**React**

**React Router** is a standard library for routing in React applications. It enables navigation among views of various components, allows changing the browser URL, and keeps the UI in sync with the URL.

**📦 Install React Router**

For React Router DOM (used in web apps):

bash

CopyEdit

npm install react-router-dom

**🧩 Core Concepts**

1. **BrowserRouter**: Uses HTML5 history API to keep your UI in sync with the URL.
2. **Routes**: Wraps your <Route> components.
3. **Route**: Defines a path and the component to render.
4. **Link**: Used instead of <a> for internal navigation.
5. **useNavigate()**: A hook to navigate programmatically.

**🔧 Basic Example**

jsx

CopyEdit

import React from 'react';

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

function Home() {

return <h2>Home Page</h2>;

}

function About() {

return <h2>About Page</h2>;

}

function App() {

return (

<BrowserRouter>

<nav>

<Link to="/">Home</Link> |

<Link to="/about">About</Link>

</nav>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/about" element={<About />} />

</Routes>

</BrowserRouter>

);

}

export default App;

**🧠 Useful Hooks**

| **Hook** | **Purpose** |
| --- | --- |
| useNavigate() | Navigate programmatically |
| useParams() | Get route parameters like /user/:id |
| useLocation() | Get the current URL object |
| useMatch() | Match a path to current route |

**🧭 Navigate Programmatically**

jsx

CopyEdit

import { useNavigate } from 'react-router-dom';

function Login() {

const navigate = useNavigate();

const handleLogin = () => {

// after successful login

navigate('/dashboard');

};

return <button onClick={handleLogin}>Login</button>;

}

Handling events in React is similar to handling events in plain HTML and JavaScript, but with some key differences due to React's JSX syntax and component-based structure.

**🔹 Basic Syntax for Event Handling in React**

In HTML:

html

CopyEdit

<button onclick="handleClick()">Click Me</button>

In React (JSX):

jsx

CopyEdit

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

✅ **Note:** In JSX, events use camelCase (onClick instead of onclick) and you pass the function **without parentheses** unless you’re calling it with parameters.

**🔹 Example: Handling Click Event**

jsx

CopyEdit

import React from 'react';

function MyComponent() {

const handleClick = () => {

alert("Button Clicked!");

};

return (

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

);

}

**🔹 Passing Arguments to Event Handlers**

If you want to pass parameters, use an arrow function:

jsx

CopyEdit

<button onClick={() => handleClick("Soumi")}>Greet</button>

jsx

CopyEdit

const handleClick = (name) => {

alert(`Hello, ${name}`);

};

**🔹 Handling Events in Class Components**

jsx

CopyEdit

class MyComponent extends React.Component {

handleClick() {

alert('Button clicked!');

}

render() {

return (

<button onClick={this.handleClick.bind(this)}>Click Me</button>

);

}

}

Or using class fields to avoid .bind():

jsx

CopyEdit

class MyComponent extends React.Component {

handleClick = () => {

alert('Button clicked!');

};

render() {

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

}

}

**🔹 Common Events in React**

| **Event** | **Description** |
| --- | --- |
| onClick | Fires when an element is clicked |
| onChange | Fires when input value changes |
| onSubmit | Fires on form submission |
| onMouseOver | Fires when mouse hovers over an element |
| onKeyDown | Fires when a key is pressed |

**🔹 Example: Handling Form Input**

jsx

CopyEdit

function FormComponent() {

const [inputValue, setInputValue] = React.useState("");

const handleChange = (event) => {

setInputValue(event.target.value);

};

const handleSubmit = (event) => {

event.preventDefault(); // prevents page reload

alert(`Submitted: ${inputValue}`);

};

return (

<form onSubmit={handleSubmit}>

<input type="text" value={inputValue} onChange={handleChange} />

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

</form>

);

}

**✅ useMemo Hook in React – Explained Simply**

The useMemo hook is used in React to **optimize performance** by **memoizing** (i.e., caching) the result of a **computationally expensive function**, so that it's **recomputed only when needed**.

**📌 Syntax:**

js

CopyEdit

const memoizedValue = useMemo(() => {

// expensive calculation

return computeExpensiveValue(a, b);

}, [a, b]);

**🧠 What It Does:**

* React normally **re-renders** components when state or props change.
* This can lead to **expensive functions** running again unnecessarily.
* useMemo helps you **store (memoize)** the result of a function **unless its dependencies change**.

**💡 When to Use useMemo:**

* When a function does **heavy calculations** (like filtering, sorting large lists).
* To **avoid re-rendering** child components unnecessarily.
* When used with **React.memo()** to improve performance.

**🧪 Example:**

js

CopyEdit

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

function ExpensiveComponent({ num }) {

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

const expensiveCalculation = (n) => {

console.log("Calculating...");

let result = 0;

for (let i = 0; i < 1000000000; i++) {

result += n;

}

return result;

};

const memoizedValue = useMemo(() => expensiveCalculation(num), [num]);

return (

<div>

<h2>Expensive Calculation Result: {memoizedValue}</h2>

<button onClick={() => setCount(count + 1)}>Increment Count ({count})</button>

</div>

);

}

**🔍 In This Example:**

* The expensive calculation runs **only when num changes**, not on every render.
* Even if the button is clicked many times, the heavy calculation won't rerun.

**⚠️ Note:**

* Use useMemo **only when needed**. Overusing it can make the code harder to read.
* It’s a **performance optimization**, not a tool for avoiding rendering altogether.

**✅ useCallback Hook in React – Explained Simply**

The useCallback hook is used to **memoize a function** so that it doesn't get **recreated on every render** — unless its **dependencies change**.

**📌 Syntax:**

js

CopyEdit

const memoizedCallback = useCallback(() => {

// function logic

}, [dependency1, dependency2]);

**🧠 Why Use useCallback?**

Every time a component re-renders, functions are **recreated** in memory.  
This is usually fine, but it can cause **performance issues** when:

* The function is passed to a **child component** that depends on reference equality (React.memo)
* You want to **avoid re-running effects** or computations when the function hasn’t changed

**🔄 Difference Between useMemo vs useCallback**

| **Hook** | **Caches a...** | **Use Case** |
| --- | --- | --- |
| useMemo | **Value/result** | Expensive computation result |
| useCallback | **Function itself** | Pass stable function references |

**🧪 Example:**

jsx

CopyEdit

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

const Child = React.memo(({ onClick }) => {

console.log('Child rendered');

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

});

function App() {

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

const handleClick = useCallback(() => {

console.log("Button clicked!");

}, []);

return (

<div>

<h2>Count: {count}</h2>

<button onClick={() => setCount(count + 1)}>Increment</button>

<Child onClick={handleClick} />

</div>

);

}

export default App;

**🔍 What's Happening Here?**

* Child is wrapped with React.memo, so it will **only re-render** if its props change.
* Without useCallback, handleClick would be recreated every render and cause Child to re-render unnecessarily.
* With useCallback, handleClick stays the **same reference** unless dependencies change.

**⚠️ When *Not* to Use useCallback**

* If you're not passing the function to a memoized child
* If the function isn’t causing performance issues  
  Overusing useCallback can **add complexity without benefit**.

**🔹 What is Redux?**

**Redux** is a **state management library** often used with **React** (but not limited to it) to manage and centralize application state.

**🔧 Why Use Redux?**

In large applications, managing state (like user data, UI state, form data) across multiple components can become complex. Redux helps by:

* Keeping the **entire state in one central store**
* Allowing any component to access or update state predictably
* Making debugging and testing easier

**🧠 Core Concepts of Redux:**

1. **Store** – The single source of truth that holds your entire app’s state.
2. **Actions** – Plain JavaScript objects that describe *what happened* (e.g., { type: "INCREMENT" }).
3. **Reducers** – Pure functions that take the current state and an action, then return the new state.
4. **Dispatch** – A function to send actions to the store to trigger state changes.
5. **Selectors** – Functions to get specific data from the store.

**🧩 Simple Flow:**

1. A **component** dispatches an **action** (e.g., user clicked a button).
2. The **action** goes to the **reducer**.
3. The **reducer** updates the **store**.
4. The UI automatically reflects the updated state.

**🛠️ Example Use Case:**

* Managing user authentication status
* Shopping cart data
* Dark/light theme toggling
* Handling API call results (loading, error, success)