React

**React Basics**

* What is React?
* Setting up a React Project (npx create-react-app)
* Understanding JSX (JavaScript XML)
* Components (Functional vs. Class Components)
* Props and Prop Drilling
* State Management with useState
* Lifecycle Methods and useEffect
* Handling Events in React
* Forms in React (Controlled Components)

**3. React Router**

* Introduction to React Router
* Setting up Routes with react-router-dom
* Navigating Between Pages (Link Component)
* Route Parameters (useParams)
* Protected Routes (Optional)

**4. Bootstrap in React**

* Why Use Bootstrap?
* Installing Bootstrap in React
* Using Bootstrap Components (Buttons, Forms, Tables)

# Introduction to React

React, also known as ReactJS, is a powerful JavaScript library for building dynamic and interactive user interfaces, primarily for single-page applications (SPAs). It was developed and maintained by Facebook and has gained significant popularity due to its efficient rendering techniques, reusable components, and active community support.

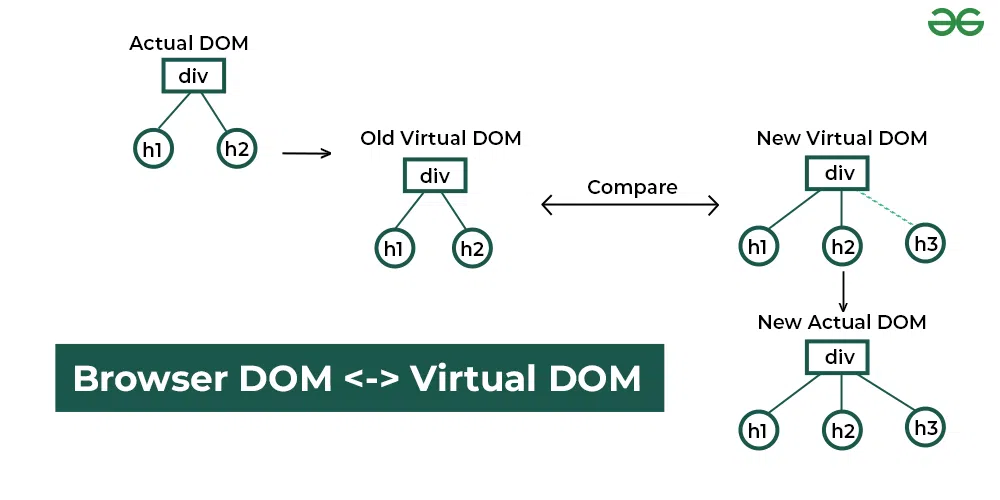
React is a JavaScript library for building user interfaces (UIs) on the web. React is a declarative, component-based library that allows developers to build reusable UI components. It follows the Virtual DOM (Document Object Model) approach, which optimizes rendering performance by minimizing DOM updates.

### History of React

React was developed by Facebook engineers who found traditional DOM manipulation inefficient. By implementing a virtual DOM, React significantly improved performance, leading to its rapid adoption. The latest stable version of ReactJS is **18.2.0** (as of June 14, 2022).

### How Does React Work?

React operates by creating an in-memory virtual DOM rather than directly modifying the browser’s DOM. This approach allows React to efficiently update only the necessary parts of the DOM, improving performance and responsiveness.



### Features of React

1. **Component-Based Architecture** - React breaks the UI into self-contained components, making code more reusable and easier to maintain.
2. **JSX (JavaScript XML)** - JSX allows developers to write HTML-like code inside JavaScript, making components more readable and expressive.
3. **Virtual DOM** - React maintains a lightweight version of the actual DOM in memory, updating only the changed parts when necessary.
4. **One-Way Data Binding** - Data flows from parent to child components, ensuring predictability.
5. **Performance Optimization** - React updates only the necessary elements using the virtual DOM, reducing rendering time.
6. **Single-Page Applications (SPAs)** - React is widely used to build SPAs where content updates dynamically without full-page reloads.

# Setting Up a React Project (npx create-react-app)

Before building a React application, we need to **set up a development environment**. React provides a simple way to start a new project using **Create React App (CRA)**, which includes everything needed to run a React application without manual configuration.

### 1. Prerequisites

To set up a React project, ensure you have:

**Node.js** installed (version 14 or later recommended). You can check if Node.js is installed using:

node -v

**npm (Node Package Manager)**, which comes bundled with Node.js. You can check the version using:

npm -v

### 2. Creating a New React App

To create a new React project, run the following command in the terminal:

npx create-react-app my-app

* npx is a package runner that comes with npm. It downloads and executes the latest version of create-react-app without installing it globally.
* create-react-app my-app sets up a new React project inside the my-app folder.

### 3. Navigating into the Project Folder

After the project is created, move into the project directory:

cd my-app

### 4. Starting the Development Server

To start the React app, run:

npm start

This launches a local development server and opens the application in your browser at http://localhost:3000/.

# Understanding JSX (JavaScript XML)

### What is JSX?

JSX (JavaScript XML) is a syntax extension for JavaScript that allows writing HTML-like code inside JavaScript. It is used in React to describe UI components. JSX makes it easier to write and visualize the structure of React components.

### Example of JSX

const element = <h1>Hello, World!</h1>;

Instead of using document.createElement(), JSX allows us to write UI components in an **HTML-like syntax**.

### Why Use JSX?

1. **Improved Readability** – It is more readable compared to traditional JavaScript methods of creating elements.
2. **Component-Based Approach** – JSX simplifies writing UI components.
3. **Prevents XSS (Cross-Site Scripting)** – JSX escapes values before rendering to prevent malicious code execution.

## Embedding JavaScript in JSX

You can insert JavaScript expressions inside JSX using **curly braces {}**:

const name = "Alice";

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

* name is dynamically inserted into the JSX.

### JSX Must Return a Single Root Element

JSX must return only **one parent element**.

|  |
| --- |
| const myElement = <h1>React is {5 + 5} times better with JSX</h1>; |

To write HTML on multiple lines, put the HTML inside parentheses:

|  |
| --- |
| const myElement = (  <ul>  <li>Apples</li>  <li>Bananas</li>  <li>Cherries</li>  </ul>  ); |

The HTML code must be wrapped in *ONE* top level element.

So if you like to write *two* paragraphs, you must put them inside a parent element, like a div element.

|  |
| --- |
| const myElement = (  <div>  <p>I am a paragraph.</p>  <p>I am a paragraph too.</p>  </div>  ); |

Alternatively, you can use a "fragment" to wrap multiple lines. This will prevent unnecessarily adding extra nodes to the DOM.

A fragment looks like an empty HTML tag: <></>.

|  |
| --- |
| const myElement = (  <>  <p>I am a paragraph.</p>  <p>I am a paragraph too.</p>  </>  ); |

JSX follows XML rules, and therefore HTML elements must be properly closed.

### Attribute class = className

The class attribute is a much used attribute in HTML, but since JSX is rendered as JavaScript, and the class keyword is a reserved word in JavaScript, you are not allowed to use it in JSX.

### Use attribute className instead.

JSX solved this by using className instead. When JSX is rendered, it translates className attributes into class attributes.

### React supports if statements, but not inside JSX.

To be able to use conditional statements in JSX, you should put the if statements outside of the JSX, or you could use a ternary expression instead.

## Understanding React:

A Javascript for building fast and interactive user interfaces. It is still the most popular library for UI. We also have Angular and Vue.

### Component:

A component is essentially a piece of the user interface. When building applications with React we build a bunch of independent, isolated and reusable components and then compose them to build complex user interfaces. Every React application has at least one component which we refer to as the root component. This component represents the internal application and contains other child components so every React application is essentially a tree of components.

Each component has some state and a render method. The state here is the data that we want to display when the component is rendered and the render method is responsible for describing what the UI should look like. The output of this render method is a React element which is a simple plain JavaScript object that maps to a DOM element. It's not a real DOM element, it is just a plain JavaScript object that represents that DOM element in memory so React keeps a lightweight representation of the DOM in memory which we refer to as the virtual DOM.

Unlike the browser or the real DOM, this virtual DOM is cheap to create. When we change the state of a component, we get a new React element. React will then compare this element and its children with the previous one. It figures out what is changed and then it will update a part of the real DOM to keep it in sync with the virtual DOM. So that means when building applications in React, unlike vanilla JavaScript or jQuery, we no longer have to work with the DOM API in browsers.

In other words, we no longer have to write code in query and manipulate the DOM or attach event handlers to DOM elements. We simply change the state of our components and React will automatically update the FOM to match that state.

And that's exactly why this library is called React because when the state changes react essentially reacts to the state change and updates the DOM.

# React Entry Point – index.js

index.js is the entry point of your React application. It’s responsible for rendering your React components into the **DOM (Document Object Model)**.

**Key Parts of index.js:**

|  |
| --- |
| import React from 'react'; // Imports React library  import ReactDOM from 'react-dom/client'; // Imports ReactDOM for rendering  import './index.css'; // Imports global styles  import App from './App'; // Imports the main App component  const root = ReactDOM.createRoot(document.getElementById('root')); // Selects the root div  root.render(  <React.StrictMode>  <App /> {/\* Renders the App component \*/}  </React.StrictMode>  ); |

### What’s Happening Here?

1. **Imports React & ReactDOM** → These are needed to work with React components and render them into the browser.
2. **Imports App.js** → This is your main React component (like the home screen).
3. **Finds root div in index.html** → The React app is injected inside the <div id="root"></div> in public/index.html.
4. **Renders the App Component** → The <App /> component is displayed inside the browser.

### Why is index.js Important?

* It **connects React to the browser** by rendering components inside the root div.
* It **organizes the app structure**, ensuring React components are managed properly.
* It **applies global styles** (like index.css).

## Understanding <React.StrictMode>

<React.StrictMode> is a **special wrapper** provided by React to **help developers** write better code. It **does not affect the UI** but helps in detecting potential issues during development.

### 1️. What Does It Do?

* **Identifies unsafe lifecycle methods** (for class components).
* **Warns about deprecated features** before they cause problems.
* **Detects side effects in useEffect that run twice in development mode** (this helps find bugs).
* **Ensures future-proof code** by preparing it for upcoming React updates.

### 2️. Where Is It Used?

It wraps around your entire app in index.js:

### 3. Does It Affect Production?

No! **React.StrictMode only works in development mode**. It won’t slow down or affect performance in production.

### 4️. What Happens If I Remove It?

Your app will still work fine, but you might **miss out on helpful warnings** that React provides to improve your code.

With <React.StrictMode>, React will **run this effect twice** (only in development mode). This helps catch issues like unwanted side effects before they become real bugs.

**Should You Keep It?**

Yes! It’s good practice to **keep <React.StrictMode>** to ensure your app is future-proof and bug-free.

## Step-by-Step Breakdown:

### 1. The HTML File (index.html in public folder)

This is a normal HTML file, but **React does not modify it directly**. Instead, React puts all of its content **inside the <div id="root">** element.

|  |
| --- |
| <body>  <div id="root"></div> <!-- React will inject components here -->  </body> |

**Think of <div id="root"> as a placeholder where React will insert the UI.**

### 2. The JavaScript Code (index.js)

This is where we tell React **where** to render our content inside the webpage.

|  |
| --- |
| const container = document.getElementById('root'); // Selects the <div id="root"> from HTML  const root = ReactDOM.createRoot(container); // Creates a React root inside the div  root.render(<p>Hello</p>); // Inserts <p>Hello</p> inside <div id="root"> |

### 3. What Happens in the Browser?

Once React runs this code, your webpage will look like this:

|  |
| --- |
| <body>  <div id="root">  <p>Hello</p> <!-- React injected this into the div -->  </div>  </body> |

**Why is This Useful?**

* **Dynamic UI**: Instead of writing static HTML, React **injects content dynamically** based on user actions.
* **Single Page Application (SPA)**: The page **doesn’t reload** when the UI updates, making it **faster**.
* **Component-Based Structure**: We can break the UI into **reusable components** like buttons, forms, etc.

# React Components

Components are the **building blocks** of a React application. A React component is a **reusable, independent piece of UI** that can be used to build complex interfaces.

There are **two main types** of components in React:

1. **Functional Components** (Preferred in modern React)
2. **Class Components** (Older approach, still used in legacy applications)

## 1. Functional Components

A **functional component** is a JavaScript function that returns **JSX (JavaScript XML)**. It takes **props** (input data) as arguments and returns a React element that describes what should be rendered in the UI.

**Example of a Functional Component**

|  |
| --- |
| import React from "react";  const Greeting = (props) => {  return <h1>Hello, {props.name}!</h1>;  };  export default Greeting; |

* This component accepts props (short for properties) as an argument.
* It returns JSX (<h1>Hello, {props.name}!</h1>).
* The Greeting component can be reused multiple times with different names.

### Using the Functional Compo\nent in Another Component

|  |
| --- |
| import Greeting from "./Greeting";  const App = () => {  return (  <div>  <Greeting name="Alice" />  <Greeting name="Bob" />  </div>  );  };  export default App; |

## 2. Class Components (Older Approach)

Before React Hooks (useState, useEffect, etc.), React used **class-based components** to manage state and lifecycle methods.

**Example of a Class Component**

|  |
| --- |
| import React, { Component } from "react";  class Greeting extends Component {  render() {  return <h1>Hello, {this.props.name}!</h1>;  }  }  export default Greeting; |

* Instead of a function, we define a **class** that extends Component.
* The render() method returns JSX.
* this.props is used to access props inside class components.

**Using the Class Component in Another Component**

|  |
| --- |
| import Greeting from "./Greeting";  class App extends React.Component {  render() {  return (  <div>  <Greeting name="Alice" />  <Greeting name="Bob" />  </div>  );  }  }  export default App; |

This produces the same output as the functional component example.

## 3. Key Differences Between Functional and Class Components

|  |  |  |
| --- | --- | --- |
| **Feature** | **Functional Component** | **Class Component** |
| **Definition** | Uses a JavaScript function | Uses a JavaScript class |
| **Syntax** | Simple and concise | More boilerplate code |
| **Props Handling** | props is passed as function arguments | Uses this.props |
| **State Handling** | Uses the useState hook | Uses this.state and setState() |
| **Lifecycle Methods** | Uses useEffect for side effects | Uses lifecycle methods like componentDidMount, componentDidUpdate, componentWillUnmount |
| **Performance** | Faster, with less overhead | Slightly slower due to more complex structure |
| **Modern Usage** | Preferred in new React projects | Used in older React versions |

### 5. Why Functional Components are Preferred Today

1. **Simpler Code** – Functional components require less code and are easier to read.
2. **Hooks Provide Full Functionality** – useState, useEffect, and other hooks eliminate the need for class components.
3. **Better Performance** – Functional components execute faster than class components.
4. **Future-Proof** – React encourages using functional components with hooks.

### 6. When to Use Functional vs. Class Components

* Use **functional components** for almost all scenarios in modern React development.
* Use **class components** only when working with legacy codebases that require class lifecycle methods.

# React Lists and Keys

React efficiently renders lists using the map() method.

|  |
| --- |
| const cities = ["Delhi", "Mumbai", "Bangalore"];  const CityList = () => (  <ul>  {cities.map((city, index) => (  <li key={index}>{city}</li>  ))}  </ul>  ); |

# Conditional Rendering

React allows conditional rendering using logical operators or ternary conditions.

|  |
| --- |
| const Greeting = ({ isLoggedIn }) => (  isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please sign in</h1>  ); |

# Props

Props (short for "properties") are **read-only** inputs that allow data to be passed from a parent component to a child component.

### 1. Why Do We Need Props?

* To make components **reusable** by passing different data.
* To allow **parent components to control child components**.
* Unlike **state**, props **cannot be changed** inside the child component.

**Example: Passing Props to a Component**

Let's create a Header component that **receives a title** as a prop.

### Parent Component (App.js)

|  |
| --- |
| import React from 'react';  import Header from './Header';  function App() {  return (  <div>  <Header title="Welcome to My Website" />  <Header title="React Props Example" />  </div>  );  }  export default App; |

### Child Component (Header.js)

|  |
| --- |
| import React from 'react';  function Header(props) {  return (  <header>  <h1>{props.title}</h1>  </header>  );  }  export default Header; |

**How It Works**

* The App component passes a title prop to the Header component.
* The Header component receives the props object and displays props.title.
* Each instance of Header can have a different title.

## Props vs State

|  |  |  |
| --- | --- | --- |
| **Feature** | **Props** | **State** |
| Can be modified inside the component? | No | Yes |
| Controlled by parent component? | Yes | No |
| Can trigger re-render? | Yes | Yes |
| Used for dynamic UI updates? | No | Yes |
| Reusable across components? | Yes | No |

Props are best for **static or parent-controlled data**, while state is used when the component needs to manage changing data itself.

# React Hooks

### Class Components (State + Render Method)

In **class components**, we use a **state object** and the render() method to display UI.

## state and render

React is a UI **component-based** library that allows for **dynamic updates**. **State** and the **render mechanism** help achieve this by ensuring the UI updates when data changes.

### What is state in React?

state is an object that stores **dynamic data** in a component. Unlike props (which are **read-only**), state **changes over time**. State is like a **memory** for a component. It helps React **remember things** between renders.

Imagine you have a **counter app**. Without state, the number would **never change** when you click a button. With state, React **remembers the number** and updates the screen.

### Why Do We Need state?

* Helps React **remember data** between renders.
* Allows components to be **interactive** (e.g., form inputs, toggles, counters).
* Triggers UI updates **automatically** when state changes.

### What is render() in React?

The render() method (in class components) **returns the JSX/UI**.  
In **functional components**, there’s no render()—React **re-renders** the component whenever the state changes.

### Why Do We Need render()?

* Ensures the UI is always **up-to-date**.
* React **automatically updates** only the parts of the UI that changed.

### Example: Counter Without State (Won't Work!)

If we don’t use state, the UI **won't update** when clicking a button.

|  |
| --- |
| function Counter() {  let count = 0; // ❌ Normal variable (UI won't update)  function increase() {  count += 1;  console.log(count);  }  return (  <div>  <h2>Counter: {count}</h2>  <button onClick={increase}>Increment</button>  </div>  );  }  export default Counter; |

* Clicking the button **logs** the updated count, but the **UI doesn’t change**!
* React **doesn’t know** when to update the UI.

### Example: Counter With state (Works!)

Using useState, React **re-renders** when count changes.

|  |
| --- |
| import React, { useState } from 'react';  function Counter() {  const [count, setCount] = useState(0); // state  function increase() {  setCount(count + 1); // State changes, React re-renders  }  return (  <div>  <h2>Counter: {count}</h2>  <button onClick={increase}>Increment</button>  </div>  );  }  export default Counter; |

**Now the UI updates** each time the button is clicked because setCount triggers a re-render.

### When to Use render()?

* In **class components**, render() is needed to return JSX.
* In **functional components**, React re-renders automatically when state updates.

React introduced hooks to manage component logic without class components:

**useState Hook**

**useEffect Hook**

## Understanding useState and useEffect with a Real-World Example

Let’s break this down with a simple analogy:

### useState: Managing State in React

Think of useState as a notebook where you keep track of changing values.

Real-World Example: A Digital Counter in a Shopping App

* Imagine a shopping website where a user clicks the "Add to Cart" button.
* Every time the user clicks, the number of items in the cart should update.

### useEffect: Handling Side Effects

Think of useEffect as a reminder system.

* It runs code automatically when something changes in your component.

Real-World Example: Fetching Data When a Page Loads

* Imagine a weather app. When a user opens the page, it should automatically fetch the latest weather data.

# State Management with useState

State in React is a way for components to store and update data. The useState hook allows functional components to have their own **state**.

**Example: Counter with useState**

|  |
| --- |
| import React, { useState } from 'react';  function Counter() {  const [count, setCount] = useState(0);  return (  <div>  <h2>Count: {count}</h2>  <button onClick={() => setCount(count + 1)}>Increase</button>  </div>  );  }  export default Counter; |

This is a **React Hook** called useState, which is used to manage state in a functional component. Let's break it down:

useState **returns an array** where:

1. The **first element** is the current state (count).
2. The **second element** is the function (setCount) to update the state.

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

* useState(0): This initializes a **state variable** with a default value of 0.
* count: This is the **current state value**.
* setCount: This is a **function used to update the state**.

In React, you can have **any state variable** using useState. It’s not limited to numbers; you can store **strings, arrays, objects, booleans, or even other complex data structures**.

# Lifecycle Methods and useEffect

In **class components**, lifecycle methods manage component behavior, but in **functional components**, we use the useEffect hook instead.

## React Lifecycle Stages

Each React component goes through **three lifecycle phases**:

1️. **Mounting Phase** (Component is added to the DOM)

* componentDidMount() → Runs once after the component is first rendered.
* **Example use case:** Fetching data from an API.

2️. **Updating Phase** (Component updates due to state or prop changes)

* componentDidUpdate(prevProps, prevState) → Runs after every update.
* **Example use case:** Running side effects when state changes.

3️. **Unmounting Phase** (Component is removed from the DOM)

* componentWillUnmount() → Runs before the component is destroyed.
* **Example use case:** Cleaning up event listeners.

## Using useEffect Instead of Lifecycle Methods

useEffect can handle all three lifecycle phases in functional **components**.

You **don’t call lifecycle methods like componentDidMount, componentDidUpdate, or componentWillUnmount manually** in React functional components. Instead, React automatically handles them for you.

**Example: Using useEffect**

|  |
| --- |
| import React, { useState, useEffect } from 'react';  function Timer() {  const [count, setCount] = useState(0);  useEffect(() => {  console.log("Component Mounted or Updated");  return () => {  console.log("Component Will Unmount (Cleanup)");  };  }, [count]); // Runs whenever `count` changes  return (  <div>  <h2>Timer: {count}</h2>  <button onClick={() => setCount(count + 1)}>Increase</button>  </div>  );  }  export default Timer; |

## How useEffect Replaces Lifecycle Methods

|  |  |
| --- | --- |
| **Lifecycle Method** | **Equivalent useEffect** |
| componentDidMount() | useEffect(() => {...}, []) |
| componentDidUpdate() | useEffect(() => {...}, [state]) |
| componentWillUnmount() | useEffect(() => { return () => {...} }, []) |

# Handling Events in React

React handles events similarly to JavaScript but follows JSX syntax. Instead of using onclick, onchange, etc., in lowercase, React uses **camelCase event handlers** like onClick, onChange, etc.

**Example: Handling a Button Click**

|  |
| --- |
| const ClickMe = () => (  <button onClick={() => alert("Button Clicked!")}>Click Me</button>  );  export default ClickMe; |

* The onClick event triggers an alert when the button is clicked.

# Form Handling

Forms in React use **controlled components**, meaning form elements (like <input>, <textarea>, and <select>) are controlled by **state**. This allows React to manage the form values dynamically.

## Basic Form Handling in React

Here’s how we create a simple form with an input field and update the state when the user types.

**Example: Controlled Input Field**

|  |
| --- |
| import { useState } from "react";  const SimpleForm = () => {  const [name, setName] = useState(""); // State for input  return (  <form>  <label>Enter your name:</label>  <input  type="text"  value={name} // Controlled input  onChange={(e) => setName(e.target.value)} // Update state  />  <p>Your name: {name}</p>  </form>  );  };  export default SimpleForm; |

**How This Works**

1. **State Management** → useState("") initializes name as an empty string.
2. **Controlled Input** → The value of the input is set to name, ensuring React **controls** the input field.
3. **Event Handling** → onChange={(e) => setName(e.target.value)} updates name as the user types.

## Handling Form Submission

When the user submits the form, we prevent the default behavior (e.preventDefault()) and handle the data.

**Example: Handling Form Submission**

|  |
| --- |
| import { useState } from "react";  const FormSubmitExample = () => {  const [name, setName] = useState("");  const handleSubmit = (e) => {  e.preventDefault(); // Prevents page reload  alert(`Form submitted with name: ${name}`);  };  return (  <form onSubmit={handleSubmit}>  <label>Enter your name:</label>  <input  type="text"  value={name}  onChange={(e) => setName(e.target.value)}  />  <button type="submit">Submit</button>  </form>  );  };  export default FormSubmitExample; |

**How This Works**

1. onSubmit={handleSubmit} → Calls handleSubmit when the form is submitted.
2. e.preventDefault() → Prevents the page from reloading (default behavior of form submission).
3. alert() → Displays the submitted name.

## Handling Multiple Input Fields

When managing multiple fields (e.g., name and email), we use **one state object** instead of multiple useState() hooks.

**Example: Handling Multiple Inputs**

|  |
| --- |
| import { useState } from "react";  const MultiFieldForm = () => {  const [formData, setFormData] = useState({  name: "",  email: "",  });  const handleChange = (e) => {  setFormData({  ...formData, // Spread previous state  [e.target.name]: e.target.value, // Update specific field  });  };  const handleSubmit = (e) => {  e.preventDefault();  alert(`Name: ${formData.name}, Email: ${formData.email}`);  };  return (  <form onSubmit={handleSubmit}>  <label>Name:</label>  <input type="text" name="name" value={formData.name} onChange={handleChange} />  <label>Email:</label>  <input type="email" name="email" value={formData.email} onChange={handleChange} />  <button type="submit">Submit</button>  </form>  );  };  export default MultiFieldForm; |

**How This Works**

1. useState({ name: "", email: "" }) → Stores multiple input values.
2. handleChange() updates only the specific field using e.target.name.
3. ...formData (spread operator) keeps the existing data intact.

## Handling Checkbox, Radio, and Select Inputs

**Example: Handling Checkbox, Radio, and Select Inputs**

|  |
| --- |
| import { useState } from "react";  const AdvancedForm = () => {  const [formData, setFormData] = useState({  username: "",  gender: "male",  subscribe: false,  });  const handleChange = (e) => {  const { name, value, type, checked } = e.target;  setFormData((prevData) => ({  ...prevData,  [name]: type === "checkbox" ? checked : value, // Handle checkbox differently  }));  };  const handleSubmit = (e) => {  e.preventDefault();  alert(`User: ${formData.username}, Gender: ${formData.gender}, Subscribed: ${formData.subscribe}`);  };  return (  <form onSubmit={handleSubmit}>  <label>Username:</label>  <input type="text" name="username" value={formData.username} onChange={handleChange} />  <label>Gender:</label>  <select name="gender" value={formData.gender} onChange={handleChange}>  <option value="male">Male</option>  <option value="female">Female</option>  </select>  <label>  <input type="checkbox" name="subscribe" checked={formData.subscribe} onChange={handleChange} />  Subscribe to newsletter  </label>  <button type="submit">Submit</button>  </form>  );  };  export default AdvancedForm; |

**How This Works**

1. **Checkbox Handling** → checked stores true or false instead of a string.
2. **Dropdown Handling** → select input updates state like a text field.
3. **One handleChange Function** → Handles different input types dynamically.

## Summary: Key Concepts in React Form Handling

|  |  |
| --- | --- |
| **Concept** | **Explanation** |
| **Controlled Components** | Inputs are controlled by React state (value is set from state). |
| **Handling Change (onChange)** | Updates state as user types (setState(e.target.value)). |
| **Handling Submit (onSubmit)** | Prevents page reload (e.preventDefault()). |
| **Managing Multiple Inputs** | Store values in an object and update using field name ([e.target.name]). |
| **Handling Checkbox and Select** | Use checked for checkboxes, value for dropdowns. |

# React Router

React Router is a library that allows navigation between different pages (or views) in a React application without needing to reload the page.

## Introduction to React Router

In a single-page application (SPA), React Router helps **switch between views dynamically** based on the URL. It uses the **History API** to change the browser’s URL without a full page reload.

Before using React Router, install it in your React project:

|  |
| --- |
| npm install react-router-dom |

## Setting Up Routes with react-router-dom

To define routes, wrap the entire application inside BrowserRouter and define paths using Route.

### Example: Setting Up Routes

|  |
| --- |
| import { BrowserRouter as Router, Routes, Route } from "react-router-dom";  import Home from "./Home";  import About from "./About";  import Contact from "./Contact";  const App = () => {  return (  <Router>  <Routes>  <Route path="/" element={<Home />} />  <Route path="/about" element={<About />} />  <Route path="/contact" element={<Contact />} />  </Routes>  </Router>  );  };  export default App; |

**How This Works**

1. **BrowserRouter (Router)**: Wraps the entire application to enable routing.
2. **Routes**: Groups multiple Route components.
3. **Route**: Defines a path and the component to render when that path is visited.
4. **path="/"**: The root URL (/) loads the Home component.
5. **path="/about"**: Loads the About component when navigating to /about.

## Navigating Between Pages (Link Component)

Instead of <a href=""> (which refreshes the page), React Router provides the <Link> component for navigation.

**Example: Navigation with Link**

|  |
| --- |
| import { Link } from "react-router-dom";  const Navbar = () => {  return (  <nav>  <Link to="/">Home</Link>  <Link to="/about">About</Link>  <Link to="/contact">Contact</Link>  </nav>  );  };  export default Navbar; |

**How This Works**

* to="/" navigates to the homepage.
* to="/about" navigates to the About page.
* to="/contact" navigates to the Contact page.
* Unlike <a href="">, <Link> **does not reload the page**.

## Adding Navbar to App.js

|  |
| --- |
| import Navbar from "./Navbar";  const App = () => {  return (  <Router>  <Navbar />  <Routes>  <Route path="/" element={<Home />} />  <Route path="/about" element={<About />} />  <Route path="/contact" element={<Contact />} />  </Routes>  </Router>  );  }; |

## Route Parameters (useParams)

React Router allows dynamic routes using parameters (:id) to handle data dynamically.

**Example: Dynamic Routes (useParams)**

|  |
| --- |
| import { useParams } from "react-router-dom";  const UserProfile = () => {  const { username } = useParams(); // Get dynamic parameter  return <h1>Welcome, {username}!</h1>;  };  export default UserProfile; |

### Adding the Route in App.js

|  |
| --- |
| <Route path="/user/:username" element={<UserProfile />} /> |

### Navigating to a Dynamic Route

|  |
| --- |
| <Link to="/user/john">Go to John's Profile</Link> |

**When visiting /user/john**, the page will display:

**Welcome, John!**

### How This Works

* :username in the route acts as a **placeholder** for dynamic values.
* useParams() extracts the parameter value from the URL.
* If the user visits /user/jane, the page shows **Welcome, Jane!**

## Redirects and 404 Pages

If a user tries to visit /old-page, we can **redirect** them to /new-page using Navigate.

|  |
| --- |
| import { Navigate } from "react-router-dom";  const OldPage = () => {  return <Navigate to="/new-page" />;  }; |

**404 Page (Catch-All Route)**

|  |
| --- |
| const NotFound = () => <h1>404 - Page Not Found</h1>;  <Route path="\*" element={<NotFound />} /> |

Any undefined route will show the **NotFound** component.

## Summary: Key React Router Concepts

|  |  |
| --- | --- |
| **Concept** | **Explanation** |
| **<BrowserRouter>** | Wraps the entire app to enable routing. |
| **<Routes>** | Groups multiple route definitions. |
| **<Route path="/" element={<Home />}>** | Defines which component to show for a specific URL. |
| **<Link to="/about">About</Link>** | Navigates between pages without reloading. |
| **useParams()** | Extracts dynamic route parameters (e.g., /user/:id). |
| **Navigate to="/new-page"** | Redirects users to another route. |

# Bootstrap in React

### Installing Bootstrap via NPM (Recommended for Vite & Create React App)

You can install Bootstrap using npm or yarn in your React project.

**Installation Command**

|  |
| --- |
| npm install bootstrap |

**Import Bootstrap in main.jsx (Vite) or index.js (CRA)**

|  |
| --- |
| import 'bootstrap/dist/css/bootstrap.min.css'; |

Now you can use Bootstrap classes in your React components.

### Using Bootstrap CDN (No Installation Required)

If you don’t want to install Bootstrap, you can add it directly to the index.html file inside the <head> section.

**Add this in index.html (Vite)**

|  |
| --- |
| <link  rel="stylesheet"  href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"  /> |

This will allow you to use Bootstrap without installing it.