, conditional rendering, and dynamic form fields based on user input.

**Explain about handling forms**

Handling forms in React involves managing form input values with component state and responding to user actions like typing or selecting. Typically, each input has an onChange event handler that updates the corresponding state variable. This ensures that the form’s displayed values always reflect the component’s state, enabling features like live validation, conditional rendering, and dynamic form fields based on user input.

**ComplaintRegister.js**

import React, { useState } from 'react';

function ComplaintRegister() {

const [employeeName, setEmployeeName] = useState('');

const [complaint, setComplaint] = useState('');

const handleSubmit = (e) => {

e.preventDefault();

const referenceNumber = Date.now();

alert(`Complaint submitted! Reference Number: ${referenceNumber}`);

// Reset the form fields after submission

setEmployeeName('');

setComplaint('');

};

return (

<div style={{ margin: '20px' }}>

<h2>Complaint Register</h2>

<form onSubmit={handleSubmit}>

<div>

<label>Employee Name: </label>

<input

type="text"

value={employeeName}

onChange={(e) => setEmployeeName(e.target.value)}

required

/>

</div>

<div>

<label>Complaint: </label>

<textarea

value={complaint}

onChange={(e) => setComplaint(e.target.value)}

required

/>

</div>

<button type="submit">Submit</button>

</form>

</div>

);

}

export default ComplaintRegister;

**App.js**

import React from 'react';

import ComplaintRegister from './ComplaintRegister';

function App() {

return (

<div>

<ComplaintRegister />

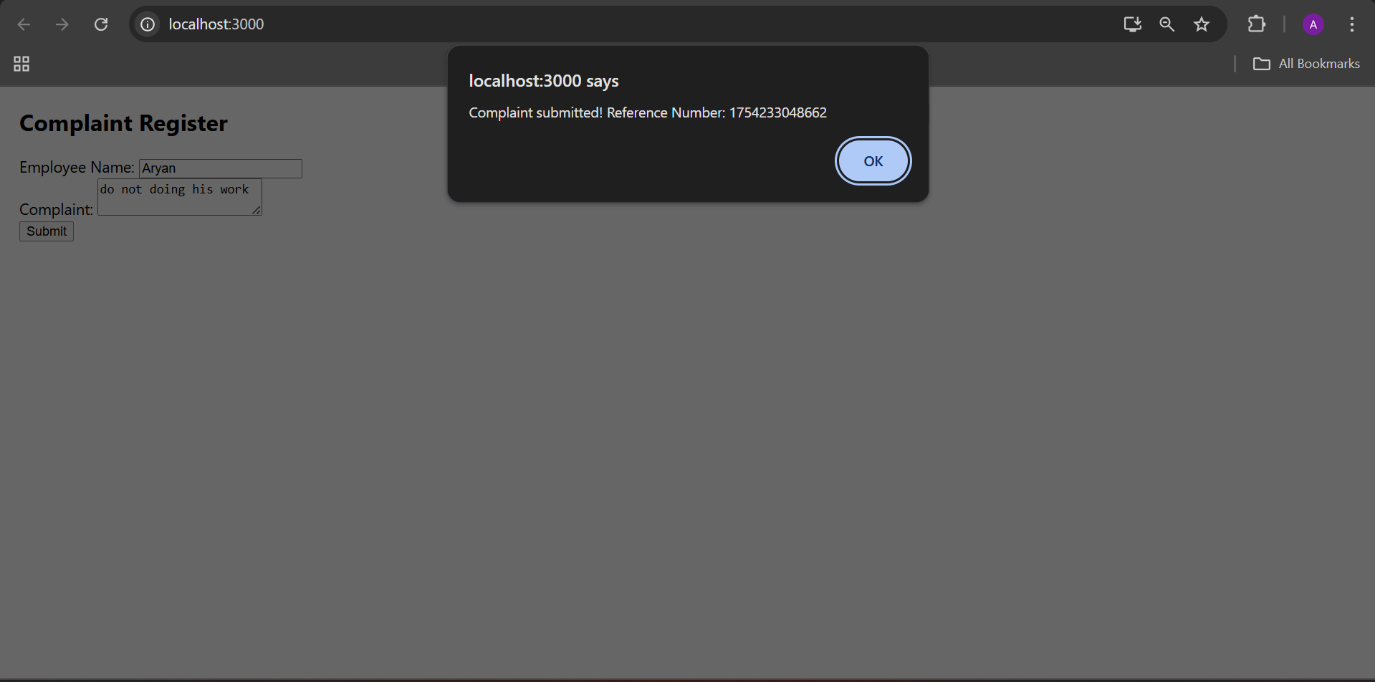
</div>

);

}

export default App;

Output:



16.ReactJS-HOL:

Objectives:

**Explain React Forms validation**

React form validation is the process of checking user input against defined rules to ensure the data is correct before submission. Since React typically manages form data in component state, validation logic can be implemented in event handlers like onChange or onSubmit. Developers often write custom validation functions to check conditions (like required fields, pattern matching, or length checks), and display validation messages dynamically in the UI. This gives full control over when and how validation occurs, enabling real-time feedback and smoother user experience.

**Identify the differences between React Form and HTML Form**

Traditional HTML forms directly manage input values within the DOM and rely on default browser behavior for submission, which refreshes the page. In contrast, React forms are usually implemented as **controlled components** where the form input values are stored in component state and updated via React event handlers. React also handles form submission through onSubmit handlers without reloading the page. This makes React forms more interactive, dynamic, and suitable for single-page applications, giving developers finer control over form data and validation.

**Explain about controlled components**

Controlled components are form elements in React whose values are **controlled by React state** rather than the DOM. This means the displayed value of an input field is always determined by the component’s state, and any user changes trigger an onChange event handler that updates this state. Controlled components help keep form data consistent, enable live validation, and simplify logic that depends on user input since the state is the single source of truth.

**Identify various React Form input controls**

React supports all standard HTML form input controls, which can be used as controlled components. These include textboxes (<input type="text">), password fields (<input type="password">), checkboxes, radio buttons, textareas (<textarea>), and dropdowns (<select>). Each input control typically has an associated value or checked state managed in React component state, allowing the app to respond dynamically to user input.

**Identify various React Form input controls**

React supports all standard HTML form input controls, which can be used as controlled components. These include textboxes (<input type="text">), password fields (<input type="password">), checkboxes, radio buttons, textareas (<textarea>), and dropdowns (<select>). Each input control typically has an associated value or checked state managed in React component state, allowing the app to respond dynamically to user input.

**Explain about submitting forms in React**

In React, form submission is handled by adding an onSubmit event handler to the <form> element. Inside this handler, developers usually call event.preventDefault() to stop the default page reload behavior. The submission logic then uses the component’s state (which holds the form data) to validate the inputs, display error messages if needed, or send the data to a server using APIs like fetch or axios. This process keeps form handling smooth, interactive, and aligned with single-page application principles.

**Explain how to handle React Forms**

Handling forms in React usually involves setting up state variables for each input control and updating them with onChange event handlers. When a user types or interacts with a control, the handler updates the corresponding state value. This approach keeps the UI and data synchronized, simplifies validation, and makes it easier to dynamically enable/disable controls, show messages, or update related data based on user input.

**register.js**

import React, { useState } from 'react';

function Register() {

const [name, setName] = useState('');

const [email, setEmail] = useState('');

const [password, setPassword] = useState('');

const [errors, setErrors] = useState({});

const validate = () => {

let newErrors = {};

if (name.length < 5) {

newErrors.name = 'Name must have at least 5 characters';

}

if (!email.includes('@') || !email.includes('.')) {

newErrors.email = 'Email must include @ and .';

}

if (password.length < 8) {

newErrors.password = 'Password must have at least 8 characters';

}

return newErrors;

};

const handleSubmit = (e) => {

e.preventDefault();

const validationErrors = validate();

if (Object.keys(validationErrors).length > 0) {

setErrors(validationErrors);

} else {

alert('Form submitted successfully!');

setErrors({});

// Reset fields

setName('');

setEmail('');

setPassword('');

}

};

return (

<div style={{ margin: '20px' }}>

<h2>Register Form</h2>

<form onSubmit={handleSubmit}>

<div>

<label>Name: </label>

<input

type="text"

value={name}

onChange={(e) => setName(e.target.value)}

/>

{errors.name && <div style={{ color: 'red' }}>{errors.name}</div>}

</div>

<div>

<label>Email: </label>

<input

type="email"

value={email}

onChange={(e) => setEmail(e.target.value)}

/>

{errors.email && <div style={{ color: 'red' }}>{errors.email}</div>}

</div>

<div>

<label>Password: </label>

<input

type="password"

value={password}

onChange={(e) => setPassword(e.target.value)}

/>

{errors.password && <div style={{ color: 'red' }}>{errors.password}</div>}

</div>

<button type="submit">Register</button>

</form>

</div>

);

}

export default Register;

**App.js**

import React from 'react';

import Register from './register';

function App() {

return (

<div>

<Register />

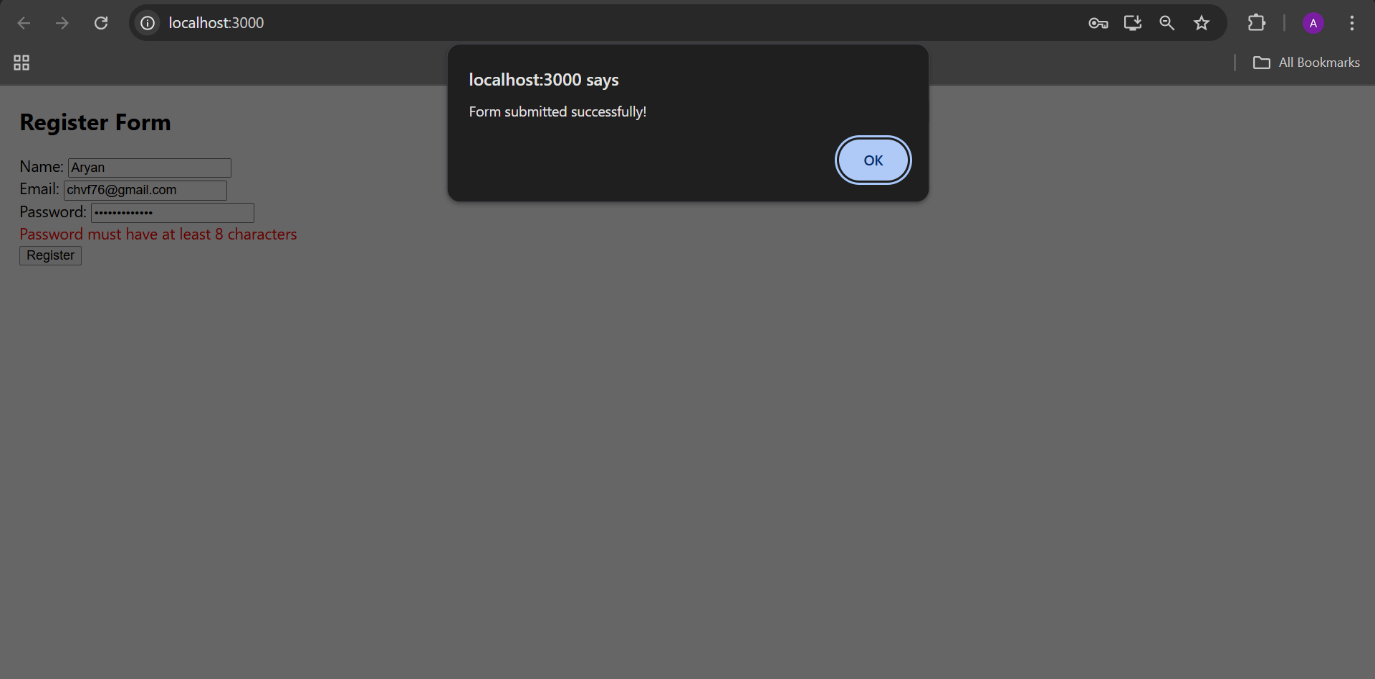
</div>

);

}

export default App;

Output:



17.ReactJS-HOL:

Objectives:

**Explain how to consume REST APIs from React applications**

To consume REST APIs in React applications, developers typically use JavaScript's built-in fetch function or libraries like axios to make HTTP requests. The recommended practice is to perform these API calls inside lifecycle methods like componentDidMount for class components or inside the useEffect hook for functional components. Once the data is fetched asynchronously, it is stored in the component's state using useState or this.setState, which then updates the UI. This method keeps the UI dynamic and responsive to data changes from external services, enabling React apps to display real-time or frequently updated information from APIs.

**Getuser.js**

import React, { Component } from 'react';

class Getuser extends Component {

constructor(props) {

super(props);

this.state = {

user: null,

};

}

componentDidMount() {

fetch('https://api.randomuser.me/')

.then((response) => response.json())

.then((data) => {

this.setState({ user: data.results[0] });

})

.catch((error) => console.error('Error fetching user:', error));

}

render() {

const { user } = this.state;

return (

<div style={{ margin: '20px' }}>

<h2>User Details</h2>

{user ? (

<div>

<p>Title: {user.name.title}</p>

<p>First Name: {user.name.first}</p>

<img src={user.picture.large} alt="User" />

</div>

) : (

<p>Loading...</p>

)}

</div>

);

}

}

export default Getuser;

**App.js**

import React from 'react';

import Getuser from './Getuser';

function App() {

return (

<div>

<Getuser />

</div>

);

}

export default App;

Output:

