REACTJS LEARNING MODULE

Day 1

Introduction to React

What is React? Benefits and Use Cases

React is a JavaScript library for building user interfaces (UI), developed by Facebook (Meta). It follows a component-based architecture and enables developers to build fast and interactive web applications.

Benefits of React

✓ **Component-Based Architecture** – Code is divided into reusable components.

✓ **Virtual DOM** – React uses a virtual representation of the DOM to optimize updates, improving performance.

✓ **Declarative Syntax** – Developers describe UI state, and React updates the UI efficiently.

✓ **Unidirectional Data Flow** – Data flows in one direction, making debugging easier.

✓ **Strong Community & Ecosystem** – Many third-party libraries and a vast developer community.

Use Cases of React

- **Single Page Applications (SPAs)** – Apps like Facebook, Instagram, and Twitter.
- **Dashboards & Data Visualization** – Interactive and dynamic user interfaces.

- **E-commerce Platforms** – Fast and scalable online shopping experiences.
- **Mobile Applications** – React Native extends React's capabilities for mobile apps.

Understanding the React Ecosystem

1. **ReactDOM**

ReactDOM is a package that handles rendering React components into the actual DOM.

Example: Rendering a component into the root HTML element.

```
import React from "react";
import ReactDOM from "react-dom";

const App = () => <h1>Hello, React!</h1>;

ReactDOM.createRoot(document.getElementById("root")).render(<App  />);
```

2. JSX (JavaScript XML)

JSX is a syntax extension for JavaScript that looks like HTML but allows JavaScript logic inside.

Example:

```
const element = <h1>Hello, {user.name}!</h1>;
```

JSX gets compiled into standard JavaScript:

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

3. Components in React

Components are reusable building blocks in React.

Types of Components:

- **Functional Components** – Uses functions and hooks.
- **Class Components** – Uses ES6 classes (older approach).

Example of a Functional Component:

```
const Welcome = () => {
  return <h1>Hello, Welcome to React Class</h1>;
};

export default Welcome;
```

Setting Up a React Development Environment

Using Create React App (CRA)

Create React App (CRA) is a tool that sets up a new React project with all configurations.

Steps to Set Up a React Project:

1) Install Node.js (if not already installed) from Node.js Official Website 2) Open a terminal and run:

```
npx create-react-app my-app
```

or

```
npm init react-app my-app
```

3) Navigate to the project folder:

```
cd my-app
```

4) Start the development server:

```
npm start
```

5) Open http://localhost:3000/ in your browser to see your React app running.

Day -1 Tasks : -

1. **Introduction to React**: Write down what React is in your own words.
2. **React Advantages**: List at least 5 benefits of using React.
3. **React Ecosystem**: Describe ReactDOM, JSX, and Components briefly.
4. **Install Node.js & npm**: Install Node.js and check the version using the terminal.
5. **Create React App Setup**: Initialize a React app using npx create-react-app my-app.
6. **Folder Structure**: Explain the purpose of the src, public, and node_modules folders.
7. **JSX Practice**: Write a simple JSX code that renders "Welcome to React".
8. **ReactDOM.render()**: Use ReactDOM to render a component to the browser.
9. **Component Creation**: Create a basic functional component that displays a message.
10. **Props Implementation**: Pass a prop to a component that displays a user's name.
11. **Inline CSS Styling**: Apply inline CSS to change the text color of a component.
12. **CSS Class Styling**: Use an external CSS file to style a heading.

13. **Component Composition**: Create two components and render them inside a parent component.

Mini Projects

Mini Project 1: Personal Bio

Description:
Create a component that displays:

- Your name
- Age
- A short bio
  Use props to pass the data and style the text using external CSS.

Mini Project 2: Greeting App

Description:
Create a component that accepts a name as a prop and displays "Hello, [Name]!" with a background color applied via inline CSS.

Mini Project 3: Simple Counter

Description:
Create a component that displays a number and a button to increase the number (Just UI without functionality).

Mini Project 4: Hobbies List

Description:
Render a list of hobbies using an array passed as props. Apply basic CSS for list styling.

Day 2:

JSX Syntax & Rendering Elements

JSX Syntax in React

JSX (JavaScript XML) is a syntax extension for JavaScript that allows writing HTML-like code inside JavaScript. React uses JSX to describe what the UI should look like.

1. Embedding Expressions in JSX

You can embed JavaScript expressions inside JSX by using {} curly braces.

Example: Using Variables in JSX

```
const name = "John";
const age = 25;

const element = <h1>Hello, {name}! You are {age} years old.</h1>;
```

JSX Needs a Parent Element

JSX expressions must have a single parent element. Use a fragment (<>...</>) or a to wrap multiple elements.

```
return (
  <>
    <h1>Welcome</h1>
    <p>This is a React component.</p>
  </>
);
```

Example: Using Functions Inside JSX

```
const getGreeting = (user) => {
  return user ? `Hello, ${user}!` : "Hello, Guest!";
};

const element = <h1>{getGreeting("Alice")}</h1>;
```

Example: Using Ternary Operators Inside JSX

```
const isLoggedIn = true;

const element = (
  <h1>{isLoggedIn ? "Welcome back!" : "Please log in."}</h1>
);
```

2. JSX Attributes

JSX attributes work similarly to HTML attributes but use camelCase instead of lowercase.

Example: Adding HTML Attributes

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

Example: Adding Class with className

JSX uses className instead of class for CSS classes.

```
const element = <h1 className="heading">Hello, React!</h1>;
```

Example: Using JavaScript Expressions in Attributes

```
const imageUrl = "logo.png";
const element = <img src={imageUrl} alt="Dynamic Image" />;
```

3. Styling in JSX

There are three ways to style elements in React:

(a) Inline Styling

Inline styles use **JavaScript objects** with camelCase properties.

```
const headingStyle = {
  color: "blue",
  fontSize: "24px",
  textAlign: "center",
};

const element = <h1 style={headingStyle}>Styled Heading</h1>;
```

(b) External CSS

Create a styles.css file:

```
.title {
  color: red;
  font-size: 20px;
}
```

Import and use it in JSX:

```
import "./styles.css";
const element = <h1 className="title">Styled with External CSS</h1>;
```

(c) Using CSS Modules

Create a CSS module styles.module.css:

```
.title {
  color: green;
  font-size: 22px;
}
```

Import and use it in JSX:

```
import styles from "./styles.module.css";
const element = <h1 className={styles.title}>Styled with CSS Modules</h1>;
```

4. Rendering React Elements with ReactDOM.render()

React elements are rendered into the real DOM using ReactDOM.render(). However, in modern React (React 18+), we use ReactDOM.createRoot() instead.

Example: Rendering a React Element

```
import React from "react";
import ReactDOM from "react-dom";
const element = <h1>Hello, React!</h1>;
ReactDOM.createRoot(document.getElementById("root")).render(element);
```

Example: Rendering a Functional Component

```
const Welcome = () => {
  return <h1>Welcome to React!</h1>;
};
ReactDOM.createRoot(document.getElementById("root")).render(<Welcome  />);
```

Day -2 Tasks : -

1. **JSX Introduction**: Write a brief explanation of JSX and its importance in React.
2. **Simple JSX Element**: Create a JSX element that displays "Hello, React!".
3. **Embedding Expressions**: Write JSX that displays the result of 5 + 5.
4. **Using Variables in JSX**: Create a variable with your name and display it in JSX.
5. **Attributes in JSX**: Add the className attribute to a JSX element with a CSS class.
6. **Inline Styling**: Apply inline styles to a heading using the style attribute in JSX.
7. **Class-based Styling**: Create an external CSS file and apply the styles using className.
8. **ReactDOM.render() Usage**: Render a simple JSX element to the browser.
9. **Self-closing Tags**: Use a self-closing tag like <img /> inside JSX.
10. **JSX Comments**: Add comments inside JSX code.
11. **JSX with Functions**: Write a function that returns a JSX element and render it.
12. **Multiple Elements in JSX**: Use a fragment or div to render multiple JSX elements.
13. **Conditional Rendering with JSX**: Display a message only if a variable is true.

Mini Projects

Mini Project 1: Simple Profile Card

Description:
Create a profile card component that displays:

- Name
- Age

- Location
  apply both inline and external styling.

Mini Project 2: Temperature Display

Description:
Create a component that accepts a temperature value displays:

- "Hot" if the temperature is greater than 30°C
- "Cold" if the temperature is less than 15°C
- "Normal" for temperatures in between

Mini Project 3: Simple Calculator

Description:
Create a calculator component that accepts two numbers displays their sum.

Mini Project 4: User Avatar

Description:
Render an image component that accepts a URL and alt text as props and displays the image with a caption.

Day 3:

React Components - Functional vs. Class

In React, components are the **building blocks** of an application. They can be created using **functional components** or **class components**.

1. Creating and Using Functional Components

A **functional component** is a **JavaScript function** that returns **JSX** (UI elements). It's the modern way of writing React components and is **simpler, faster, and easy to manage**.

Syntax of a Functional Component:

```
const Welcome = () => {
  return <h1>Hello, React!</h1>;
};

export default Welcome;
```

Using Functional Components Inside App.js

```
import Welcome from "./Welcome";

function App() {
  return (
    <div>
      <Welcome />
    </div>
```

```
  );
}
export default App;
```

Key Features of Functional Components:

✓ **Simple Syntax** – Just a function that returns JSX.

✓ **Hooks Support** – Use useState, useEffect, etc., for state and lifecycle management.

✓ **Stateless by Default** – But can manage state with hooks.

✓ **Easier to Read & Maintain** – Preferred in modern React development.

2. Class Components & Their Syntax

A **class component** is an **ES6 class** that extends React.Component. It uses a render() method to return JSX.

Syntax of a Class Component:

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

class Welcome extends Component {
  render() {
    return <h1>Hello, React from a Class Component!</h1>;
  }
}

export default Welcome;
```

Using Class Components Inside App.js

```
import Welcome from "./Welcome";

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

export default App;
```

Key Features of Class Components:

✓ **Uses ES6 Class Syntax** – Extends React.Component.
✓ **Has Lifecycle Methods** – componentDidMount(), componentDidUpdate(), etc.
✓ **Uses this.state for State Management** – No hooks, state is managed using this.setState().
✓ **More Boilerplate Code** – Requires a constructor and this keyword.

Comparing Functional vs. Class Components

Feature	Functional Components ✅	Class Components
Syntax	Function-based	Class-based (extends React.Component)
State Handling	useState hook	this.state & setState()
Lifecycle Methods	useEffect hook	Uses lifecycle methods like componentDidMount()
Performance	Faster & lightweight	Slightly slower due to this binding
Code Complexity	Simple & concise	More boilerplate code
Recommended?	✅ Yes (modern React)	⚠ Only for legacy code

Day -3 Tasks : -

1. **Functional Component Creation**: Create a simple functional component that displays "Welcome to React!".
2. **Functional Component with Static Text**: Display a static message like "Learning React is fun!" inside a functional component.
3. **Render Multiple Functional Components**: Render three functional components displaying different messages.
4. **Basic Class Component**: Create a class component that displays "This is a Class Component".
5. **Class Component with Static Text**: Display "Hello from Class Component!" inside the class component.
6. **Render Multiple Class Components**: Render two class components with different static text messages.

7. **State in Class Component**: Create a class component that initializes a counter with 0 and displays it.
8. **State Update in Class Component**: Add a button to the counter component that increments the value on click.
9. **JSX Styling in Functional Component**: Apply inline styling to a functional component message.
10. **External CSS Styling in Class Component**: Use an external CSS file to style the text in a class component.
11. **Conditional Rendering in Functional Component**: Display "React is Cool!" only if a boolean variable is true.
12. **Render Nested Components**: Render one functional component inside another functional component.
13. **Lifecycle Method in Class Component**: Use componentDidMount() to display a message in the console when the class component is mounted.

Mini Projects

Mini Project 1: Welcome Message

Description:
Create a functional component that displays "Welcome to React Learning!" with some inline styling.

Mini Project 2: Click Counter

Description:
Create a class component that initializes a counter to 0 and increments the counter when a button is clicked.

Mini Project 3: Show/Hide Text

Description:
Build a functional component that displays "Hello, World!" only if a boolean variable is true.

Mini Project 4: Class Component Timer

Description:
Create a class component that displays the current time and updates every second using componentDidMount() and setInterval().

Day 4:

Props - Passing Data to Components

What are Props in React?

Props (short for **"properties"**) allow you to pass data **from a parent component to a child component**. Props make components **reusable and dynamic**.

React components use `props` to communicate with each other. Every parent component can pass some information to its child components by giving them props. Props might remind you of HTML attributes, but you can pass any JavaScript value through them, including objects, arrays, and functions.

1) Passing Data from a Parent to a Child Component

Example: Passing a name from a Parent to Child component using props.

Parent Component (App.js)

```
import React from "react";
import Child from "./Child";
const App = () => {
  return <Child name="Alice" />;
};
export default App;
```

Child Component (Child.js)

```
const Child = (props) => {
  return <h2>Hello, {props.name}!</h2>;
};
export default Child;
```

2) Destructuring Props

Instead of using props.name, you can **destructure** props for cleaner code.

Example:

```
const Child = ({ name }) => {
  return <h2>Hello, {name}!</h2>;
};
export default Child;
```

3) Default Props in React

If a prop is **not provided**, you can set **default values** using defaultProps.

Example:

```
const Welcome = ({ name = "Guest" }) => {
  return <h2>Welcome, {name}!</h2>;
};
export default Welcome;
```

🔹 OR using defaultProps

```
Welcome.defaultProps = {
  name: "Guest",
};
```

🔹 Use in App.js (without passing a name)

```
<Welcome />
```

Output: "Welcome, Guest!"

4) Passing JSX as Props

You can **pass JSX elements** as props for more flexibility.

Example:

```
const Card = ({ content }) => {
  return <div className="card">{content}</div>;
};
export default Card;
```

🔹 Use in App.js

```
<Card content={<h3>This is a Card</h3>} />
```

Output: A card displaying "This is a Card"

5) Passing Functions as Props (Callback Props)

A parent component can pass a **function** as a prop to allow the child to communicate back.

Example:

Parent Component (App.js)

```
import Button from "./Button";
const App = () => {
  const showAlert = () => {
    alert("Button clicked!");
  };
  return <Button onClick={showAlert} />;
};
export default App;
```

Child Component (Button.js)

```
const Button = ({ onClick }) => {
  return <button onClick={onClick}>Click Me</button>;
};

export default Button;
```

Day -4 Tasks : -

1. **What are Props?**: Write a short note on props and why they are used in React functional components.
2. **Simple Text Prop**: Create a functional component that accepts a message prop and displays it.
3. **Multiple Props**: Create a functional component that accepts name and age props and displays them together.
4. **Destructuring Props (Basic)**: Use destructuring to access name and city props inside a functional component.
5. **Boolean Prop**: Pass a boolean prop like isStudent to display "Student" or "Not a Student".
6. **Default Props**: Set default props to display "Guest" when no name is provided.
7. **List Rendering with Props**: Pass an array of hobbies as props and display them using the map() method.
8. **Object Props**: Pass a user object containing name, email, and phone, then display each property.
9. **Conditional Rendering with Props**: Display "Welcome, [Name]" if the name prop is provided, otherwise display "Welcome, Guest".
10. **Inline Styling with Props**: Pass a color prop and apply it to style the text dynamically.
11. **Button Click Event with Props**: Pass a function as a prop to display an alert message on button click.

12. **Reusable Card Component**: Create a card component that displays a title and description passed as props.
13. **Child Component Communication**: Create a parent component that passes text data to two child components for display.

Mini Projects

Mini Project 1: Profile Card

Description:
Create a Profile component that displays:

- Name
- Age
- City
  Use destructuring and default props to display the information.

Mini Project 2: Task List

Description:
Create a functional component that accepts a list of tasks as props and displays each task in an unordered list.

Mini Project 3: Colorful Text

Description:
Create a functional component that accepts text and color props, then displays the text in the given color.

Mini Project 4: Click Counter

Description:
Create a functional component that accepts an onClick function as a prop and

displays a button. Clicking the button should call the function and display "Button Clicked" in the console.

Day 5:

Conditional and List Rendering

1) Rendering Lists in React (Using map())

You will often want to display multiple similar components from a collection of data. You can use the JavaScript array methods to manipulate an array of data. On this page, you'll use filter() and map() with React to filter and transform your array of data into an array of components.

Example 1: Rendering a Simple List

```
const FruitsList = () => {
  const fruits = ["Apple", "Banana", "Orange", "Mango"];
  return (
    <ul>
      {fruits.map((fruit, index) => (
        <li key={index}>{fruit}</li>
      ))}
    </ul>
  );
};
export default FruitsList;
```

Key Points:

✓ Always provide a unique key when rendering lists (e.g., index).

✓ .map() loops through the array and returns JSX elements.

Example 2: Rendering a List of Objects

```
const UsersList = () => {
  const users = [
    { id: 1, name: "Alice", age: 25 },
    { id: 2, name: "Bob", age: 30 },
    { id: 3, name: "Charlie", age: 28 },
  ];

  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>
          {user.name} - {user.age} years old
        </li>
      ))}
    </ul>
  );
};

export default UsersList;
```

2) Conditional Rendering

Your components will often need to display different things depending on different conditions. In React, you can conditionally render JSX using JavaScript syntax like if statements, &&, and ? : operators.

a) Conditional Rendering using **&&** (Short Circuit Evaluation)

The && operator helps render content only if a condition is true.

Example: Show a Message Only if Logged In

```
const WelcomeMessage = ({ isLoggedIn }) => {
  return (
    <div>
      <h2>Welcome to our site</h2>
      {isLoggedIn && <p>You are logged in!</p>}
    </div>
  );
};

export default WelcomeMessage;
```

- If isLoggedIn is true, the message "You are logged in!" will be displayed.
- If isLoggedIn is false, nothing is displayed.

b) Conditional Rendering using Ternary Operator ? :

The ternary operator is useful for if-else conditions in JSX.

Example: Show Different Messages Based on Login Status

```
const UserStatus = ({ isLoggedIn }) => {
  return <h2>{isLoggedIn ? "Welcome Back!" : "Please Log In"}</h2>;
};

export default UserStatus;
```

How it Works?

✓ **If isLoggedIn === true, it renders "Welcome Back!".**

✓ **Otherwise, it renders "Please Log In".**

C) Conditional Rendering using Multiple Returns

Instead of using && or ? :, you can return different JSX based on conditions.

Example: Show Different UI Based on Login Status

```
const Dashboard = ({ isLoggedIn }) => {
  if (!isLoggedIn) {
    return <h2>Please Log In</h2>;
  }

  return <h2>Welcome to Your Dashboard</h2>;
};

export default Dashboard;
```

How it Works?

✓ **If isLoggedIn === false, it** returns early **with "Please Log In".**

✓ **If isLoggedIn === true, it** renders the dashboard**.**

Day -5 Tasks : -

1. **List Rendering Introduction**: Write a short note on how lists are rendered in React using map().
2. **Basic List Rendering**: Render an array of numbers using the map() method inside a functional component.
3. **List with Keys**: Render a list of names with unique keys for each item.
4. **List of Objects**: Render a list of user objects with properties like name and age.
5. **List with Inline Styling**: Render a list of fruits with each item having a different color using inline styling.
6. **Conditional Rendering with &&**: Display "Hello, User!" only if isLoggedIn is true.
7. **Conditional Rendering with Ternary Operator**: Display "Welcome" if isLoggedIn is true, otherwise display "Please Log In".
8. **List with Conditional Rendering**: Render a list of tasks and highlight the completed tasks.
9. **Rendering Empty Lists**: Display "No items available" if the list is empty.
10. **Multiple Returns Based on Condition**: Return "Morning" if time < 12, "Afternoon" if time < 18, and "Evening" otherwise.
11. **List with Click Event**: Display a list of items and show an alert message when an item is clicked.
12. **Toggle List Items**: Create a button that toggles the display of a list on and off.
13. **Filtered List Rendering**: Render only even numbers from an array of numbers.

Mini Projects

Mini Project 1: Student List

Description:
Render a list of student names with their marks. Highlight the names of students who scored more than 50.

Mini Project 2: Task Manager

Description:
Render a list of tasks with a checkbox. Mark the task as completed when the checkbox is clicked.

Mini Project 3: Greeting Message

Description:
Display different greeting messages based on the time of the day using multiple return conditions.

Mini Project 4: Number Filter

Description:
Render a list of numbers. Only display even numbers and apply styling to them.

Day 6:

State in React

1) What is State in React?

State is a built-in React object used to store and manage dynamic data in a component.

Key Features of State:

✓ **Mutable** (Unlike Props, State can be changed).
✓ When **State updates, the component re-renders**.
✓ Managed **inside** the component (local component state).

useState is a React Hook that lets you add a state variable to your component.

Syntax

```
const [state, setState] = useState(initialState)
```

initialState:

The value you want the state to be initially. It can be a value of any type, but there is a special behavior for functions. This argument is ignored after the initial render.

set functions, like setSomething(nextState)

The set function returned by useState lets you update the state to a different value and trigger a re-render. You can pass the next state directly, or a function that calculates it from the previous state:

2) How State Works in React?

Example: Without State (Static Data)

```
const Counter = () => {
  let count = 0;

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => (count += 1)}>Increase</button>
    </div>
  );
};
export default Counter;
```

❌ **Issue:** Clicking the button won't update the count **because React doesn't track the changes**.

3) Using the useState Hook

To make the count dynamic, we use the useState hook.

Example: Counter using useState

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

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

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
};

export default Counter;
```

How it Works?

useState(0) initializes count with 0.

setCount(count + 1) updates the count and re-renders the component.

4) Understanding State vs. Props

Feature	State	Props
Mutability	✅ Mutable (Can change)	❌ Immutable (Cannot change)
Where it's defined?	Inside the component	Passed from parent
Usage	Stores dynamic data	Used for passing data
Triggers Re-render?	✅ Yes	❌ No

Example: State vs Props

```
const Parent = () => {
  return <Child name="Alice" />;
};

const Child = (props) => {
  const [age, setAge] = useState(25);
  return (
    <div>
      <h2>Name: {props.name} (Prop) </h2>
      <h2>Age: {age} (State)</h2>
      <button onClick={() => setAge(age + 1)}>Increase Age</button>
    </div>
  );
};
```

✓ name (prop) comes from Parent and cannot change inside Child.

✓ age (state) is managed inside Child and can change.

Day -6 Tasks : -

1. **What is State?**: Write a short explanation of what state is in React and why it is used.
2. **State vs. Props**: Write three key differences between state and props.
3. **Basic useState Example**: Create a functional component that uses useState to store and display a counter starting at 0.
4. **State Update on Button Click**: Create a button that increments the counter value when clicked.
5. **State with Strings**: Use useState to store and display a user's name. Update the name on button click.
6. **Multiple useState Hooks**: Use two useState hooks to store name and age separately.
7. **Toggle Boolean State**: Create a button that toggles a true/false value in state.
8. **Input Field with State**: Use useState to store input field text and display it below the input field.
9. **Reset State**: Add a button that resets the counter value to 0.
10. **State with Arrays**: Use useState to manage an array of tasks and display them in a list.
11. **Add Item to List**: Add a button that adds a new item to the list stored in state.
12. **Remove Item from List**: Add a button that removes the last item from the list.
13. **State with Conditional Rendering**: Display "List is empty" if the task list is empty using state.

Mini Projects

Mini Project 1: Counter App

Description:
Create a counter that increases, decreases, and resets the value using useState.

Mini Project 2: Name Display

Description:
Create an input field where users can type their name. Display the name below the input field as they type.

Mini Project 3: Task List

Description:
Create a to-do list where users can add tasks and remove the last task using useState.

Mini Project 4: Light Toggle

Description:
Create a button that toggles between "Light ON" and "Light OFF" using state and conditional rendering.

Day 7:

Handling Events

Responding to Events

React lets you add event handlers to your JSX. Event handlers are your own functions that will be triggered in response to interactions like clicking, hovering, focusing form inputs, and so on.

Handling User Input (Click & Form Events)

Click Events

React uses onClick to handle button clicks.

Example: Button Click Event

```
const ClickButton = () => {
  const handleClick = () => {
  alert("Button Clicked!");
  };

  return <button onClick={handleClick}>Click Me</button>;
};

export default ClickButton;
```

✓ Clicking the button triggers handleClick(), showing an alert.

Form Events (onChange, onSubmit)

React handles form inputs using **controlled components** (state stores input values).

Example: Handling Input Fields (onChange)

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

const TextInput = () => {
  const [text, setText] = useState("");

  return (
    <div>
```

```
      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}
      />
      <p>You typed: {text}</p>
    </div>
  );
};

export default TextInput;
```

✓ The onChange event updates the state, keeping the input field controlled.

Example: Handling Form Submission (onSubmit**)**

```
const FormSubmit = () => {
  const handleSubmit = (e) => {
    e.preventDefault(); // Prevent page refresh
    alert("Form Submitted!");
  };
  return (
    <form onSubmit={handleSubmit}>
      <input type="text" placeholder="Enter name" />
      <button type="submit">Submit</button>
    </form>
  );
};

export default FormSubmit;
```

✓ e.preventDefault() prevents the default page reload behavior.

Understanding Event Object & Passing Arguments

React automatically passes an event object (e) to event handlers.

Example: Using Event Object (e)

```
const InputField = () => {
  const handleChange = (e) => {
    console.log("Input Value:", e.target.value);
  };

  return <input type="text" onChange={handleChange} />;
};

export default InputField;
```

✓ e.target.value gets the text typed in the input field.

Example: Passing Arguments to Event Handlers

```
const ButtonClick = () => {
  const handleClick = (name) => {
    alert(`Hello, ${name}`);
  };

  return <button onClick={() => handleClick("Alice")}>Click Me</button>;
};

export default ButtonClick;
```

✓ Why use an arrow function?

If we write onClick={handleClick("Alice")}, it executes immediately instead of waiting for a click.

Using onClick={() => handleClick("Alice")} delays execution until clicked.

Day -7 Task : -

1. **Introduction to Events**: Write a short explanation of event handling in React.
2. **Button Click Event**: Create a button that displays "Button Clicked" in the console when clicked.
3. **Function Binding in Functional Components**: Create a button that calls a function using the onClick event handler.
4. **Inline Event Handler**: Use an inline arrow function to handle button clicks.
5. **Event Object**: Display the event object in the console when a button is clicked.
6. **Passing Arguments to Event Handlers**: Create a button that displays "Hello, John" when clicked, passing "John" as an argument.
7. **Input Change Event**: Display the current value of an input field as the user types.
8. **Form Submit Event**: Create a form with an input field and display the submitted value below the form on submit.
9. **Prevent Default Event**: Use preventDefault() to stop the form from reloading the page on submit.
10. **Toggle Visibility on Button Click**: Show or hide a paragraph when a button is clicked.
11. **Mouse Over Event**: Display "Mouse Over" in the console when hovering over a button.
12. **Keyboard Event**: Display "Key Pressed" in the console when a key is pressed in an input field.
13. **Disable Button on Click**: Disable a button after it is clicked once.

Mini Projects

Mini Project 1: Click Counter

Description:
Create a button that increases the counter value each time it is clicked.

Mini Project 2: Name Form

Description:
Create a form that accepts a name input and displays the name below the form on submission.

Mini Project 3: Show/Hide Paragraph

Description:
Create a button that toggles the visibility of a paragraph using event handlers.

Mini Project 4: Disable After Click

Description:
Create a button that displays "Button Clicked" and becomes disabled after one click.

Day 8:

Review & Mini Project

Mini Project: To-Do List App

Features of the To-Do List App

✓ Users can **add tasks**

✓ Users can **delete tasks**

✓ Task list updates **dynamically**

Step 1: Setup the Project

1) Create a React app using:

```
npx create-react-app todo-app
cd todo-app
npm start
```

2) Inside src/, create a folder components/ for reusable components.

Step 2: Create Components 1 **)** App.js **(Main Component)**

It holds the state for tasks and renders TodoList & TodoForm.

```
import { useState } from "react";
import TodoForm from "./components/TodoForm";
import TodoList from "./components/TodoList";

const App = () => {
  const [tasks, setTasks] = useState([]);
```

```
  const addTask = (task) => {
    setTasks([...tasks, task]);
  };

  const deleteTask = (index) => {
    setTasks(tasks.filter((_, i) => i !== index));
  };

  return (
    <div>
      <h1>To-Do List</h1>
      <TodoForm addTask={addTask} />
      <TodoList tasks={tasks} deleteTask={deleteTask} />
    </div>
  );
};

export default App;
```

2) TodoForm.js **(Form to Add Tasks)**

Accepts user input and adds a task.

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

const TodoForm = ({ addTask }) => {
  const [task, setTask] = useState("");

  const handleSubmit = (e) => {
```

```
    e.preventDefault();
    if (task.trim()) {
    addTask(task);
    setTask("");
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={task}
        onChange={(e) => setTask(e.target.value)}
        placeholder="Enter a task..."
      />
      <button type="submit">Add Task</button>
    </form>
  );
};

export default TodoForm;
```

3) TodoList.js **(Display & Delete Tasks)**

Shows all tasks and deletes a task when clicked.

```
const TodoList = ({ tasks, deleteTask }) => {
  return (
    <ul>
      {tasks.map((task, index) => (
```

```
      <li key={index}>
        {task} <button onClick={() => deleteTask(index)}>Delete</button>
      </li>
    ))}
  </ul>
 );
};
export default TodoList;
```

Step 3: Run & Test the App

Start the project:

```
npm start
```

Add tasks and delete them dynamically!

Day 9:

React Fragment & Keys

1) Using <React.Fragment> Instead of Extra

In React, every component must return a single root element. Often, developers wrap elements in unnecessarily. Instead, we use <React.Fragment> (or shorthand <>...</>) to avoid extra DOM elements.

Example Without <React.Fragment> (Using Extra)

```
const Profile = () => {
  return (
    <div> {/* Unnecessary div */}
      <h2>John Doe</h2>
      <p>Software Developer</p>
    </div>
  );
};
```

Problem: Adds an extra to the DOM, affecting CSS styling and structure.

Solution: Using <React.Fragment>

```
import React from "react";
const Profile = () => {
return (
    <React.Fragment>
      <h2>John Doe</h2>
      <p>Software Developer</p>
    </React.Fragment>
  );
};
```

✓ **Now there's no unnecessary , keeping the DOM clean.**

Shorter Syntax (Shorthand <>...</>)

```
const Profile = () => {
  return (
    <>
      <h2>John Doe</h2>
      <p>Software Developer</p>
    </>
  );
};
```

✓ **Same effect, but cleaner syntax!**

2) Importance of Unique Keys in List Rendering

When rendering lists using .map(), **React requires a unique** key for each item.

Why Use Unique Keys?

✓ Helps React **efficiently update** the DOM.
✓ Avoids rendering issues and incorrect UI updates.
✓ Prevents **unnecessary re-renders** that impact performance.

Example Without Unique Keys (Bad Practice)

```
const fruits = ["Apple", "Banana", "Cherry"];

const FruitList = () => {
  return (
    <ul>
```

```
    {fruits.map((fruit) => (
      <li>{fruit}</li> // ❌ Missing key
    ))}
  </ul>
 );
};
```

Problem: React will show a warning:

"Each child in a list should have a unique 'key' prop."

Correct Example With Unique Keys

```
const fruits = ["Apple", "Banana", "Cherry"];

const FruitList = () => {
 return (
  <ul>
    {fruits.map((fruit, index) => (
      <li key={index}>{fruit}</li> // ✅ Unique key assigned
    ))}
  </ul>
 );
};
```

✓ Now, React can track each item properly!

Better Practice: Use Unique IDs Instead of Index

```
const fruits = [
 { id: 1, name: "Apple" },
 { id: 2, name: "Banana" },
 { id: 3, name: "Cherry" },
```

```
];

const FruitList = () => {
  return (
    <ul>
      {fruits.map((fruit) => (
        <li key={fruit.id}>{fruit.name}</li> // ✅ Using a unique ID
      ))}
    </ul>
  );
};
```

✓ Using unique IDs ensures stability, even if the list order changes.

Day -9 Tasks : -

1. **Introduction to React Fragment**: Write a short explanation of what React.Fragment is and why it is used.
2. **Basic Fragment Example**: Create a functional component that renders two paragraphs using React.Fragment.
3. **Shorthand Fragment Syntax**: Use the shorthand <>...</> instead of React.Fragment to render multiple elements.
4. **Fragment with List**: Render a list of items using React.Fragment without extra divs.
5. **Styling without Extra Divs**: Apply CSS to sibling elements without adding unnecessary parent divs using fragments.
6. **Keys in List Rendering**: Write a short note on the importance of keys in React lists.
7. **Basic List with Keys**: Render a list of fruits with unique keys.
8. **List without Keys**: Render a list without keys and observe the console warning.

9. **Dynamic List with Keys**: Render a list of user names from an array with keys generated from the index.
10. **Unique IDs as Keys**: Use unique id properties from an object array as keys when rendering a list.
11. **Fragment with Table Rows**: Render table rows using React.Fragment without unnecessary wrapper elements.
12. **Map List with Conditional Rendering**: Render a list of numbers, displaying only even numbers with keys.
13. **Nested List with Keys**: Render a nested list where each inner list has unique keys.

Mini Projects

Mini Project 1: Fruit List

Description:
Create a list of fruits using React.Fragment and assign unique keys to each fruit.

Mini Project 2: User Table

Description:
Render a table of users with name and age columns using React.Fragment and unique keys.

Mini Project 3: Task List with Conditional Rendering

Description:
Render a list of tasks and display only completed tasks using unique keys.

Mini Project 4: Dynamic Number List

Description:
Render a list of random numbers and display only numbers greater than 50, using keys for each list item.

Day 10:

useEffect Hook – Introduction

Definition of useEffect in React

useEffect is a React Hook that lets you perform side effects in function components. Side effects include tasks that happen outside the component's rendering, such as fetching data, subscribing to events, or modifying the DOM. useEffect runs after the component renders and can re-run when dependencies change.

Common Use Cases of useEffect:

✓ Fetching data from an API
✓ Updating the document title
✓ Setting up event listeners
✓ Running code only when a component mounts/unmounts

Basic Syntax of useEffect

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

useEffect(() => {
  // Code to run after the component renders
}, [dependencies]);
```

The callback function runs after the component renders.
The dependency array ([]) controls when useEffect runs.

2) Using useEffect for Side Effects Example

1: Changing the Document Title

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

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

  useEffect(() => {
    document.title = `Clicked ${count} times`;
  }, [count]); // Runs when 'count' changes

  return (
    <div>
      <h2>Button Clicked {count} times</h2>
      <button onClick={() => setCount(count + 1)}>Click Me</button>
    </div>
  );
};

export default TitleChanger;
```

How It Works:

✓ Every time count changes, document.title updates.

✓ The effect doesn't run on every render, only when count changes.

3) Fetching Data with useEffect

Example 2: Fetching API Data

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

```
const UsersList = () => {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then((response) => response.json())
      .then((data) => setUsers(data));
  }, []); // Empty array → Runs only on mount

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

export default UsersList;
```

How It Works:

✓ Runs only once when the component mounts (because [] is empty).
✓ Fetches user data and updates the users state.
✓ Efficient & avoids unnecessary API calls.

Why Use an Empty Dependency Array ([]) in useEffect**?**

In React's useEffect, the dependency array **([]) controls when the effect runs.**

Using an empty array ([]) means the effect runs only once when the component mounts and cleans up when it unmounts.

Example: Running Code Only Once (on Mount)

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

const Example = () => {
  useEffect(() => {
    console.log("✅ Runs only once when the component mounts");

    return () => {
      console.log("❌ Cleanup when the component unmounts");
    };
  }, []); // Empty array → Runs only on mount & unmount

  return <h1>Check the console!</h1>;
};
```

🔹 **Why the Empty Array?**

1) useEffect runs after the component renders.
2) Without dependencies ([]), it runs only on mount and cleans up on unmount. 3) If we omit the array, it runs on every render, which is inefficient.

What Happens with Different Dependency Arrays?

Dependency Array	Effect Runs When?	Common Use Case
useEffect(() => {...}, [])	Only on mount & unmount	API calls, event listeners, timers
useEffect(() => {...}, [count])	On mount & when count changes	Reacting to state/prop updates
useEffect(() => {...}) (no array)	On every render	Not recommended (can cause performance issues)

Mounting & Unmounting in useEffect

To handle setup (on mount) and cleanup (on unmount), use this pattern:

```
useEffect(() => {
  console.log("✅ Mounted");

  return () => {
    console.log("❌ Unmounted");
  };
}, []);
```

✓ Runs on mount (first part).

✓ Cleans up on unmount (inside return).

Cleanup in useEffect

If an effect sets up something (like an interval or event listener), clean it up when the component unmounts.

Example 3: Cleaning Up an Interval

```
import { useState, useEffect } from "react";
const Timer = () => {
```

```
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds((prev) => prev + 1);
    }, 1000);

    return () => clearInterval(interval); // Cleanup function
  }, []);

  return <h2>Timer: {seconds}s</h2>;
};

export default Timer;
```

✓ Without cleanup, the interval would keep running after the component unmounts, causing memory leaks!

Day -10 Tasks : -

1. **Introduction to useEffect**: Write a short explanation of what the useEffect hook is and why it is used.
2. **Basic useEffect Example**: Use useEffect to display "Component Mounted" in the console when the component mounts.
3. **Effect with Empty Dependency Array**: Use useEffect to log a message only once when the component mounts.
4. **Effect with Dependency Array**: Use useEffect to log a message whenever a counter state changes.
5. **Multiple useEffect Hooks**: Create two useEffect hooks, one for logging name changes and another for age changes.

6. **Cleanup Function**: Use useEffect with a cleanup function to log "Component Unmounted" when the component is removed.
7. **API Data Fetching Simulation**: Use useEffect to log "Fetching Data..." when the component mounts.
8. **Dynamic Title Update**: Use useEffect to update the page title with the current counter value.
9. **Form Input Logging**: Log the value of an input field whenever it changes using useEffect.
10. **Condition-Based Side Effects**: Display "Counter is Even" whenever the counter is an even number using useEffect.
11. **Fetching Data with Fake API**: Use useEffect to fetch user data from a placeholder API like https://jsonplaceholder.typicode.com/users.
12. **Delayed API Fetching**: Use setTimeout inside useEffect to simulate delayed API fetching.
13. **Button Click API Fetching**: Fetch data from an API only when a button is clicked.

Mini Projects

Mini Project 1: Counter with Title Update

Description:
Create a counter that updates the document title with the counter value whenever the value changes.

Mini Project 2: User List

Description:
Fetch a list of users from https://jsonplaceholder.typicode.com/users and display them in a list.

Mini Project 3: Show/Hide Timer

Description:
Create a timer that starts when the component mounts and stops when the component is removed using a cleanup function.

Mini Project 4: Search Filter

Description:
Fetch a list of users and allow users to filter the list using an input field. Update the filtered list whenever the input changes.

Day 11:

useState vs. useEffect

1) Understanding useState and useEffect Together useState

(Managing Component State)

Stores **local state** in a component.

Re-renders the component when the state updates.

Example: Counter Using useState

```
import { useState } from "react";
const Counter = () => {
  const [count, setCount] = useState(0);
return (
   <div>
```

```
<h2>Count: {count}</h2>
    <button onClick={() => setCount(count + 1)}>Increment</button>
  </div>
 );
};

export default Counter;
```

✓ Every time we click **"Increment"**, setCount updates the state, triggering a re-render.

useEffect (Handling Side Effects)

Runs after rendering to handle side effects.

Can update state using useState.

Example: Updating Document Title Dynamically

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

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

 useEffect(() => {
  document.title = `Clicked ${count} times`;
 }, [count]); // Runs whenever 'count' changes

 return (
  <div>
   <h2>Button Clicked {count} times</h2>
   <button onClick={() => setCount(count + 1)}>Click Me</button>
```

```
  </div>
 );
};

export default TitleChanger;
```

✓ Whenever count updates, useEffect runs and updates the document title.

2) Using useEffect to Update useState Dynamically Example:

Toggle Dark Mode

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

const ThemeToggler = () => {
 const [darkMode, setDarkMode] = useState(false);
 useEffect(() => {
  document.body.style.backgroundColor = darkMode ? "#333" : "#fff";
  document.body.style.color = darkMode ? "#fff" : "#000";
 }, [darkMode]); // Runs when darkMode changes
 return (
  <div>
   <h2>Dark Mode: {darkMode ? "Enabled" : "Disabled"}</h2>
   <button onClick={() => setDarkMode(!darkMode)}>Toggle Theme</button>
  </div>
 );
};
export default ThemeToggler;
```

✓ **When we** toggle **darkMode, useEffect updates the background dynamically. 3**

) Fetching Data and Updating State with useState & useEffect

Example: Fetch and Display Users

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

const UsersList = () => {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then((response) => response.json())
      .then((data) => {
        setUsers(data);
        setLoading(false);
      })
      .catch(() => setLoading(false));
  }, []); // Runs only on mount

  return (
    <div>
      <h2>User List</h2>
      {loading ? <p>Loading...</p> : (
        <ul>
          {users.map((user) => (
            <li key={user.id}>{user.name}</li>
          ))}
```

```
      </ul>
    )}
   </div>
  );
};

export default UsersList;
```

✓ **Uses useState for storing users & loading state.**

✓ **Uses useEffect to fetch data when the component mounts.**

Day -11 Tasks : -

1. **useState vs. useEffect Explanation**: Write a short explanation of the difference between useState and useEffect.
2. **Basic Counter with useState & useEffect**: Create a counter that logs the updated count whenever the button is clicked.
3. **Input Field with useState and useEffect**: Display the input field value in the console whenever the user types.
4. **useEffect with Empty Dependency Array**: Log "Component Mounted" only once when the component mounts.
5. **useEffect with Dependency Array**: Update the document title whenever the counter value changes.
6. **API Fetching with useEffect**: Fetch user data from https://jsonplaceholder.typicode.com/users and display it in the console.
7. **Data Fetch with useState and useEffect**: Store fetched data in state and display it in a list.
8. **Conditional Data Fetching**: Fetch data only when a button is clicked and store the data using useState.
9. **Timer with Cleanup Function**: Create a timer that updates the count every second and stops when the component is removed.

10. **Toggle Visibility with useState**: Toggle the visibility of a paragraph using a button and useState.
11. **Search Filter with useEffect**: Filter a list of users by name whenever the input field value changes.
12. **Fetch Data on Button Click**: Use useState and useEffect to fetch and display user data only when a button is clicked.
13. **Loading State with API Fetch**: Display "Loading..." while fetching data and show the data when it's loaded.

Mini Projects

Mini Project 1: Counter with Auto Update

Description:
Create a counter that automatically increases every second using useEffect and useState.

Mini Project 2: Search User List

Description:
Fetch user data and allow users to filter the list using an input field.

Mini Project 3: Toggle API Data

Description:
Fetch user data from an API when a button is clicked and hide the data when the button is clicked again.

Mini Project 4: Random Quote Generator

Description:
Fetch random quotes from an API every 5 seconds and display them.

Day 12:

useContext Hook

1) What is the Context API?

The Context API is used to share state between multiple components without prop drilling. Instead of passing props down manually, context allows any component to access shared data.

When to Use Context API?

Global state management (theme, user authentication, language settings).
Avoiding prop drilling (passing props through multiple levels).
Sharing state across deeply nested components.

Syntax : -

```
const value = useContext(SomeContext)
```

useContext is a React Hook that lets you read and subscribe to context from your component.

What is createContext?

createContext is used to create a new context object that allows components to share values.

Example: Creating a Context

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

// Create a Context for user data
const UserContext = createContext(null);

export default UserContext;
```

- createContext(null) creates a UserContext with a default value of null.
- Any component can now use this UserContext.

What is a Provider in React Context API?

Definition:

A Provider is a special component in React Context API that wraps child components and supplies a value to them. It is used to share state or data globally without prop drilling.

Syntax of Provider

```
<Context.Provider value={providedValue}>
 {children}
</Context.Provider>
```

- Context.Provider is used to pass down the providedValue to all child components.
- value={providedValue} is the data that will be accessible by components using useContext().
- {children} represents the components inside the provider, which can consume the context.

Example: Using Provider to Share Data

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

// 1 )  Create a Context
const UserContext = createContext();

// 2 ) Create a Provider Component
const UserProvider = ({ children }) => {
  const [user, setUser] = useState({ name: "John Doe", age: 25 });

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

// 3 ) Use the Provider in the App Component
const App = () => {
  return (
    <UserProvider>
      <UserProfile />
    </UserProvider>
  );
};

export default App;
```

🔹 UserProvider wraps child components and provides the user state.

🔹 value={{ user, setUser }} allows child components to read (user) and update (setUser) the state.

🔹 Any component inside <UserProvider> can access UserContext without props.

What is UseContext?

useContext(SomeContext)

Call useContext at the top level of your component to read and subscribe to Context

```
import { useContext } from 'react';

function MyComponent() {
  const theme = useContext(ThemeContext);
  // ...
```

Example: Consuming Context Data

Once we have a Provider, we can access its data using useContext().

```
import { useContext } from "react";
import UserContext from "./UserContext";
const UserProfile = () => {
  const { user } = useContext(UserContext); // Access context value
  return <h1>Welcome, {user.name}!</h1>;
};
export default UserProfile;
```

- useContext(UserContext) retrieves the user object.
- Now UserProfile can display the user's name without passing props manually.

Example – ThemeSwitcher (Project)

Step 1: Create a Context

First, create a **context file** to store shared state.

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

// Creating a Theme Context
const ThemeContext = createContext();

export default ThemeContext;
```

Step 2: Wrap Components with Context Provider

Use the **Provider** to wrap your components and **pass a value**.

```
import { useState } from "react";
import ThemeContext from "./ThemeContext";
import ThemeSwitcher from "./ThemeSwitcher";
const App = () => {
 const [theme, setTheme] = useState("light");

 return (
  <ThemeContext.Provider value={{ theme, setTheme }}>
   <div className={theme === "light" ? "light-mode" : "dark-mode"}>
    <h1>Theme: {theme}</h1>
    <ThemeSwitcher />
   </div>
  </ThemeContext.Provider>
```

```
  );
};

export default App;
```

✓ The Provider supplies the theme and setTheme state to all child components.

Step 3: Access Context with useContext

Instead of passing theme as a prop, consume it with useContext.

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

const ThemeSwitcher = () => {
  const { theme, setTheme } = useContext(ThemeContext);

  return (
    <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>
      Toggle Theme
    </button>
  );
};

export default ThemeSwitcher;
```

✓ Now, ThemeSwitcher can access and update the theme without prop drilling.

3) Understanding Provider & Consumer Pattern

The **Provider** gives the context value to all components inside it.

The **Consumer (useContext)** lets components access the context value.

Day – 12 Tasks:

1. **Introduction to Context API**: Write a short explanation of what the Context API is and why it is used in React.
2. **Create Context**: Create a simple context using React.createContext().
3. **Context Provider Example**: Wrap a parent component with the Context.Provider and pass a value.
4. **Access Context with useContext**: Use the useContext hook to access context values in a child component.
5. **Multiple Components with Context**: Access the same context value in two sibling components.
6. **Default Context Value**: Create a context with a default value and access it without a provider.
7. **Dynamic Context Value**: Pass a dynamic value (like a counter) to the context provider.
8. **Nested Context Access**: Access context values from nested child components.
9. **Update Context Value with useState**: Use useState to dynamically update the context value from a child component.
10. **Multiple Contexts**: Use two different contexts in the same component.
11. **Conditional Rendering with Context**: Display different content based on context values.
12. **Theme Toggle using Context**: Create a context that provides a theme (Light or Dark) and allows toggling.
13. **Context with API Data**: Fetch user data and share it across components using context.

Mini Projects

Mini Project 1: Theme Switcher

Description:
Create a context that provides a Light or Dark theme and allows users to toggle between them.

Mini Project 2: User Authentication

Description:
Create a context that stores whether the user is logged in or not and displays different content accordingly.

Mini Project 3: Counter with Context

Description:
Create a context that stores a counter value and allows multiple components to increment or display the value.

Mini Project 4: Language Selector

Description:
Create a context that provides a selected language (English or French) and displays text in the selected language.

Day 13:

Custom Hooks

1) What Are Custom Hooks?

🔹 A **Custom Hook** is a **reusable function** in React that uses hooks (useState, useEffect, etc.).

🔹 Helps **extract logic** from components to keep code **clean** and **organized**.

🔹 Follows the **naming convention**: Always start with "use", e.g., useFetch, useAuth.

Example: Why Use Custom Hooks?

Before Custom Hook: 👇 (Logic is duplicated in multiple components)

```
function Component1() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    console.log(`Count: ${count}`);
  }, [count]);
}
function Component2() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    console.log(`Count: ${count}`);
  }, [count]);
}
```

❌ Code repetition makes maintenance harder.

Solution: Create a Custom Hook!

```
function useCounter() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    console.log(`Count: ${count}`);
  }, [count]);
  return { count, setCount };
}
function Component1() {
  const { count, setCount } = useCounter();
}
function Component2() {
  const { count, setCount } = useCounter();
}
```

✅ **Reusability:** Now, both components share the same logic using useCounter().

2) Why Use Custom Hooks?

- Avoids Code Duplication → Write logic **once** and reuse it.
- Keeps Components Clean → Removes unnecessary logic from UI components.
- Encapsulates Side Effects → Manage useEffect logic **inside** a custom hook.
- Easier to Test → **Hooks** isolate logic, making them easier to test.

3) How to Create & Use a Custom Hook

- Steps to Create a Custom Hook:

1) Define a function that starts with use. 2
) Use built-in React hooks inside it.
3) Return data or functions for reuse.

Example: Creating a useFetch Hook to Fetch API Data

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

function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch(url)
      .then((res) => res.json())
      .then((data) => {
        setData(data);
        setLoading(false);
      })
      .catch((err) => {
        setError(err);
        setLoading(false);
      });
  }, [url]);
```

```
  return { data, loading, error };
}
export default useFetch;
```

Reusability: Use useFetch() in any component.

Using useFetch Hook in a Component

```
import useFetch from "./useFetch";

function UsersList() {
  const { data, loading, error } =
useFetch("https://jsonplaceholder.typicode.com/users");

  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error.message}</p>;

  return (
    <ul>
      {data.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}

export default UsersList;
```

✓ **Simplicity:** Fetch logic is now reusable across multiple components.

4) Common Custom Hooks Examples

Custom Hook	Purpose
useFetch	Fetches API data & handles loading/errors
useCounter	Manages a counter (increment/decrement/reset)
useLocalStorage	Saves & retrieves data from local storage
useAuth	Manages user authentication state
useDebounce	Delays function execution (useful for search inputs)

5) Summary Table

Feature	Definition
Custom Hook	A reusable function in React that **uses hooks**
Why Use It?	Avoids repetition, keeps components clean, and isolates logic
Naming Rule	Must start with "use", e.g., useTheme, useFetch
Example	useFetch for API requests, useCounter for managing counters

Day – 13 Tasks:

1) **Create a useCounter Hook**

- Implement increment, decrement, and reset functions.
- Example usage: const { count, increment, decrement } = useCounter();

2) **Create a useToggle Hook**

- A hook that toggles between true and false.
- Example: const [isOpen, toggle] = useToggle();

3) **Create a useLocalStorage Hook**

- Save and retrieve data from localStorage.
- Example: const [value, setValue] = useLocalStorage("theme", "dark");

4) **Create a useFetch Hook**

- Fetch API data and handle loading & errors.
- Example: const { data, loading, error } = useFetch(url);

5) **Create a useInput Hook**

- Manage form inputs with automatic state handling.
- Example: const { value, onChange } = useInput("");

6) **Create a useDebounce Hook**

- Delay function execution (useful for search inputs).
- Example: const debouncedValue = useDebounce(value, 500);

7)**Create a usePrevious Hook**

- Store and return the previous state value.
- Example: const prevValue = usePrevious(value);

8)**Create a useHover Hook**

- Detect if an element is being hovered over.
- Example: const [isHovered, ref] = useHover();

9) **Create a useTitle Hook**

- Dynamically update the document title.
- Example: useTitle("My Page Title");

10) **Create a useTimeout Hook**

- Run a function after a delay (like setTimeout).
- Example: useTimeout(() => alert("Hello!"), 3000);

11) **Create a useClipboard Hook**

- Copy text to the clipboard.
- Example: const { copy } = useClipboard();

12) **Create a useMediaQuery Hook**

- Detect screen size changes (e.g., mobile vs desktop).
- Example: const isMobile = useMediaQuery("(max-width: 768px)");

13) **Create a useOnlineStatus Hook**

- Detect if the user is online or offline.
- Example: const isOnline = useOnlineStatus();

14) **Create a useGeolocation Hook**

- Get the user's location using the Geolocation API.
- Example: const { latitude, longitude } = useGeolocation();

15)**Create a useTheme Hook**

- Manage light/dark mode toggling.
- Example: const { theme, toggleTheme } = useTheme();

Day 14:

Forms in React

1) Controlled vs. Uncontrolled Components

Controlled Components

The form input **state is controlled by React** using useState.

Every keystroke updates the state.

Example: Controlled Input Field

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

const ControlledForm = () => {
  const [name, setName] = useState("");

  const handleChange = (event) => {
    setName(event.target.value); // Updates state on every input change
  };

  return (
    <div>
      <input type="text" value={name} onChange={handleChange} />
      <p>Typed Name: {name}</p>
    </div>
  );
};

export default ControlledForm;
```

✓ **React fully controls the input value** via useState.

Uncontrolled Components

Uses **refs** instead of useState.

React **does not** control the input state directly.

Example: Uncontrolled Input Field

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

const UncontrolledForm = () => {
  const nameRef = useRef(); // Reference to the input field

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`Entered Name: ${nameRef.current.value}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={nameRef} />
      <button type="submit">Submit</button>
    </form>
  );
};

export default UncontrolledForm;
```

✓ Uses useRef() to directly access input value without storing it in React state.

2) Handling Multiple Inputs

🔷 Manage multiple form fields using a single state object.

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

const MultiFieldForm = () => {
  const [formData, setFormData] = useState({ name: "", email: "" });

  const handleChange = (event) => {
    setFormData({
      ...formData,
      [event.target.name]: event.target.value,
    });
  };

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`Name: ${formData.name}, Email: ${formData.email}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" name="name" value={formData.name}
onChange={handleChange} placeholder="Name" />
      <input type="email" name="email" value={formData.email}
onChange={handleChange} placeholder="Email" />
      <button type="submit">Submit</button>
    </form>
  );
};

export default MultiFieldForm;
```

✅ Efficiently handles multiple fields with one state object.

3) Handling Checkboxes, Radio Buttons, and Select Dropdowns Handling Checkboxes

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

const CheckboxForm = () => {
  const [isChecked, setIsChecked] = useState(false);

  return (
    <form>
      <label>
        <input type="checkbox" checked={isChecked} onChange={(e) =>
setIsChecked(e.target.checked)} />
        Accept Terms & Conditions
      </label>
      <p>{isChecked ? "✅ Accepted" : "❌ Not Accepted"}</p>
    </form>
  );
};

export default CheckboxForm;
```

Handling Radio Buttons

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

const RadioButtonForm = () => {
  const [gender, setGender] = useState("");

  return (
```

```
   <form>
     <label>
       <input type="radio" name="gender" value="Male" onChange={(e) =>
setGender(e.target.value)} />
       Male
     </label>
     <label>
       <input type="radio" name="gender" value="Female" onChange={(e) =>
setGender(e.target.value)} />
       Female
     </label>
     <p>Selected: {gender}</p>
   </form>
  );
};

export default RadioButtonForm;
```

Handling Select Dropdown

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

const SelectForm = () => {
  const [fruit, setFruit] = useState("apple");

  return (
    <form>
     <select value={fruit} onChange={(e) => setFruit(e.target.value)}>
       <option value="apple">Apple</option>
       <option value="banana">Banana</option>
       <option value="mango">Mango</option>
     </select>
```

```
    <p>Selected Fruit: {fruit}</p>
   </form>
  );
};

export default SelectForm;
```

4) Handling Form Submission with Validation

🔹 Basic validation before submission.

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

const ValidationForm = () => {
  const [email, setEmail] = useState("");
  const [error, setError] = useState("");

  const handleSubmit = (event) => {
    event.preventDefault();
    if (!email.includes("@")) {
      setError("Invalid email format");
      return;
    }
    alert(`Submitted Email: ${email}`);
    setError("");
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="email" value={email} onChange={(e) => setEmail(e.target.value)}
placeholder="Enter email" />
      {error && <p style={{ color: "red" }}>{error}</p>}
      <button type="submit">Submit</button>
```

```
  </form>
 );
};

export default ValidationForm;
```

✅ Ensures **valid input** before submission.

✅ Shows **error messages** if validation fails.

5) Form with useReducer for Complex Forms

🔹 useReducer manages form state in complex cases.

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

const formReducer = (state, action) => {
 return { ...state, [action.name]: action.value };
};

const ComplexForm = () => {
 const [formState, dispatch] = useReducer(formReducer, { username: "",
password: "" });
 return (
  <form>
   <input type="text" name="username" value={formState.username}
onChange={(e) => dispatch(e.target)} placeholder="Username" />
   <input type="password" name="password" value={formState.password}
onChange={(e) => dispatch(e.target)} placeholder="Password" />
   <button type="submit">Submit</button>
  </form>
 );
};

export default ComplexForm;
```

✅ useReducer simplifies managing **large form states**.

6) Using Custom Hook for Forms

🔷 Encapsulate form logic in a **custom hook**.

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

const useForm = (initialState) => {
  const [values, setValues] = useState(initialState);

  const handleChange = (event) => {
    setValues({ ...values, [event.target.name]: event.target.value });
  };

  return { values, handleChange };
};

// Usage Example
const CustomHookForm = () => {
  const { values, handleChange } = useForm({ name: "", email: "" });

  return (
    <form>
      <input type="text" name="name" value={values.name}
onChange={handleChange} placeholder="Name" />
      <input type="email" name="email" value={values.email}
onChange={handleChange} placeholder="Email" />
      <button type="submit">Submit</button>
    </form>
  );
};
export default CustomHookForm;
```

✓ **Reusable** across multiple forms.

Day – 14 Tasks:

1. **Introduction to Forms**: Write a short explanation of forms in React and the difference between controlled and uncontrolled components.
2. **Basic Controlled Input**: Create an input field that displays its value using useState.
3. **Multiple Inputs in Form**: Create a form with name and email inputs and display the values on submit.
4. **Textarea Input**: Create a controlled textarea that displays user input.
5. **Select Dropdown**: Create a controlled dropdown that displays the selected value.
6. **Checkbox Input**: Create a checkbox that toggles true or false in state.
7. **Radio Button Group**: Create a group of radio buttons that selects one option and displays the selected value.
8. **Form Submission**: Create a form that alerts the submitted values on submit.
9. **Prevent Default Behavior**: Use preventDefault() to stop the page from reloading on form submission.
10. **Validation Message**: Display an error message if the input is empty on form submission.
11. **Dynamic Form Fields**: Add an input field dynamically when the user clicks an "Add Field" button.
12. **Reset Form Fields**: Create a button that resets all input fields.
13. **Form with useEffect**: Automatically focus the first input field when the component mounts using useEffect.

Mini Projects

Mini Project 1: Login Form

Description:
Create a login form with email and password fields. Display validation messages if fields are empty.

Mini Project 2: Dynamic Todo List

Description:
Create a form that allows users to add and remove tasks dynamically.

Mini Project 3: Contact Form

Description:
Create a contact form with name, email, and message fields. Validate email format and required fields.

Mini Project 4: Subscription Form

Description:
Create a subscription form with name, email, and a checkbox for agreeing to terms. Disable the submit button if terms are not agreed upon.

Main Projects :

Project 1: Personal Blog Website

Description:
Create a simple blog website where users can:

- View a list of blog posts (Day 5: List Rendering)
- Add new blog posts through a form (Day 14: Forms in React)
- Show dynamic post titles on the page title (Day 10: useEffect)
- Toggle between Light and Dark themes (Day 12: useContext)

- Store posts in local storage (Day 13: Custom Hooks with useLocalStorage)

Project 2: Todo List Application

Description:
Develop a dynamic Todo List app where users can:

- Add, update, and delete tasks (Day 6: State Management)
- Mark tasks as completed with conditional rendering (Day 5: Conditional Rendering)
- Filter tasks based on their completion status (Day 5: List Rendering with Filter)
- Persist tasks using local storage (Day 13: useLocalStorage)
- Display a confirmation popup before deleting a task

Project 3: Weather App

Description:
Build a weather app that allows users to:

- Search for a city and fetch weather data (Day 10: useEffect with API Fetching)
- Display weather information like temperature and humidity (Day 6: State Management)
- Show a loading spinner while fetching data (Day 11: useState with useEffect)
- Toggle between Celsius and Fahrenheit (Day 12: useContext)
- Automatically detect the user's location and display weather info (Bonus: Geolocation API)

Project 4: User Management System

Description:
Develop a user management system where users can:

- Add new users through a form (Day 14: Forms)

- View user details in a list (Day 5: List Rendering)
- Edit user details using a modal popup (Day 6: State Management)
- Delete users with confirmation prompts (Day 7: Handling Events)
- Fetch user data from an API (Day 10: useEffect with API)

Project 5: Ecommerce Product List

Description:
Build a product listing page where users can:

- View a list of products (Day 5: List Rendering)
- Filter products by category or price (Day 11: useState with List Filtering)
- Add products to the cart (Day 6: State Management)
- Show cart items in a separate list (Day 9: React Fragment with List Rendering)
- Display total price dynamically (Day 6: State)

Day 16:

React Router Basics

1) Introduction to React Router

- React is a single-page application (SPA) framework.
  Without React Router, navigating between pages requires reloading the entire app.
- React Router enables navigation between different pages in a React app without a full page reload.
- It uses client-side routing, meaning it changes the URL and updates the UI without refreshing the page.

Installing React Router

Run this command to install React Router in your project:

```
npm install react-router-dom
```

2) BrowserRouter, Routes, and Route Explained

BrowserRouter (Main Router Wrapper)

🔹 Wraps the entire application to enable routing.

🔹 Keeps the UI in sync with the URL.

Example:

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

function App() {
  return (
    <BrowserRouter>
```

```
    <h1>Welcome to My App</h1>
  </BrowserRouter>
 );
}

export default App;
```

✓ BrowserRouter enables routing inside a React app.

✓ Must wrap all <Route> components.

Routes (Route Container)

🔹 Contains multiple <Route> components.

🔹 Ensures only one matching route is displayed at a time.

Example:

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

function App() {
 return (
  <BrowserRouter>
   <Routes>
    <Route path="/" element={<Home />} />
    <Route path="/about" element={<About />} />
   </Routes>
  </BrowserRouter>
 );
}
```

✓ Ensures only one matching route is rendered at a time.

Route (Defines Each Page Path)

🔹 Specifies which component should render based on the URL.

🔹 Uses path (URL) and element (component to render).

Example:

```
<Route path="/contact" element={<Contact />} />
```

Breakdown:

/contact → The URL path.

<Contact /> → The component to render when the user visits /contact.

✅ Multiple routes can be defined inside <Routes>.

✅ Only one matching <Route> will be shown.

3) Setting Up Routes in React

Basic Route Setup

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Home from "./Home";
import About from "./About";

const App = () => {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
};

export default App;
```

✓ BrowserRouter wraps the entire app to enable routing.

✓ Routes contains all the defined routes.

✓ Route defines individual **path → component** mappings.

3) Navigating Between Pages Using <Link> for Navigation

Instead of using <a href="..." /> (which reloads the page), use <Link> for **client-side navigation**:

```
import { Link } from "react-router-dom";

const Navbar = () => {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
    </nav>
  );
};

export default Navbar;
```

✓ to="/" navigates to the home page.

✓ to="/about" navigates to the About page.

4) Nested Routes (Subpages)

Sometimes, you need **nested pages** (e.g., /profile/settings).

Example: Parent and Child Routes

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Profile from "./Profile";
import Settings from "./Settings";
const App = () => {
```

```
 return (
  <BrowserRouter>
   <Routes>
    <Route path="/profile" element={<Profile />}>
     <Route path="settings" element={<Settings />} />
    </Route>
   </Routes>
  </BrowserRouter>
 );
};

export default App;
```

✓ The **Settings** page is accessible at /profile/settings.

5) Redirecting Users

Use **Navigate** to redirect users to another page:

```
import { Navigate } from "react-router-dom";

const NotFound = () => {
 return <Navigate to="/" />; // Redirect to Home Page
};
```

Day 16 Tasks:

1. **Introduction to React Router**: Write a short explanation of what React Router is and why it's used.
2. **Install React Router**: Set up React Router in a project using npm install react-router-dom.
3. **BrowserRouter Setup**: Wrap the entire application with <BrowserRouter> in the root component.
4. **Basic Route Setup**: Create two components (Home and About) and display them on different routes.

5. **Navigation with Link**: Use the <Link> component to navigate between Home and About pages.
6. **Exact Path Usage**: Use the exact attribute to ensure the correct route is matched.
7. **Default Route (Redirect to Home)**: Set up a default route that redirects to the Home page.
8. **404 Page Setup**: Create a NotFound component and display it when no route is matched.
9. **Active Link Styling**: Apply active styling to the currently selected link using NavLink.
10. **Nested Routes Setup**: Create nested routes under the About page.
11. **Navigate Programmatically**: Use the useNavigate hook to navigate on button click.
12. **Navigation Bar Component**: Create a reusable Navbar component with links to different pages.
13. **Protected Page Example (Static Check)**: Display a message like "Access Denied" if the user is not logged in (without authentication).

Mini Projects

Mini Project 1: Simple Portfolio Website

Description:
Create a portfolio website with Home, About, and Contact pages using React Router.

Mini Project 2: Navigation App

Description:
Build an app with three pages (Home, Services, and Contact) and a navigation bar with active link styling.

Mini Project 3: Static Blog

Description:
Create a static blog website with Home, Posts, and About pages and navigate between them.

Mini Project 4: Product Showcase

Description:
Create a product showcase app with Home, Products, and Contact pages, along with navigation links.

Day 17:

Dynamic Routing & Route Parameters

1)What Are Dynamic Routes?

- A **dynamic route** contains a placeholder (parameter) in the URL.
- The **actual value** changes based on user interaction.
- Helps in fetching **specific data** (e.g., /users/5, /products/laptop).

Example:

/user/1 → Displays **User 1's** profile.

/user/2 → Displays **User 2's** profile.

2) How to Define a Dynamic Route in React Router

- Use : to define a **route parameter** (e.g., :id).
- Extract parameters using useParams().

Example: Dynamic User Profile Page

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

// Dynamic User Component
const UserProfile = () => {
  const { id } = useParams(); // Get dynamic parameter from URL
  return <h2>👤 User Profile: {id}</h2>;
};

// App Component
function App() {
return (
   <BrowserRouter>
     <nav>
       <Link to="/user/1">User 1</Link> |
       <Link to="/user/2">User 2</Link>
     </nav>

     <Routes>
       <Route path="/user/:id" element={<UserProfile />} />
     </Routes>
   </BrowserRouter>
  );
}

export default App;
```

✅ Clicking **User 1** → Goes to /user/1, displaying "User Profile: 1".

✅ Clicking **User 2** → Goes to /user/2, displaying "User Profile: 2".

3) Accessing Route Parameters with useParams()

🔹 useParams() is a React Router hook that **retrieves route parameters**.

🔹 Returns an **object** with key-value pairs.

Example:

If URL is /product/laptop:

```
const { id } = useParams();
console.log(id); // Output: "laptop"
```

✅ Helps fetch **data based on dynamic values**.

4) Using Route Parameters for API Calls

🔹 Fetch **specific data** based on the route parameter.

Example: Fetch User Data Dynamically

```
import { useEffect, useState } from "react";
import { useParams } from "react-router-dom";

const UserProfile = () => {
  const { id } = useParams();
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetch(`https://jsonplaceholder.typicode.com/users/${id}`)
      .then((res) => res.json())
      .then((data) => setUser(data))
      .catch((error) => console.error("Error:", error));
  }, [id]);

  return (
    <div>
      {user ? <h2>👤 {user.name}</h2> : <p>Loading...</p>}
    </div>
  );
```

```
};

export default UserProfile;
```

✅ When visiting /user/3, it fetches **User 3's** details.

5) Multiple Route Parameters

🔹 Supports **multiple parameters** (e.g., /category/:categoryId/product/:productId).

Example: Show a Specific Product in a Category

```
const ProductDetails = () => {
  const { categoryId, productId } = useParams();
  return <h2>🎁 Category: {categoryId}, Product: {productId}</h2>;
};
```

```
<Route path="/category/:categoryId/product/:productId"
element={<ProductDetails />} />
```

✅ Visiting /category/electronics/product/5 → Displays Category: electronics, Product: 5.

Day -17 Tasks :

1. **Introduction to Dynamic Routing**: Write a short explanation of dynamic routes and their use cases in React.
2. **Basic Dynamic Route Setup**: Create a dynamic route like /user/:id using React Router.
3. **Access Route Parameters**: Use useParams() to access the route parameter inside a component.
4. **Display Route Parameter**: Show the route parameter value on the page.

5. **Multiple Route Parameters**: Create a route like /user/:id/:name and display both parameters.
6. **Optional Parameters**: Set up a route with an optional parameter like /user/:id?.
7. **Route Navigation with Parameters**: Use the Link component to navigate to routes with parameters.
8. **Programmatic Navigation with Parameters**: Use useNavigate() to navigate to dynamic routes on button click.
9. **Default Parameter Value**: Provide a default value if the parameter is missing.
10. **URL Parameter Validation**: Display an error message if the parameter is not a valid number or string.
11. **Nested Route with Parameters**: Create nested dynamic routes like /user/:id/posts.
12. **Filter Data Based on Route Parameter**: Fetch user details based on the id parameter.
13. **Cleanup with useEffect**: Clean up fetched data when the route parameter changes.

Mini Projects

Mini Project 1: User Profile

Description:
Create a dynamic user profile page where users can navigate to /user/:id and view user details fetched from an API.

Mini Project 2: Product Details Page

Description:
Build an app with a product list where clicking on a product navigates to /product/:id and displays the product's details.

Mini Project 3: Blog App

Description:
Create a blog app where clicking on a post title navigates to /post/:id and displays the full post content.

Mini Project 4: Movie Details App

Description:
Fetch and display a list of movies. Clicking on a movie navigates to /movie/:id and shows the movie's details.

Day 18 :

React Axios

Introduction to Axios in React

Axios is a popular HTTP client library used to make API requests in React applications. It provides an easy way to send HTTP requests, handle responses, and manage errors.

Why Axios?

Promise-based API
Easy to use
Automatic JSON data transformation
Error handling
Request & Response Interceptors

1. Installing Axios

To use Axios in your React project, install it first:

Install Axios

```
npm install axios
```

2. Making API Requests with Axios

Syntax

```
axios.get(url, config)
axios.post(url, data, config)
axios.put(url, data, config)
axios.delete(url, config)
```

Example - GET Request

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

const UserList = () => {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    axios.get("https://jsonplaceholder.typicode.com/users")
      .then(response => {
        setUsers(response.data);
      })
      .catch(error => {
        console.error("Error fetching users:", error);
      });
  }, []);
  return (
    <div>
      {users.map(user => (
        <p key={user.id}>{user.name}</p>
      ))}
    </div>
  );
```

```
};
export default UserList;
```

3. Handling Responses and Errors

Axios Response Structure

```
response.data     // Data from API
response.status   // Status code
response.headers  // Response headers
response.config   // Request configuration
```

Error Handling

```
axios.get(url)
  .then(response => console.log(response.data))
  .catch(error => {
    if (error.response) {
      console.error("Server responded with:", error.response.status);
    } else if (error.request) {
      console.error("No response received:", error.request);
    } else {
      console.error("Error:", error.message);
    }
  });
```

4. Configuring Axios Requests

Adding Headers

```
axios.get(url, {
  headers: {
    Authorization: "Bearer token123",
    "Content-Type": "application/json",
  },
});
```

Setting Timeout

```
axios.get(url, { timeout: 5000 })
  .then(response => console.log(response))
  .catch(error => console.error("Timeout Error:", error));
```

5. POST Request Example

```
const createUser = async () => {
  try {
    const response = await
axios.post("https://jsonplaceholder.typicode.com/users", {
      name: "John Doe",
      email: "john@gmail.com",
    });
    console.log("User Created:", response.data);
  } catch (error) {
    console.error("Error Creating User:", error);
  }
};
```

6. PUT Request Example (Update Data)

```
const updateUser = async (id) => {
  try {
    const response = await
axios.put(`https://jsonplaceholder.typicode.com/users/${id}`, {
      name: "John Updated",
    });
    console.log("User Updated:", response.data);
  } catch (error) {
    console.error("Error Updating User:", error);
  }
};
```

7. DELETE Request Example

```
const deleteUser = async (id) => {
  try {
    const response = await
axios.delete(`https://jsonplaceholder.typicode.com/users/${id}`);
    console.log("User Deleted");
  } catch (error) {
    console.error("Error Deleting User:", error);
  }
};
```

8. Advanced Axios Usage

Axios Interceptors

Interceptors allow you to modify requests or responses before they are handled.

Request Interceptor

```
axios.interceptors.request.use(request => {
  request.headers.Authorization = "Bearer token123";
  console.log("Request Sent");
  return request;
});
```

Response Interceptor

```
axios.interceptors.response.use(
  response => {
  console.log("Response Received");
  return response;
  },
  error => {
    console.error("Error in Response:", error);
    return Promise.reject(error);
  }
);
```

9. Global Axios Configuration

Set Base URL globally:

```
axios.defaults.baseURL = "https://jsonplaceholder.typicode.com";
axios.defaults.headers.common["Authorization"] = "Bearer token123";
```

Day -18 Tasks :

1. **Introduction to Axios**: Write a short explanation of what Axios is and its advantages over fetch().
2. **Install Axios**: Install Axios using npm install axios.
3. **Basic GET Request**: Fetch dummy user data from an API (e.g., JSONPlaceholder) and display it.
4. **Handle API Errors**: Show an error message if the API request fails.
5. **Loading State**: Display a loading message while fetching data.
6. **POST Request**: Create a form and send user data to an API using Axios.
7. **PUT Request**: Update user data by making a PUT request.
8. **DELETE Request**: Delete user data by making a DELETE request.
9. **Global Axios Configuration**: Set up a base URL and common headers globally using Axios defaults.
10. **Axios Interceptors**: Use interceptors to log requests and responses.
11. **Cancel API Requests**: Cancel ongoing API requests using Axios Cancel Tokens.
12. **Retry Failed Requests**: Automatically retry failed requests using Axios interceptors.
13. **Async/Await with Axios**: Use async/await to make API requests cleaner.

Mini Projects

Mini Project 1: User List

Description:

Fetch and display a list of users from an API. Show loading and error messages.

Mini Project 2: Product CRUD App

Description:
Create an app that allows users to **Create**, **Read**, **Update**, and **Delete** products using Axios.

Mini Project 3: Weather App

Description:
Fetch weather data from an API based on user input and display the weather information.

Mini Project 4: Image Gallery

Description:
Fetch a list of images from an API and display them in a gallery format with loading and error states.

Day 19:

Higher-Order Components (HOCs):

What Are Higher-Order Components (HOCs)?

A Higher-Order Component (HOC) is a function that takes a component as input and returns a new component with additional functionality.

🔷 **Definition:**

A Higher-Order Component is a pattern in React that allows component reusability and logic sharing by wrapping a component with another component.

🔹 It allows reusing component logic (like authentication, logging, or API calls).
🔹 HOCs follow the pattern of functional programming where functions modify other functions.

🔹 **Syntax of an HOC:**

```
const withExtraFeature = (WrappedComponent) => {
  return (props) => {
    return <WrappedComponent {...props} extraProp="Extra Value" />;
  };
};
```

✓ The **HOC function** takes a WrappedComponent as an argument.
✓ It **returns a new component** with additional props or behavior.

Why Use Higher-Order Components?

🔹 Reusability**:** Share logic between multiple components.
🔹 Code Separation**:** Keep components clean and focused on UI.
🔹 Conditional Rendering: Modify the UI based on conditions.
🔹 Enhancing Components: Add extra props or behavior dynamically.

3) Basic Syntax of an HOC

A Higher-Order Component is a **function** that takes a component and returns a new component.

```
const withExtraFeature = (WrappedComponent) => {
  return (props) => {
    return <WrappedComponent {...props} extra="I am an extra prop!" />;
  };
};
```

🔹 WrappedComponent → The original component being **wrapped**.
🔹 withExtraFeature → The HOC that **enhances** the component.
🔹 {...props} → Ensures the original component gets all its props.

Example 1: Creating an HOC for Authentication

Let's create an HOC that **checks if a user is logged in** before rendering a component.

Step 1: Create the HOC

```
const withAuth = (WrappedComponent) => {
  return (props) => {
    const isLoggedIn = true; // Simulating authentication

    if (!isLoggedIn) {
      return <h2>Please log in to access this page.</h2>;
    }

    return <WrappedComponent {...props} />;
  };
};
```

Step 2: Use the HOC with a Component

```
const Dashboard = () => {
  return <h2>Welcome to the Dashboard!</h2>;
};

const ProtectedDashboard = withAuth(Dashboard);
```

✓ Now, ProtectedDashboard will only show the dashboard if the user is logged in.

Example 2: Adding Extra Props Using HOCs

```
const withExtraInfo = (WrappedComponent) => {
  return (props) => <WrappedComponent {...props} extra="Extra Info" />;
};

const MyComponent = (props) => {
  return <h2>{props.extra}</h2>;
};

const EnhancedComponent = withExtraInfo(MyComponent);
```

✓ EnhancedComponent now receives an extra prop (**extra="Extra Info"**).

Key Takeaways

✓ HOCs wrap components to add functionality without modifying them directly.
✓ They are pure functions**,** meaning they don't alter the original component.
✓ Useful for authentication, logging, permissions, and UI enhancements.

Day – 19 Tasks:

1. **Introduction to HOCs**: Write a brief explanation of what Higher-Order Components are and their benefits.
2. **Basic HOC Structure**: Create a simple HOC that logs "Component Rendered" whenever a wrapped component renders.
3. **Props Forwarding in HOCs**: Pass props from HOC to the wrapped component and display them.
4. **Style Wrapper HOC**: Create an HOC that adds custom CSS styles to the wrapped component.
5. **Conditional Rendering HOC**: Implement an HOC that conditionally renders a component based on a boolean prop.
6. **Authentication HOC**: Create an HOC that renders a component only if the isAuthenticated prop is true.

7. **Loading Spinner HOC**: Show a loading spinner until the data is loaded.
8. **Logging HOC**: Log component lifecycle events (mounted/unmounted) using the HOC.
9. **Data Fetching HOC**: Fetch data from an API inside an HOC and pass it to the wrapped component.
10. **Mouse Hover Tracker HOC**: Track mouse hover events and display the coordinates.
11. **Error Boundary HOC**: Catch and display errors from wrapped components.
12. **Title Update HOC**: Update the document title with the component name whenever it renders.
13. **Reuse Multiple HOCs**: Combine two or more HOCs into one component.

Mini Projects

Mini Project 1: Authentication Protected Page

Description:
Create an HOC that checks if the user is authenticated before displaying the component, otherwise show an "Access Denied" message.

Mini Project 2: Loading and Error Handler

Description:
Create an HOC that displays a loading spinner while fetching API data and handles errors if the API request fails.

Mini Project 3: Click Tracker

Description:
Create an HOC that tracks and displays the number of times a button is clicked.

Mini Project 4: Themed Components

Description:
Create an HOC that wraps components with different themes (light or dark mode).

Day 20:

React Portals

A React Portal is a way to render a component outside the main DOM hierarchy while still keeping it in the React component tree.

Definition:

React Portals allow us to render a child component inside a different DOM node that exists outside the root div (#root).

Why Use Portals?

Helps with modals, tooltips, dropdowns, and popups that need to break out of their parent's styles.

Prevents CSS conflicts when dealing with z-index or overflow issues.

Keeps event bubbling inside the React component tree.

How to Create a React Portal

Step 1: Create a Target DOM Node

In your index.html file, **add a new div** outside the #root div:

```
<div id="modal-root"></div>
```

Step 2: Use ReactDOM.createPortal()

The createPortal() method renders components in a different part of the DOM**.**

It takes two arguments**:**

- The React component to render.
- The DOM element where it should be mounted.

Example: Creating a Modal with Portals

```
import React from "react";
import ReactDOM from "react-dom";

const Modal = ({ message, onClose }) => {
  return ReactDOM.createPortal(
    <div style={modalStyle}>
      <p>{message}</p>
      <button onClick={onClose}>Close</button>
    </div>,
    document.getElementById("modal-root")
  );
};

const modalStyle = {
  position: "fixed",
  top: "50%",
  left: "50%",
  transform: "translate(-50%, -50%)",
  background: "white",
  padding: "20px",
  boxShadow: "0px 0px 10px rgba(0,0,0,0.3)",
  zIndex: 1000
};

export default Modal;
```

✓ The modal **renders inside #modal-root**, but it remains inside the **React tree**.

Step 3: Use the Modal in a Parent Component

```
import React, { useState } from "react";
import Modal from "./Modal";

const App = () => {
  const [showModal, setShowModal] = useState(false);

  return (
    <div>
      <h1>Main App</h1>
      <button onClick={() => setShowModal(true)}>Open Modal</button>
      {showModal && <Modal message="Hello from Portal!" onClose={() =>
setShowModal(false)} />}
    </div>
  );
};

export default App;
```

✓ Clicking **"Open Modal"** will **display the modal using a portal**.

✓ Clicking **"Close"** will **hide the modal**.

Day – 20 Tasks

1. **Introduction to Portals**: Write a short explanation of what React Portals are and why they are used.
2. **Basic Portal Setup**: Create a simple portal that renders a div element outside the main DOM tree.
3. **Create Portal Modal**: Implement a modal dialog using React Portals.

4. **CSS Styling with Portals**: Apply custom styles to portal elements without affecting the parent component.
5. **Event Bubbling**: Demonstrate event bubbling in portals by clicking on both the portal component and its parent.
6. **Portal with Click Outside Detection**: Close the portal when clicking outside the modal.
7. **Nested Portals**: Create a portal within another portal component.
8. **Form inside Portal**: Render a form component inside a portal and handle form submissions.
9. **Multiple Portals**: Render two different portal components on the same page.
10. **Portal Tooltip**: Create a tooltip that appears on hover using React Portals.
11. **Animation with Portals**: Add CSS animations to portal components when they mount or unmount.
12. **Accessing DOM Nodes in Portals**: Use useRef to access DOM nodes inside portal components.
13. **Reusable Portal Component**: Create a higher-order component (HOC) that wraps components with portal functionality.

Mini Projects

Mini Project 1: Modal Popup

Description:
Create a modal popup that opens on button click and closes when clicking outside or on the close button.

Mini Project 2: Tooltip Component

Description:
Build a tooltip that displays extra information when hovering over text.

Mini Project 3: Confirmation Dialog

Description:
Implement a confirmation dialog that asks the user to confirm an action before proceeding.

Mini Project 4: Image Lightbox

Description:
Create an image gallery where clicking on an image opens it in a fullscreen lightbox using React Portals.

Day 21:

State Management with Redux Toolkit (Introduction)

What is Redux?

Redux is a state management library for JavaScript applications, commonly used with React. It helps manage global state in a predictable way, making it easier to handle complex application data.

Definition:

Redux is a predictable state container for JavaScript apps. It centralizes application state and enforces strict unidirectional data flow.

Why Use Redux?

- Centralized State Management – Keeps the entire state in one place.
- Predictable State Updates – Uses pure functions (reducers) to update the state.

- Easier Debugging – Redux DevTools allow time-travel debugging.
- Better State Sharing – Easily share state across multiple components.

1) What is Redux Toolkit?

Redux Toolkit (RTK) is the official, recommended way to use Redux for managing global state in JavaScript applications — especially React.

It is a library that provides:

- Pre-built utilities (createSlice, createAsyncThunk, etc.)
- Best practices by default
- Reduced boilerplate code
- Better structure for writing Redux logic
- Redux Toolkit is part of the @reduxjs/toolkit package.

Why Use Redux Toolkit in React?

Here are the main reasons:

◆ **1. Less Boilerplate**

- Old Redux: Write actions, action creators, and reducers separately.
- RTK: Use createSlice() to define state, actions, and reducers all in one place.

📌 ***Old Redux:***

```
js
const INCREMENT = 'INCREMENT';
const increment = () => ({ type: INCREMENT });
const reducer = (state = 0, action) => {
  switch (action.type) {
    case INCREMENT: return state + 1;
    default: return state;
};
```

📌 **Redux Toolkit:**

```
js
const counterSlice = createSlice({
  name: 'counter',
  initialState: 0,
  reducers: {
    increment: (state) => state + 1,
  }
});
```

◆ **2. Includes Redux Thunk by Default**

RTK includes middleware like redux-thunk for handling async actions (e.g., API calls) out of the box.

```
js
export const fetchUsers = createAsyncThunk('users/fetch', async () => {
  const res = await fetch('/api/users');
  return res.json();
});
```

◆ **3. Improved DevTools Integration**

Easily track Redux actions, state changes, and API status via Redux DevTools.

◆ **4. Simpler Store Setup**

RTK simplifies creating and configuring the Redux store using `configureStore()`.

```
js
const store = configureStore({
  reducer: rootReducer,
});
```

◆ **5. Built-in Best Practices**

RTK promotes:

- Immutability with Immer
- Normalized state using createEntityAdapter
- Performance with createSelector

✅ **When to Use Redux Toolkit in React?**

Use it when:

- You have shared/global state (e.g., user auth, shopping cart)
- Multiple components need access to the same data
- You want predictable state updates
- Your app has complex state interactions (API calls, caching, etc.)

2. Project Setup with React + Redux Toolkit

```
npx create-react-app redux-counter-app
cd redux-counter-app
npm install @reduxjs/toolkit react-redux
```

3. Create Redux Store using configureStore()

🔧 File: app/store.js

```
js
// src/app/store.js
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from '../features/counter/counterSlice';

const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
});

export default store;
```

configureStore() automatically sets up Redux DevTools and applies the right middleware.

✅ 4. Create Slice using createSlice()

📄 File: features/counter/counterSlice.js

```
js
// src/features/counter/counterSlice.js
import { createSlice } from '@reduxjs/toolkit';

const initialState = {
  value: 0,
};

const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    increment: state => {
      state.value += 1;
    },
    decrement: state => {
      state.value -= 1;
    },
  },
});

// Export actions
export const { increment, decrement } = counterSlice.actions;

// Export reducer
export default counterSlice.reducer;
```

🔍 createSlice:

- Automatically generates action creators (increment, decrement)

- Uses Immer under the hood, so you can mutate state directly

✅ 5. Access State using useSelector()

👇 Example inside a React component:

```
js
import { useSelector } from 'react-redux';

const count = useSelector((state) => state.counter.value);
```

 useSelector() reads values from the Redux store.

✅ 6. Dispatch Actions using useDispatch()

👇 Example:

```
js
import { useDispatch } from 'react-redux';
import { increment, decrement } from '../features/counter/counterSlice';

const dispatch = useDispatch();

<button onClick={() => dispatch(increment())}>+</button>
<button onClick={() => dispatch(decrement())}>-</button>
```

 useDispatch() sends actions to the Redux store to update the state.

✅ 7. Create Counter Component

📁 File: components/Counter.js

```
js
// src/components/Counter.js
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from '../features/counter/counterSlice';

function Counter() {
  const count = useSelector((state) => state.counter.value);
  const dispatch = useDispatch();
```

```
  return (
    <div style={{ textAlign: 'center', marginTop: '50px' }}>
      <h1>Counter: {count}</h1>
      <button onClick={() => dispatch(increment())}>+</button>
      <button onClick={() => dispatch(decrement())}>-</button>
    </div>
  );
}

export default Counter;
```

✅ 8. Wrap App with Redux Provider

📄 File: `index.js`

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

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

✅ Final App Component

📄 File: App.js

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

```
import Counter from './components/Counter';

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

export default App;
```

Day 21 Concepts Summary:

Concept	What You Did
configureStore()	Created Redux store
createSlice()	Defined state + reducers + actions
useSelector()	Accessed Redux state in components
useDispatch()	Triggered actions to update state
Provider	Connected React to Redux store

Day – 21 Tasks

◆ **Task 1: Basic Counter**

Create a counter with +, -, and display the current count using Redux Toolkit.

◆ **Task 2: Reset Button**

Add a reset button that sets the counter back to zero.

◆ **Task 3: Increment by Amount**

Add a text input and a button to increment the counter by the entered number.

◆ **Task 4: Conditional Increment**
Add a button to increment only if the current count is odd.

◆ **Task 5: Disable Buttons**
Disable + button if count ≥ 10 and - button if count ≤ 0 using conditional rendering.

◆ **Task 6: Multiple Counters**
Create two separate counters using two different slices.

◆ **Task 7: Show Redux DevTools**
Inspect actions and state changes using Chrome Redux DevTools.

◆ **Task 8: Create a Theme Slice**
Make a Redux slice to toggle between light and dark themes in your app.

◆ **Task 9: Store Username**
Use Redux to store and update a username entered through a form input.

◆ **Task 10: Counter History**
Store the previous 5 values of the counter in an array using Redux.

◆ **Task 11: Separate Slice File Structure**
Move actions, reducers, and selectors into a cleaner structure (folder-based modules).

◆ **Task 12: Combine Reducers Manually**
Manually create a store that combines `counter` and `theme` slices.

◆ **Task 13: Unit Test a Slice**
Write basic unit tests for a slice (e.g., test if increment increases count).

Mini Projects :

1. Counter App

- Increment and decrement a number
- Use createSlice, configureStore
- Show current count on screen

💡 **2. Toggle Theme App**

- Toggle between light and dark modes
- Store theme state in Redux
- Update CSS or class based on theme

💡 **3. Simple Todo App**

- Add and remove todos
- Store todo list in Redux state
- Basic slice: addTodo, removeTodo

💡 **4. Login Status Toggle**

- Toggle login/logout status with button
- Store user status in Redux slice
- Show "Welcome, User" or "Please log in"

Day 22:

Day 22: Async Logic and Redux Middleware

✅ **1. Async Thunks with createAsyncThunk()**

createAsyncThunk lets you handle asynchronous logic like API calls.

📄 **Create Thunk: fetchUsers**

```
js
// src/features/users/userSlice.js
```

```
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';
import axios from 'axios';

export const fetchUsers = createAsyncThunk('users/fetchUsers', async () => {
  const response = await axios.get('https://jsonplaceholder.typicode.com/users');
  return response.data;
});
```

 Redux Toolkit automatically creates 3 action types for this thunk:

- users/fetchUsers/pending
- users/fetchUsers/fulfilled
- users/fetchUsers/rejected

✅ 2. Handling Loading, Success, and Error States

Define initialState and handle the above 3 actions inside your slice.

```
Js
const initialState = {
  users: [],
  loading: false,
  error: null,
};

const userSlice = createSlice({
  name: 'users',
  initialState,
  reducers: {},
  extraReducers: (builder) => {
    builder
      .addCase(fetchUsers.pending, (state) => {
        state.loading = true;
        state.error = null;
      })
      .addCase(fetchUsers.fulfilled, (state, action) => {
        state.loading = false;
        state.users = action.payload;
```

```
    })
    .addCase(fetchUsers.rejected, (state, action) => {
      state.loading = false;
      state.error = 'Failed to fetch users';
    });
  },
});

export default userSlice.reducer;
```

✅ 3. Middleware Basics

🔍 Redux Toolkit includes:

- redux-thunk middleware by default to support async logic.
- You do not need to install or configure it manually.

✅ 4. API Calls with Redux Toolkit in Component

📄 File: components/UserList.js

```
js
import React, { useEffect } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { fetchUsers } from '../features/users/userSlice';

function UserList() {
  const dispatch = useDispatch();
  const { users, loading, error } = useSelector((state) => state.users);

  useEffect(() => {
    dispatch(fetchUsers());
  }, [dispatch]);

  if (loading) return <p>Loading users...</p>;
  if (error) return <p>{error}</p>;

  return (
```

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

export default UserList;
```

✅ 5. Redux DevTools Setup

Redux Toolkit automatically enables DevTools in development mode. Just:

- Open Chrome → DevTools
- Go to Redux tab

Observe:

- Actions: pending, fulfilled, rejected
- State changes live

✅ Full Folder Setup Summary:

```
src/
|
├── app/
|   └── store.js
├── components/
|   └── UserList.js
├── features/
|   └── users/
|       └── userSlice.js
└── App.js
```

✅ Final App Boilerplate

File: App.js

```
js
import React from 'react';
import UserList from './components/UserList';

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

export default App;
```

File: store.js

```
js
import { configureStore } from '@reduxjs/toolkit';
import userReducer from '../features/users/userSlice';

const store = configureStore({
  reducer: {
    users: userReducer,
  },
});

export default store;
```

File: index.js

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

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

Concepts Learned Recap:

Concept	Description
createAsyncThunk	Handles async API calls with auto pending, fulfilled, rejected
extraReducers	Handles async action states inside the slice
redux-thunk	Middleware included by default in Redux Toolkit
API Integration	Clean separation of logic: async call, state updates, UI display
DevTools	See async actions and debug state

Day – 22 Tasks

◆ **Task 1:**

Setup a Redux slice with initial states:

```
js
loading: false, data: [], error: null
```

◆ **Task 2:**

Use createAsyncThunk to fetch a list of users from:
https://jsonplaceholder.typicode.com/users

◆ **Task 3:**

Handle all three thunk lifecycle states in extraReducers:

- pending → set loading: true
- fulfilled → store data
- rejected → store error

◆ **Task 4:**

Display fetched data in a component using useSelector().

◆ **Task 5:**
Show loading message or spinner when API is being called.

◆ **Task 6:**
Show error message when request fails (simulate using a wrong URL).

◆ **Task 7:**
Add a "Reload" button that refetches the data on click (use dispatch(thunk())).

◆ **Task 8:**
Build a GitHub user fetcher:

- Input: username
- Output: avatar, name, repo count

◆ **Task 9:**
Validate empty input and show an error toast or message.

◆ **Task 10:**
Build a Weather fetcher using createAsyncThunk:

- Input: city name
- Output: temperature, condition
- Use OpenWeatherMap API

◆ **Task 11:**
Create a custom logging middleware that logs:

- action type
- action payload

◆ **Task 12:**
Use Redux DevTools **to inspect actions and state changes for your thunk.**

◆ **Task 13:**
Dispatch multiple thunks from a single button click (e.g., fetch users and fetch

posts together).

Mini Projects :

💡 1. User List Fetcher

- Fetch and display users from `https://jsonplaceholder.typicode.com/users`
- Handle loading and error states

💡 2. Post Viewer

- Fetch posts via thunk
- Display titles
- Add loading and error display

💡 3. Weather Info App

- Input a city → fetch weather from OpenWeatherMap
- Show temperature and condition
- Use loading/error indicators

💡 4. GitHub Profile Finder

- Input username → fetch data from GitHub API
- Show avatar, repos, followers
- Handle error for invalid users

Day 23:

Day 3: Advanced Redux Toolkit in Real Projects

◆ 1. Multiple Slices & Combining State

✅ Concept:

In large apps, we split logic into multiple slices (e.g., authSlice, productSlice, cartSlice) and combine them in the store.

 Example:

store.js

```
js
import { configureStore } from '@reduxjs/toolkit';
import authReducer from '../features/auth/authSlice';
import productReducer from '../features/products/productSlice';

const store = configureStore({
  reducer: {
    auth: authReducer,
    products: productReducer,
  },
});
```

Use slices independently:

```
Js
const isLoggedIn = useSelector((state) => state.auth.isLoggedIn);
const allProducts = useSelector((state) => state.products.items);
```

◆ 2. Entity Adapter (createEntityAdapter)

 Concept:

Simplifies managing normalized data (like a list of users/products) with IDs.

 Example:

```
js
import { createSlice, createEntityAdapter } from '@reduxjs/toolkit';

const userAdapter = createEntityAdapter();

const initialState = userAdapter.getInitialState({
  loading: false,
});

const userSlice = createSlice({
```

```
  name: 'users',
  initialState,
  reducers: {
    userAdded: userAdapter.addOne,
    userRemoved: userAdapter.removeOne,
  },
});

export const { selectAll: selectAllUsers } = userAdapter.getSelectors(
  (state) => state.users
);

export const { userAdded, userRemoved } = userSlice.actions;
export default userSlice.reducer;
```

◆ 3. Memoized Selectors with createSelector

✅ **Concept:**

Improves performance by memoizing derived values, especially from large lists or computations.

 Example:

```
js
import { createSelector } from '@reduxjs/toolkit';

const selectAllProducts = (state) => state.products.items;

export const selectCheapProducts = createSelector(
  [selectAllProducts],
  (products) => products.filter(product => product.price < 100)
);
```

Used in component:

```
js
const cheapItems = useSelector(selectCheapProducts);
```

◆ 4. Custom Middleware (Optional)

✅ **Concept:**

Middleware runs between dispatch and the reducer; used for logging, analytics, etc.

 Example:

```
js
const loggerMiddleware = storeAPI => next => action => {
  console.log('Dispatching:', action);
  return next(action);
};

const store = configureStore({
  reducer: rootReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(loggerMiddleware),
});
```

◆ 5. Persist Redux State (with redux-persist)

✅ **Concept:**

Keep Redux state saved across browser reloads (e.g., cart, auth info).

 Setup:

```
bash
npm install redux-persist
```

```
js
import storage from 'redux-persist/lib/storage';
import { persistReducer, persistStore } from 'redux-persist';
import { configureStore } from '@reduxjs/toolkit';
import rootReducer from './reducers';

const persistConfig = {
  key: 'root',
  storage,
};
```

```
const persistedReducer = persistReducer(persistConfig, rootReducer);

export const store = configureStore({
  reducer: persistedReducer,
});

export const persistor = persistStore(store);
```

Wrap <Provider> with <PersistGate> in index.js.

◆ 6. Testing Redux Slices

✅ Concept:

Test reducer logic directly by calling reducer with an action.

❓ Example:

```
Js
import counterReducer, { increment } from './counterSlice';

test('should handle increment', () => {
  const initialState = { value: 0 };
  const nextState = counterReducer(initialState, increment());
  expect(nextState.value).toBe(1);
});
```

You can also test async thunks by mocking fetch calls and checking states (`loading`, `data`, `error`).

✅ Summary Table

Concept	Use Case
Multiple Slices	Modular architecture in large apps
createEntityAdapter	Manage lists (users/products) efficiently
createSelector	Memoize computed values from state
Custom Middleware	Add logging, analytics, async checks

Concept	Use Case
redux-persist	Save Redux state across refreshes
Slice Testing	Unit test your reducers and actions

Day – 23 Tasks

✅ **Task 1:**
Create two slices:

- authSlice for login status (isLoggedIn)
- profileSlice for user info (name, email)

Use configureStore to combine them and access their states in a component.

✅ **Task 2:**
Update the authSlice to log in/out users and show/hide the user profile conditionally based on isLoggedIn.

✅ **Task 3:**
Add a settingsSlice to handle app theme and language. Use useSelector() to display current settings.

◆ Section 2: createEntityAdapter

✅ **Task 4:**
Use createEntityAdapter to manage a list of products with properties like id, title, price.

✅ **Task 5:**
Add reducer logic to:

- Add one product
- Remove one product
- Update a product's price

Use adapter methods like addOne, removeOne, updateOne.

✅ **Task 6:**
Display the list of products using selectAll selector from the entity adapter.

✅ **Task 7:**
Create a selectProductById selector using adapter's getSelectors() and display a single product detail.

◆ Section 3: createSelector (Memoized Selectors)

✅ **Task 8:**
Write a createSelector to get only expensive products (price > 1000) from the product list.

✅ **Task 9:**
Write another selector to get the total inventory value using reduce() inside createSelector.

✅ **Task 10:**
Use React DevTools and console logs to show that createSelector does not recompute unless input state changes.

◆ Section 4: Middleware & Persist

✅ **Task 11:**
Create a custom middleware that logs:

- Action type
- Time of dispatch
- Current state after the action

✅ **Task 12:**
Integrate redux-persist to persist a cart or auth state between refreshes.
Wrap your root reducer with persistReducer and use <PersistGate> in React.

◆ Section 5: Testing

✅ **Task 13:**
Write a unit test for one of your reducers (e.g., counterSlice) to test if an action updates state correctly.

Example:

```
js
expect(counterReducer({ value: 0 }, increment())).toEqual({ value: 1 });
```

Mini Projects :

💡 **1. Product Inventory with Entity Adapter**
- Use createEntityAdapter to manage products
- Add, update, delete products
- Display total inventory value using selector

💡 **2. Cart System with Persist**
- Add/remove products from cart
- Persist cart across refresh using `redux-persist`

💡 **3. Expense Tracker**
- Input expenses
- Use memoized `createSelector` to compute total spent
- Store expenses using entity adapter

💡 **4. Logger Middleware App**
- Create custom middleware to log actions
- Dispatch actions and see logs in console
- Combine with any app (e.g., Todo)

Day 24:

Error Boundaries in React

What are Error Boundaries in React?

Error Boundaries in React are special components that catch JavaScript errors in their child component tree during rendering, lifecycle methods, or constructor phase and display a fallback UI instead of breaking the entire application.

Why Use Error Boundaries?

Without Error Boundaries, if any component throws an error, the whole React app will crash. Error Boundaries prevent this by displaying a custom fallback UI while the rest of the app continues to work.

Functional Component Approach with Error Boundaries 🔥

Since React 16.8+, Error Boundaries can be implemented in functional components using the ErrorBoundary package or React Error Boundary libraries like react-error-boundary.

1. Install Error Boundary Package

```
npm install react-error-boundary
```

2. Create Error Boundary Component

Create a component to **handle errors gracefully**.

```
import React from "react";
import { ErrorBoundary } from "react-error-boundary";
```

```
const ErrorFallback = ({ error, resetErrorBoundary }) => {
  return (
    <div>
      <h2>Something went wrong!</h2>
      <p>Error: {error.message}</p>
      <button onClick={resetErrorBoundary}>Try Again</button>
    </div>
  );
};

export default ErrorFallback;
```

3. Wrap Components with Error Boundary

Use the ErrorBoundary component from the library to wrap any component that might crash.

```
import React from "react";
import { ErrorBoundary } from "react-error-boundary";
import ErrorFallback from "./ErrorFallback";

const CrashingComponent = () => {
  throw new Error("I crashed!");
};

const App = () => {
  return (
    <ErrorBoundary FallbackComponent={ErrorFallback}>
      <CrashingComponent />
    </ErrorBoundary>
  );
};

export default App;
```

Explanation:

- FallbackComponent: This component shows the custom error message.
- resetErrorBoundary: This function **resets the error boundary** when clicked.
- All child components of ErrorBoundary are **protected** from errors.

Bonus Example with API Error

Catch API request errors in functional components.

```
const FetchData = () => {
  throw new Error("Failed to fetch data!");
};

const App = () => {
  return (
    <ErrorBoundary FallbackComponent={ErrorFallback}>
      <FetchData />
    </ErrorBoundary>
  );
};
```

When Do Error Boundaries Catch Errors?

✓ During **Rendering**
✓ During **Lifecycle Methods**
✓ In **Constructor**

What Error Boundaries Can't Catch:

❌ Errors inside event handlers
❌ Asynchronous code
❌ Server-side rendering
❌ Errors from outside React

Key Takeaways:

- Error Boundaries improve user experience by preventing full app crashes.
- They only catch rendering errors and lifecycle method errors.
- Use third-party libraries like react-error-boundary for functional components.

Day 24 – Tasks:

1. **Introduction to Error Boundaries**: Write a brief explanation of what error boundaries are and why they are used in React.
2. **Basic Error Boundary Setup**: Create a functional component using ErrorBoundary with React hooks.
3. **Error Simulation**: Create a component that throws an error when a button is clicked.
4. **Error Fallback UI**: Display a fallback UI when an error occurs.
5. **Custom Error Message**: Pass custom error messages to the fallback component.
6. **Try-Catch in Functional Components**: Use try-catch inside event handlers to catch synchronous errors.
7. **Global Error State**: Manage error state using useState hook.
8. **Error Boundary with Children**: Wrap multiple child components with the error boundary component.
9. **Reset Error State**: Add a button that resets the error state and re-renders the component.
10. **Log Errors to Console**: Log caught errors to the console.

11. **Async Error Handling**: Catch errors from asynchronous operations inside useEffect.
12. **Reusable Error Boundary**: Create a reusable ErrorBoundary component that can wrap any component.
13. **Multiple Error Boundaries**: Wrap different components with separate error boundaries.

Mini Projects

Mini Project 1: Error Boundary with Counter

Description:
Create a counter app that intentionally throws an error when the count reaches 5, showing a fallback UI.

Mini Project 2: API Fetch Error Boundary

Description:
Build an app that fetches user data from an API and displays a custom error message if the API request fails.

Mini Project 3: Form Submission Error

Description:
Create a form where an error occurs when submitting without filling required fields, displaying an error message.

Mini Project 4: Random Joke Generator with Error Handling

Description:
Fetch random jokes from an API, showing a fallback UI if the API fails or returns an error.

Day 25:

Performance Optimization in React

Why Performance Optimization in React?

In large applications, frequent re-renders and unnecessary computations can slow down performance. React provides built-in hooks and techniques to optimize performance.

1. React.memo

What is React.memo?

React.memo is a higher-order component (HOC) that memorizes the output of a functional component and only re-renders when its props change.

Syntax:

```
import React from "react";

const ChildComponent = ({ name }) => {
  console.log("Child Component Rendered");
  return <h2>Hello, {name}</h2>;
};

export default React.memo(ChildComponent);
```

Explanation:

- If the name prop **doesn't change**, the component will **not re-render**.
- If the name prop **changes**, the component will re-render.

When to Use?

✅ When the component renders the same output given the same props.

✅ For pure functional components.

2. useMemo

What is useMemo?

useMemo is a hook that memorizes the result of a function and only recomputes the result when its dependencies change.

Syntax:

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

const App = () => {
  const [count, setCount] = useState(0);
  const [number, setNumber] = useState(5);

  const expensiveCalculation = useMemo(() => {
    console.log("Calculating...");
    return number * 2;
  }, [number]);

  return (
    <div>
      <h2>Expensive Calculation: {expensiveCalculation}</h2>
      <button onClick={() => setNumber(number + 1)}>Change Number</button>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
    </div>
  );
};

export default App;
```

Explanation:

- The expensive calculation runs **only when the number changes**.
- Clicking the **Increment Count** button won't trigger recalculation.

When to Use?

✓ For **heavy or expensive calculations**.
✓ When you need to **avoid unnecessary recalculations**.

3. useCallback

What is useCallback?

useCallback is a hook that memorizes a function and returns the same function instance between re-renders unless its dependencies change.

Syntax:

```
import React, { useState, useCallback } from "react";
const Child = React.memo(({ onClick }) => {
console.log("Child Component Rendered");
  return <button onClick={onClick}>Click Me</button>;
});

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

  const handleClick = useCallback(() => {
    console.log("Button Clicked");
  }, []);

  return (
    <div>
      <h2>Count: {count}</h2>
      <Child onClick={handleClick} />
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
    </div>
  );
};
export default App;
```

Explanation:

- useCallback ensures that the handleClick function reference stays the **same** between renders.
- Without useCallback, the child component would **always re-render**.

When to Use?

✓ For passing callback functions to child components.
✓ When the child component is wrapped with React.memo.

Summary

Technique	Purpose	When to Use
React.memo	Memoize Component Output	Pure Functional Components
useMemo	Memoize Expensive Calculations	Expensive Calculations
useCallback	Memoize Callback Functions	Passing Callbacks to Children

Day 25 – Tasks:

1. **Introduction to Performance Optimization**: Write a short explanation of why performance optimization is important in React applications.
2. **Install React DevTools**: Set up React DevTools for component profiling.
3. **Identify Performance Issues**: Use React DevTools to find unnecessary component re-renders.
4. **React.memo Basic Usage**: Wrap a functional component with React.memo to prevent re-renders.
5. **React.memo with Props**: Demonstrate how React.memo prevents re-renders when props remain unchanged.
6. **useMemo Basic Example**: Cache the result of an expensive computation using useMemo.

7. **useMemo with Dependency Array**: Control when useMemo recalculates its value using dependencies.
8. **useCallback Basic Example**: Use useCallback to memoize event handlers.
9. **useCallback with Dependencies**: Demonstrate how useCallback updates memoized functions when dependencies change.
10. **Avoid Inline Functions**: Refactor inline functions using useCallback.
11. **Component Profiling**: Use React DevTools Profiler to measure component render time before and after optimization.
12. **Optimization with Pure Functions**: Refactor components to use pure functions where possible.
13. **Prevent Unnecessary Re-renders**: Combine React.memo, useMemo, and useCallback for maximum optimization.

Mini Projects

Mini Project 1: List Filtering

Description:
Create a list filtering app where the filtered list is memoized using useMemo.

Mini Project 2: Counter with Memoized Button

Description:
Build a counter app where the increment button is memoized with React.memo and useCallback.

Mini Project 3: Search Suggestion App

Description:
Implement a search suggestion app where search results are optimized using useMemo and re-renders are minimized.

Mini Project 4: Expensive Calculation App

Description:
Create an app that performs an expensive calculation and optimize it using useMemo.

Day 26:

Server-Side Rendering (SSR) & Static Site Generation (SSG) Topics

What is Server-Side Rendering (SSR)?

Server-Side Rendering (SSR) is a technique where a web page is generated on the server at request time and sent to the client with fully populated HTML content. This improves **SEO** and initial page load performance.

How SSR Works in Next.js?

Next.js provides built-in SSR using the getServerSideProps() function.

Syntax:

```
import React from "react";

const SSRPage = ({ data }) => {
  return (
    <div>
      <h1>Server-Side Rendering Example</h1>
      <p>{data}</p>
    </div>
  );
};
```

```
export async function getServerSideProps() {
  const data = "Hello from SSR!";
  return {
    props: { data },
  };
}
export default SSRPage;
```

Explanation:

- The getServerSideProps() function runs **on every request**.
- The page is pre-rendered **at the time of request**.
- Ideal for pages with **dynamic data** like user dashboards.

Benefits of SSR:

- SEO Friendly
- Faster Initial Page Load
- Real-time Dynamic Data

What is Static Site Generation (SSG)?

Static Site Generation (SSG) pre-renders pages at **build time** and serves them as static HTML files.

How SSG Works in Next.js?

Next.js uses the getStaticProps() function to generate pages at **build time**.

Syntax:

```
import React from "react";
const SSGPage = ({ data }) => {
return (
   <div>
     <h1>Static Site Generation Example</h1>
```

```
    <p>{data}</p>
   </div>
  );
};

export async function getStaticProps() {
  const data = "Hello from SSG!";
  return {
    props: { data },
  };
}
export default SSGPage;
```

Explanation:

- getStaticProps() runs only **once during build time**.
- Ideal for **static content** like blogs and product catalogs.

Benefits of SSG:

- Super Fast Performance
- SEO Friendly
- Cached Pages

SSR vs. SSG Comparison

Feature	SSR	SSG
Rendering Time	On Every Request	At Build Time
SEO Friendly	✅	✅
Performance	Slower	Faster
Use Cases	Dynamic Content (User Data)	Static Content (Blogs)

Day 26 – Tasks :

1. **Introduction to SSR & SSG**: Write a short explanation of what SSR and SSG are and their benefits.
2. **Install Next.js**: Set up a basic Next.js project using npx create-next-app.
3. **Project Folder Structure**: Understand the folder structure of Next.js.
4. **Basic Page Creation**: Create a basic page using the pages directory.
5. **SSR Example**: Use getServerSideProps to fetch data at request time and render the page.
6. **SSG Example**: Use getStaticProps to fetch data at build time.
7. **Dynamic SSG Pages**: Implement getStaticPaths with getStaticProps to generate dynamic pages at build time.
8. **Compare SSR vs SSG**: Create two pages that fetch the same data but using SSR and SSG, and compare their performance.
9. **SEO Benefits**: Add meta tags to pages using the Head component in Next.js.
10. **Fallback Pages in SSG**: Implement fallback loading pages using getStaticPaths.
11. **Revalidate Pages in SSG**: Use revalidate to regenerate static pages at runtime.
12. **Client-Side Navigation**: Use the Link component to navigate between SSR and SSG pages.
13. **Error Handling**: Display custom error pages for failed API requests using Next.js error pages.

Mini Projects

Mini Project 1: Blog Listing with SSR

Description:
Fetch blog posts from an API at request time using SSR and display them with SEO-friendly meta tags.

Mini Project 2: Product Catalog with SSG

Description:
Generate static product catalog pages using SSG with dynamic routes.

Mini Project 3: User Profile Page

Description:
Create a user profile page that uses SSR to fetch user data at request time.

Mini Project 4: News Website

Description:
Build a news website where the homepage uses SSG for top news and individual articles use SSR.

Day 27:

Code Splitting & Lazy Loading

What is Code Splitting in React?

Code splitting is a performance optimization technique that breaks down a large application into smaller, manageable chunks of code. Instead of loading the entire app at once, the code is loaded only when needed.

Why Code Splitting?

✓ Improves page loading time
✓ Reduces initial bundle size
✓ Loads only necessary components
✓ Enhances user experience

1. React.lazy

React.lazy() is used to dynamically import components and enable lazy loading.

Syntax:

```
import React, { lazy, Suspense } from "react";

const LazyComponent = lazy(() => import("./MyComponent"));

const App = () => {
  return (
    <div>
      <h1>Main App</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent />
      </Suspense>
    </div>
  );
};

export default App;
```

Explanation:

- lazy() dynamically imports the component.
- Suspense displays a **fallback UI** while the component is loading.
- The component will only be loaded when it is **needed on the page**.

2. Suspense

Suspense works as a wrapper around lazy components to **display fallback content** until the component is fully loaded.

Syntax:

```
<Suspense fallback={<div>Loading Component...</div>}>
  <LazyComponent />
</Suspense>
```

Fallback Props

The fallback prop takes any **React element** (like loading spinners or text).

3. Benefits of Code Splitting

Feature	Description
Performance	Faster initial loading time
UX	Better user experience
Lazy Loading	Loads components **on demand**
SEO Friendly	Improves page rendering speed

When to Use Code Splitting?

✓ Large Applications
✓ Routes and Pages
✓ Components with heavy logic
✓ Third-party libraries

Bonus Example

Dynamic Route-Based Code Splitting

```
import { lazy, Suspense } from "react";
import { BrowserRouter as Router, Route, Switch } from "react-router-dom";

const Home = lazy(() => import("./Home"));
const About = lazy(() => import("./About"));

const App = () => {
  return (
    <Router>
      <Suspense fallback={<h3>Loading Page...</h3>}>
        <Switch>
          <Route path="/" exact component={Home} />
          <Route path="/about" component={About} />
        </Switch>
      </Suspense>
    </Router>
  );
};

export default App;
```

Day 27 – Tasks:

1. **Introduction to Code Splitting**: Write a short explanation of what code splitting is and why it improves performance.
2. **Install Project**: Set up a React project using npx create-react-app.
3. **Basic Component Creation**: Create two simple components (Header and Footer).
4. **Default Import**: Import components normally without lazy loading.
5. **React.lazy Setup**: Refactor one component to use React.lazy.

6. **Suspense Usage**: Wrap the lazy-loaded component with Suspense and add a fallback loading message.
7. **Multiple Lazy Components**: Lazy load both Header and Footer components with Suspense.
8. **Error Handling with Suspense**: Show an error message if the component fails to load.
9. **Lazy Loaded Routes**: Use React.lazy with React Router to lazy load page components.
10. **Dynamic Import Explanation**: Write code to dynamically import a utility function only when needed.
11. **Code Splitting with Functions**: Use import() to dynamically import a utility function inside an event handler.
12. **Profiling Performance Gains**: Use React DevTools to measure performance before and after code splitting.
13. **Chunk Naming**: Customize chunk names in Webpack with /* webpackChunkName: "chunkName" */.

Mini Projects

Mini Project 1: Lazy Loaded Dashboard

Description:
Create a dashboard app where sidebar, header, and content sections are lazy-loaded separately.

Mini Project 2: Image Gallery

Description:
Build an image gallery app that lazy loads images in separate chunks.

Mini Project 3: Weather App

Description:
Display weather information with different components for current weather and forecast, both lazy-loaded.

Mini Project 4: E-Commerce Product Page

Description:
Create a product page where product details, reviews, and suggestions are lazy-loaded in separate components.

Main Project

1. Expense Tracker App

Description:
A simple app that allows users to add, edit, and delete expenses with dynamic calculations.
Topics Covered:

- Functional Components
- useState
- Conditional Rendering
- Forms
- Context API
- useEffect
- Local Storage
- Error Boundaries

2. E-Commerce Website

Description:
A fully functional e-commerce website with product listing, cart functionality, and checkout.
Topics Covered:

- React Router
- Redux (State Management)
- Axios for API Requests

- Context API
- useState & useEffect
- Lazy Loading
- Error Boundaries
- Form Validation

3. Weather Forecast App

Description:
Fetch and display weather data based on user input location with lazy loading and API integration.
Topics Covered:

- Axios API Calls
- useState & useEffect
- Conditional Rendering
- Error Boundaries
- React.memo
- useMemo
- Dynamic Routing

4. Blogging Platform

Description:
A blogging platform where users can create, edit, delete, and view blogs.
Topics Covered:

- CRUD Operations
- Redux
- SSR & SSG with Next.js
- Axios API Requests
- Form Handling
- Authentication HOC

- React Router
- Custom Hooks

5. Movie Search App

Description:
Search for movies using an API, display movie details, and allow users to add favorites.
Topics Covered:

- Axios API Requests
- useState & useEffect
- React Router
- Context API
- Custom Hooks
- Pagination
- Lazy Loading

Day 28 :

E-Commerce Integration: React with Django APIs

1. Connecting React with Django APIs using Axios

Axios is a popular **HTTP client library** used to send API requests from React to Django backend.

Installation:

```
npm install axios
```

Example API Call:

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

```
const ProductList = () => {
  const [products, setProducts] = useState([]);

  useEffect(() => {
    axios.get("http://localhost:8000/api/products/")
      .then(response => setProducts(response.data))
      .catch(error => console.log(error));
  }, []);

  return (
    <div>
      {products.map(product => (
        <p key={product.id}>{product.name}</p>
      ))}
    </div>
  );
};

export default ProductList;
```

2. Managing Cart with Redux & Backend Sync

Redux stores the cart items globally and syncs them with the backend API.

Install Redux:

```
npm install redux react-redux @reduxjs/toolkit
```

Redux Store Setup:

```
import { createSlice, configureStore } from "@reduxjs/toolkit";

const cartSlice = createSlice({
  name: "cart",
  initialState: [],
  reducers: {
    addToCart: (state, action) => {
```

```
      state.push(action.payload);
    },
    removeFromCart: (state, action) => {
      return state.filter(item => item.id !== action.payload);
    },
  },
});

export const { addToCart, removeFromCart } = cartSlice.actions;
export const store = configureStore({ reducer: { cart: cartSlice.reducer } });
```

Using Redux in Components:

```
import { useDispatch } from "react-redux";
import { addToCart } from "../store";

const Product = ({ product }) => {
  const dispatch = useDispatch();

  const handleAddToCart = () => {
    dispatch(addToCart(product));
  };

  return (
    <div>
      <h3>{product.name}</h3>
      <button onClick={handleAddToCart}>Add to Cart</button>
    </div>
  );
};

export default Product;
```

3. Order Placement & Checkout Process

Order data is sent to the backend for processing.

Checkout API Call:

```
const placeOrder = async (cart) => {
  try {
    const response = await axios.post("http://localhost:8000/api/orders/", { cart });
    alert("Order placed successfully");
  } catch (error) {
    alert("Order failed");
  }
};
```

4. Storing Cart Data in LocalStorage & Database

Save Cart to LocalStorage:

```
useEffect(() => {
  localStorage.setItem("cart", JSON.stringify(cart));
}, [cart]);
```

Load Cart from LocalStorage:

```
const loadCart = () => {
  const cartData = JSON.parse(localStorage.getItem("cart"));
  if (cartData) {
    setCart(cartData);
  }
};
```

5. Secure Routes (Only logged-in users can place orders)

Secure pages like Checkout require **authentication**.

Private Route Component:

```
import { Navigate } from "react-router-dom";
```

```
const PrivateRoute = ({ children, isAuthenticated }) => {
  return isAuthenticated ? children : <Navigate to="/login" />;
};
```

Using Private Route:

```
<Route path="/checkout" element={<PrivateRoute
isAuthenticated={user}>{<Checkout />}</PrivateRoute>} />
```

Day 29:

Testing in React

Introduction to Testing in React

Testing in React ensures that components and applications work as expected, making the app more reliable and bug-free**.**

Why Testing?

✓ Catch bugs early
✓ Improve code quality
✓ Ensure components work as expected
✓ Refactor code with confidence

1. Tools for Testing in React

- **Jest:** JavaScript testing framework for unit tests
- **React Testing Library:** Utility to test React components

Install Testing Libraries

```
npm install --save-dev jest @testing-library/react @testing-library/jest-dom
```

2. Writing Unit Tests with Jest

Basic Jest Test Example:

Create a file named sum.js:

```
export const sum = (a, b) => a + b;
```

Create the test file sum.test.js:

```
import { sum } from "./sum";

test("adds 2 + 3 to equal 5", () => {
  expect(sum(2, 3)).toBe(5);
});
```

Run Tests:

```
npm test
```

3. Testing React Components with React Testing Library

Component Example:

Create Button.jsx:

```
const Button = ({ label, onClick }) => {
  return <button onClick={onClick}>{label}</button>;
};

export default Button;
```

Writing Test:

Create Button.test.js:

```
import { render, screen, fireEvent } from "@testing-library/react";
import Button from "./Button";

test("renders button with label", () => {
  render(<Button label="Click Me" />);
  expect(screen.getByText("Click Me")).toBeInTheDocument();
});

test("calls onClick function when clicked", () => {
  const handleClick = jest.fn();
  render(<Button label="Click Me" onClick={handleClick} />);
  fireEvent.click(screen.getByText("Click Me"));
  expect(handleClick).toHaveBeenCalledTimes(1);
});
```

4. Testing Form Components

Create LoginForm.jsx:

```
const LoginForm = ({ onSubmit }) => {
  return (
    <form onSubmit={onSubmit}>
      <input type="text" placeholder="Username" />
      <button type="submit">Login</button>
    </form>
  );
};

export default LoginForm;
```

Writing Test:

```
import { render, fireEvent, screen } from "@testing-library/react";
import LoginForm from "./LoginForm";
```

```
test("calls onSubmit when form is submitted", () => {
  const handleSubmit = jest.fn();
  render(<LoginForm onSubmit={handleSubmit} />);

  fireEvent.submit(screen.getByRole("form"));
  expect(handleSubmit).toHaveBeenCalled();
});
```

5. Best Practices for Testing

- Write tests for **every component**
- Cover **edge cases**
- Use **mock functions** for API calls
- Avoid testing **implementation details**

Day 30:

Deployment & Final Project

Introduction to Deployment in React

Deployment is the final step in the development process where your React application is built, optimized, and made available for users on the web.

Why Deployment?

✅ Makes your app accessible online
✅ Improves performance with production build
✅ Allows users to interact with your app

1. Building the React App for Production

Before deploying, you need to create a **production build** of your app.

Steps to Build the App:

1. Open terminal
2. Navigate to your project folder
3. Run the following command:

```
npm run build
```

What Happens?

✓ Minifies JS & CSS files
✓ Optimizes the app for faster loading
✓ Creates a build/ folder

2. Deploying React App to Netlify

Netlify is a free platform to deploy static websites.

Steps:

1. Go to **https://www.netlify.com/**
2. Sign up with GitHub
3. Create a new site
4. Drag & Drop your build/ folder
5. Netlify will automatically deploy the app

3. Deploying React App to Vercel

Vercel is another free hosting platform for frontend apps.

Steps:

1. Go to **https://vercel.com/**
2. Sign up with GitHub
3. Install Vercel CLI:

```
npm install -g vercel
```

4. **Run the command:**

```
vercel
```

5. Follow the prompts

4. Folder Structure Before Deployment

```
my-app/
|
├ – public/          # Static files
├ – src/             # React Components
├ – build/           # Production Build
└ package.json       # Dependencies
```

5. Environment Variables

Store sensitive information like API keys.

Create .env File:

```
REACT_APP_API_URL=https://api.example.com
```

Access in Components:

```
const apiUrl = process.env.REACT_APP_API_URL;
```

6. Deployment Best Practices

Task	Platform
Static Websites	Netlify
Fullstack Projects	Vercel
Environment Variables	.env