

Introduction to ReactJS

ReactJS is an open-source JavaScript library developed by **Meta (formerly Facebook)**. It is used for building user interfaces (UIs) and is particularly focused on creating **single-page applications (SPAs)**. React simplifies the process of developing dynamic and interactive web applications by enabling developers to build reusable components.

```
npx create-react-app myapp --use-npm
```

```
cd myapp
```

```
npm install react@18 react-dom@18
```

```
npm start
```

Key Features of ReactJS

1. Component-Based Architecture

React applications are built using components, which are reusable, independent pieces of UI.

- Example: A button, a navigation bar, or an entire page.

2. Virtual DOM

React uses a **virtual DOM** to efficiently update the UI. Instead of directly interacting with the real DOM, React creates a virtual copy of it and applies changes there first. This improves performance by minimizing direct DOM manipulation.

3. Declarative Syntax

React allows developers to describe the "what" instead of the "how" for the UI. You define the desired state of your UI, and React takes care of updating it when the data changes.

4. Unidirectional Data Flow

React follows a **one-way data binding** approach, making data management predictable and easier to debug.

5. JSX (JavaScript XML)

JSX is an extension of JavaScript used in React that allows developers to write HTML-like code within JavaScript.

- Example:

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

6. React Hooks

Introduced in React 16.8, Hooks allow you to use state and other React features in functional components.

- Examples: `useState`, `useEffect`.

Benefits of ReactJS

1. **Reusability**
Components can be reused across projects, saving development time and ensuring consistency.
 2. **Improved Performance**
The Virtual DOM ensures fast updates and efficient rendering.
 3. **SEO Friendly**
With server-side rendering (SSR) via tools like Next.js, React helps improve search engine optimization.
 4. **Large Ecosystem**
React has a massive developer community and many supporting libraries like React Router, Redux, and Material-UI.
 5. **Cross-Platform Development**
React Native, an extension of ReactJS, is used for building mobile applications.
-

Example Code

Here's a simple React component:

```
import React from 'react';

function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}

export default Greeting;
```

Usage in an application:

```
import React from 'react';
import ReactDOM from 'react-dom';
import Greeting from './Greeting';

ReactDOM.render(<Greeting name="John" />, document.getElementById('root'));
```

Popular Applications Built Using ReactJS

1. Facebook
 2. Instagram
 3. Netflix
 4. Airbnb
 5. Uber
-

Why Learn ReactJS?

1. **High Demand:** Many companies use ReactJS for front-end development.
2. **Modern Approach:** React encourages modular, maintainable, and scalable development.

3. **Great Career Prospects:** It is widely adopted in the industry and offers opportunities to work on exciting projects.

Let me know if you'd like examples or help with setting up a ReactJS environment!

Explaining the Virtual DOM in an Interesting Way

Imagine you are a **detective solving cases** (the browser's task). The **real DOM** is like visiting every house in a city, knocking on every door to find a single suspect. It takes a lot of time and effort because you check every house one by one, even if nothing changes.

Now, here comes the **Virtual DOM**—your **smart assistant**. This assistant knows the layout of the city (the DOM structure) and keeps a detailed **map** of all the houses. When there's a change, your assistant compares the new map to the old one, finds the exact house where the change happened, and takes you straight there.

This saves you tons of time because you only visit the affected house, not the entire city!

Step-by-Step Working of the Virtual DOM (in Detective Mode 🕵️🗺️♂️🗨️)

1. **Initial Case Setup**

When the web page is loaded, the Virtual DOM creates a **map** (a virtual representation of the real DOM). This map mirrors the real DOM but is lightweight and exists in memory.

Analogy: You, the detective, ask your assistant to draw a map of the city when you first arrive.

2. **Detect Changes**

When the user interacts with the app (e.g., clicking a button or typing), React creates a **new version** of the Virtual DOM with these updates.

Analogy: The assistant notices that someone changed the paint color of a house or added a new fence.

3. **Diffing Algorithm**

React compares the new Virtual DOM with the old one to figure out exactly what has changed. This process is called "**diffing**".

Analogy: The assistant uses the old map to quickly spot differences like a new fence or a window being added.

4. **Patch Updates**

React applies only the necessary changes to the real DOM instead of reloading the entire page. This is called **reconciliation**.

Analogy: Instead of revisiting every house, you only visit the house with the new fence and paint.

Example Code with Virtual DOM in Action

Here's an example to visualize:

1. HTML Structure:

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

2. React Code:

```
import React, { useState } from 'react';
import ReactDOM from 'react-dom';

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

  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

ReactDOM.render(<App />, document.getElementById('root'));
```

3. How Virtual DOM Works Here:

- Initially, the Virtual DOM creates a map with `<h1>Counter: 0</h1>`.
- When the button is clicked, the Virtual DOM updates the counter value to 1.
- It compares the new Virtual DOM with the old one and finds that only the `<h1>` element has changed.
- The real DOM is updated **only** for `<h1>` without re-rendering the button or `<div>`.

Here are some real-world websites and platforms where you can demonstrate the **Virtual DOM** concept to your students:

1. Facebook (News Feed Update)

Scenario:

- Ask students to log in to Facebook and scroll through the **News Feed**.
- Explain how React (used by Facebook) updates only the specific post that gets new likes or comments instead of reloading the entire feed.

Demo Idea:

- Open the browser developer tools.
 - Show the **Network tab** and highlight how only specific elements get updated without refreshing the entire page.
-

2. Instagram (Live Likes on Posts)

Scenario:

- When a user likes a photo or comment, React updates only the heart icon and the like count.
- The entire page doesn't reload; only that specific part of the Virtual DOM is updated.

Demo Idea:

- Have two students open the same Instagram post on different devices.
 - One student likes the photo; the other instantly sees the update, demonstrating Virtual DOM efficiency.
-

3. Netflix (Content Updates)

Scenario:

- When you hover over a movie or series, additional information (like the description or trailer) appears.
- React ensures that only the hovered section is updated, not the entire movie grid.

Demo Idea:

- Open Netflix and hover over multiple movies.
 - Highlight how smooth the transitions are because of React's Virtual DOM handling.
-

4. Airbnb (Filter Updates)

Scenario:

- When users apply filters (e.g., price range, location), Airbnb dynamically updates the results section without reloading the entire page.
- The Virtual DOM ensures that only the results container is updated.

Demo Idea:

- Open Airbnb and apply some filters.

- Use the **Elements tab** in developer tools to show how only certain sections are modified dynamically.
-

5. GitHub (File/Code Viewer)

Scenario:

- When viewing a file in a repository and switching between different branches, only the file content is updated without reloading the page.
- React powers this smooth file navigation.

Demo Idea:

- Open a public repository on GitHub.
 - Switch between branches or files and demonstrate the instantaneous updates.
-

6. Trello (Task Updates)

Scenario:

- When you drag a task card from one list to another, only the task lists are updated.
- The Virtual DOM ensures smooth transitions and prevents unnecessary re-rendering.

Demo Idea:

- Create a sample Trello board.
 - Drag and drop cards between lists while showing how the interface updates instantly without affecting the rest of the board.
-

7. CodeSandbox (Online Code Editor)

Scenario:

- When you type code in the editor, the preview updates in real-time without refreshing the page.
- CodeSandbox uses React to efficiently handle these updates via the Virtual DOM.

Demo Idea:

- Open CodeSandbox and type a simple React app.
- Show how changes in the code instantly reflect in the live preview without refreshing the page.

Introduction to JSX (JavaScript XML)

JSX (JavaScript XML) is a **syntax extension** for JavaScript that allows you to write **HTML-like code inside JavaScript**. It makes writing UI components in React easier and more readable.

✓ What is JSX?

In regular JavaScript, you would create an element like this:

```
const heading = React.createElement("h1", {}, "Hello, React!");
```

But with JSX, you can write:

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

✦ **JSX makes the code more readable and looks like HTML!**

✦ It is **not HTML** but gets **converted into JavaScript** behind the scenes.

🔍 Why Use JSX?

1. **Easier to Read & Write** 📖

- Instead of calling `React.createElement()`, JSX lets you write components like HTML.

2. **More Powerful Than HTML** ⚡

- You can use **JavaScript expressions** inside JSX using `{}`.

3. **Prevents Injection Attacks** 🛡️

- JSX **automatically escapes** values to prevent cross-site scripting (XSS) attacks.
-

JSX Syntax & Examples

1 Embedding JavaScript in JSX

- You can use JavaScript expressions inside `{}`.

```
const name = "John";
const greeting = <h1>Hello, {name}!</h1>; // Output: Hello, John!
```

- You can also call functions inside JSX:

```
function formatName(user) {
  return user.firstName + " " + user.lastName;
}

const user = { firstName: "Alice", lastName: "Smith" };
```

```
const greeting = <h1>Hello, {formatName(user)}!</h1>; // Output: Hello, Alice Smith!
```

2 ❏ JSX Must Have One Parent Element

✗ This won't work:

```
return (  
  <h1>Hello</h1>  
  <p>Welcome to React</p>  
);
```

✓ Wrap elements inside a `<div>` or a **React Fragment** (`<> ... </>`):

```
return (  
  <>  
    <h1>Hello</h1>  
    <p>Welcome to React</p>  
  </>  
);
```

3 ❏ Adding Attributes in JSX

- JSX uses **camelCase** for attributes instead of lowercase HTML attributes.

HTML	JSX
<code><div class="container"></code>	<code><div className="container"></code>
<code><input type="text" /></code>	<code><input type="text" /></code>
<code><button onclick="handleClick()"></code>	<code><button onClick={handleClick}></code>

✦ In JSX, **class** is written as **className** because `class` is a reserved keyword in JavaScript.

Example:

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

4 ❏ JSX with Conditional Rendering

You can use **ternary operators** inside JSX:

```
const isLoggedIn = true;  
const message = <h1>{isLoggedIn ? "Welcome back!" : "Please log in"}</h1>;
```

Or **short-circuit rendering**:


```
const isAdmin = true;
return <{isAdmin && <h2>Admin Panel</h2>}</>;
```

★ If `isAdmin` is `true`, it shows `<h2>Admin Panel</h2>`, otherwise, it renders nothing.

5 JSX with Lists (Rendering Multiple Elements)

JSX can be used inside `.map()` to create lists dynamically.

```
const users = ["Alice", "Bob", "Charlie"];

const userList = (
  <ul>
    {users.map((user, index) => (
      <li key={index}>{user}</li>
    ))}
  </ul>
);
```

★ Always use a **key prop** when rendering lists to help React efficiently update items.

6 JSX in Function Components

JSX is commonly used inside React **functional components**.

```
function Welcome(props) {
  return <h1>Hello, {props.name}!</h1>;
}

export default Welcome;
```

★ This component can be used like:

```
<Welcome name="John" />
```

How JSX Works Behind the Scenes?

JSX is **not HTML**. It gets **converted into JavaScript** before rendering.

For example, this JSX:

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

is transformed into:

```
const element = React.createElement("h1", null, "Hello, React!");
```

React Component Lifecycle Explained

In React, every component goes through a lifecycle consisting of **mounting, updating, and unmounting**. These phases determine how the component behaves and when React executes specific logic.

1. Mounting (Component Creation Phase)

Happens when a component is first added to the DOM.

Class Component Lifecycle Methods

- `constructor()` → Initializes state & binds methods.
- `static getDerivedStateFromProps(props, state)` → Updates state based on props before rendering.
- `render()` → Returns JSX to be displayed.
- `componentDidMount()` → Runs **once after the component is added** to the DOM (useful for API calls, subscriptions, etc.).

Functional Equivalent (Hooks)

```
useEffect(() => {  
  console.log("Component Mounted");  
}, []); // Empty dependency array means it runs only once after mounting.
```

2. Updating (Component Re-Renders)

Happens when **state or props change**.

Class Component Lifecycle Methods

- `static getDerivedStateFromProps(props, state)` → Runs on every update if props change.
- `shouldComponentUpdate(nextProps, nextState)` → Controls whether the component should re-render.
- `render()` → Re-renders JSX.
- `getSnapshotBeforeUpdate(prevProps, prevState)` → Captures values before DOM updates.
- `componentDidUpdate(prevProps, prevState)` → Runs after an update (useful for API calls when props/state change).

Functional Equivalent (Hooks)

```
useEffect(() => {  
  console.log("Component Updated");  
}, [someState]); // Runs when someState changes.
```

● 3. Unmounting (Component Removal Phase)

Happens when the component is removed from the DOM.

Class Component Lifecycle Methods

- `componentWillUnmount()` → Runs before the component is removed (useful for cleanup like event listeners or timers).

Functional Equivalent (Hooks)

```
useEffect(() => {  
  return () => {  
    console.log("Component Will Unmount");  
  };  
}, []); // Cleanup function runs when component unmounts.
```

Lifecycle Methods: Are They Still Executed in React 18+?

Lifecycle Method	Status in React 18+	Replacement in Functional Components
<code>constructor()</code>	✓ Still used	<code>useState</code> for initializing state
<code>static getDerivedStateFromProps()</code>	✓ Still used	<code>useEffect</code> with <code>useState</code>
<code>render()</code>	✓ Still used	Just return JSX in functional components
<code>componentDidMount()</code>	✓ Still used	<code>useEffect(() => {...}, [])</code>
<code>shouldComponentUpdate()</code>	✓ Still used	<code>React.memo()</code> for optimization
<code>getSnapshotBeforeUpdate()</code>	✓ Still used	<code>useEffect</code> (rarely needed)
<code>componentDidUpdate()</code>	✓ Still used	<code>useEffect(() => {...}, [dependency])</code>
<code>componentWillUnmount()</code>	✓ Still used	<code>useEffect(() => { return () => {...} }, [])</code>
<code>componentWillMount()</code>	✗ Deprecated	Use <code>useEffect</code> instead
<code>componentWillUpdate()</code>	✗ Deprecated	Use <code>useEffect</code> instead
<code>componentWillReceiveProps()</code>	✗ Deprecated	Use <code>useEffect</code> instead

✚ Understanding Props in React (with Simple Explanation & Examples)

🔗 What Are Props in React?

📖 **Props (short for "Properties")** are a way to **pass data from a parent component to a child component** in React.

📖 Props **make components dynamic & reusable** by allowing different inputs.

📖 Props are **read-only**, meaning **child components cannot modify them** directly.

📖 Basic Syntax of Props

```
<ComponentName propName="value" />
```

- `ComponentName` → The React component.
 - `propName` → The name of the property (customizable).
 - `"value"` → The value being passed to the child component.
-

🔗 Example 1: Passing Props to a Component

✚ Let's create a **Greeting component** that receives a `name` prop.

◆ Step 1: Create `Greeting.js` (Child Component)

```
import React from "react";

function Greeting(props) {
  return <h1>Hello, {props.name}! 🙌</h1>;
}

export default Greeting;
```

✚ Here, `props.name` is used to display the `name` passed by the parent component.

◆ Step 2: Use `Greeting` in `App.js` (Parent Component)

```
import React from "react";
import Greeting from "../Greeting";

function App() {
  return (
    <div>
      <Greeting name="Alice" />
      <Greeting name="Bob" />
      <Greeting name="Charlie" />
    </div>
  );
}
```

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

✦ This will render:

```
Hello, Alice! 🖱  
Hello, Bob! 🖱  
Hello, Charlie! 🖱
```

✓ We used the same **Greeting** component for different names by passing different props.

🔗 Example 2: Passing Multiple Props

✦ Let's create a **Profile Card** that accepts name, age, and image props.

◆ Step 1: Create `ProfileCard.js`

```
jsx  
CopyEdit  
import React from "react";  
  
function ProfileCard(props) {  
  return (  
    <div className="card">  
      <img src={props.image} alt={props.name} className="profile-img" />  
      <h2>{props.name}</h2>  
      <p>Age: {props.age}</p>  
    </div>  
  );  
}  
  
export default ProfileCard;
```

✦ This component takes three props:

- `props.name` → Displays the user's name.
 - `props.age` → Displays the user's age.
 - `props.image` → Displays the user's profile picture.
-

◆ Step 2: Use `ProfileCard` in `App.js`

```
import React from "react";  
import ProfileCard from "../ProfileCard";  
  
function App() {  
  return (  
    <div>
```

```

    <ProfileCard
      name="Alice Johnson"
      age={25}
      image="https://randomuser.me/api/portraits/women/50.jpg"
    />
    <ProfileCard
      name="Bob Smith"
      age={30}
      image="https://randomuser.me/api/portraits/men/50.jpg"
    />
  </div>
);
}

export default App;

```

✦ What Happens?

- We **pass different props** (name, age, and image) for different users.
- The **same component** (ProfileCard) is used multiple times with different data.
- This makes our UI **dynamic and reusable**.

Example 3: Props with Default Values

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

◆ Example:

```

function Welcome(props) {
  return <h1>Welcome, {props.name || "Guest"}!</h1>;
}

// Default prop
Welcome.defaultProps = {
  name: "Guest",
};

export default Welcome;

```

✦ If no name prop is passed, it will show **"Welcome, Guest!"**.

Example 4: Passing Functions as Props

🔗 Props can also be **functions**, allowing child components to send data back to the parent.

◆ Step 1: Create Button.js (Child Component)

```

import React from "react";

function Button(props) {

```

```
    return <button onClick={props.handleClick}>Click Me!</button>;
  }

  export default Button;
```

✦ Here, `props.handleClick` is a function passed by the parent component.

◆ Step 2: Use `Button` in `App.js`

```
import React from "react";
import Button from "../Button";

function App() {
  const showMessage = () => {
    alert("Button Clicked! 🚀");
  };

  return <Button handleClick={showMessage} />;
}

export default App;
```

✦ What Happens?

- The `handleClick` function is passed as a prop.
- When the button is clicked, it triggers the function and shows an alert.

✦ Events in React

In React, **events** are similar to events in HTML (e.g., `onclick`, `onchange`), but they follow a **camelCase naming convention** and work using **JSX syntax**.

🔗 How React Events Work

1. **Naming:** React events use camelCase (e.g., `onClick`, `onChange`).
2. **Handlers:** Event handlers are passed as **functions** (not strings).
3. **Synthetic Events:** React uses a **SyntheticEvent** wrapper for cross-browser compatibility.

🔗 Adding an Event Handler

Here's how to handle a button click event:

Example: Button Click

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

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

export default App;
```

◆ Common Events in React

Event	Description	Example
onClick	Triggered on a click	<button onClick={fn}>
onChange	Triggered on input change	<input onChange={fn}>
onSubmit	Triggered on form submission	<form onSubmit={fn}>
onMouseOver	Triggered when hovered	<div onMouseOver={fn}>
onKeyDown	Triggered on key press	<input onKeyDown={fn}>
onFocus	Triggered on focus	<input onFocus={fn}>
onBlur	Triggered when focus leaves	<input onBlur={fn}>

◆ Example: Input Field Handling

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

  return (
    <div>
      <input type="text" onChange={handleChange} placeholder="Type something..." />
    </div>
  );
}

export default App;
```

★ What Happens?

- The `onChange` event triggers every time the input value changes.
 - The `event.target.value` contains the current input value.
-

◆ Passing Parameters to Event Handlers

Example: Button with Dynamic Parameter

```
function App() {
  const greetUser = (name) => {
```



```

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

  return (
    <div>
      <button onClick={() => greetUser("Saurabh")}>Greet Saurabh</button>
      <button onClick={() => greetUser("Mahesh")}>Greet Mahesh</button>
    </div>
  );
}

export default App;

```

★ Why use an arrow function?

- Directly passing `greetUser("Saurabh")` would call the function immediately.
- Wrapping it in an arrow function delays execution until the button is clicked.

🔗 Prevent Default Behavior

Some events (e.g., `onSubmit`) have default behaviors that need to be prevented.

Example: Prevent Form Submission

```

function App() {
  const handleSubmit = (event) => {
    event.preventDefault();
    alert("Form Submitted!");
  };

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

export default App;

```

★ `event.preventDefault()` prevents the page from reloading after form submission.

🔗 Event Binding

Event handlers in React automatically bind to the component instance, so there's no need to manually bind them in most cases.

🚀 Real-Time Example: Counter App

Here's a simple app to demonstrate events:

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

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

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}

export default Counter;
```

★ What Happens?

- Clicking **Increment** increases the count.
- Clicking **Decrement** decreases the count.

🔗 SyntheticEvent in React

React events are instances of `SyntheticEvent`, a wrapper around the browser's native events. It provides:

1. Cross-browser compatibility.
2. A consistent interface for accessing event properties.

Example: Accessing Event Details

```
function App() {
  const handleClick = (event) => {
    console.log("Event Type:", event.type);
    console.log("Target Element:", event.target);
  };

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

export default App;
```

★ Conditional Rendering in React

Conditional rendering in React means **showing or hiding UI elements** based on certain conditions. Instead of writing separate templates like in other frameworks, React lets you use **JavaScript conditions** inside JSX.

🔗 1. Using `if` Statement

This is the simplest way to conditionally render components.

```
function Greeting({ isLoggedIn }) {
  if (isLoggedIn) {
    return <h1>Welcome back, Rajesh! 🙌</h1>;
  } else {
    return <h1>Please log in to continue. 🚫</h1>;
  }
}

export default function App() {
  return <Greeting isLoggedIn={true} />;
}
```

★ What Happens?

- If `isLoggedIn` is `true`, it shows **"Welcome back, Rajesh!"**
- If `false`, it shows **"Please log in to continue."**

🔗 2. Using Ternary Operator (`? :`)

If the condition is simple, we can use the ternary operator inside JSX.

```
function App() {
  const user = "Neha";

  return (
    <div>
      <h1>{user ? `Hello, ${user}! 🙌` : "Guest User"}</h1>
    </div>
  );
}

export default App;
```

★ What Happens?

- If `user` is **not empty**, it shows **"Hello, Neha! 🙌"**.
- If `user` is `null` or `""`, it shows **"Guest User"**.

🔗 3. Using `&&` (Short-Circuit Rendering)

If you only need to show something when a condition is **true**, use `&&`.

```
function App() {
  const isAdmin = true;

  return (
```

```

    <div>
      <h1>Welcome to the Dashboard</h1>
      {isAdmin && <button>🔑 Admin Panel</button>}
    </div>
  );
}

export default App;

```

★ What Happens?

- If `isAdmin` is `true`, it shows the **Admin Panel button**.
- If `false`, nothing is displayed.

🔗 4. Using `||` (Default Value)

If a variable is `false`, `null`, or `undefined`, use `||` to show a **fallback value**.

```

function App() {
  const city = "";

  return <h1>City: {city || "Not Available"}</h1>;
}

export default App;

```

★ What Happens?

- If `city` is **empty**, it shows **"Not Available"**.
- If `city` has a value (e.g., `"Mumbai"`), it shows **"City: Mumbai"**.

🔗 5. Using `switch` for Multiple Conditions

If there are multiple conditions, a `switch` statement is better.

```

function TrafficLight({ signal }) {
  switch (signal) {
    case "red":
      return <h1>🛑 Stop</h1>;
    case "yellow":
      return <h1>⚠️ Get Ready</h1>;
    case "green":
      return <h1>✅ Go</h1>;
    default:
      return <h1>❓ Unknown Signal</h1>;
  }
}

export default function App() {
  return <TrafficLight signal="green" />;
}

```

★ What Happens?

- If `signal="red"`, it shows "🛑 Stop".
- If `signal="yellow"`, it shows "⚠️ Get Ready".
- If `signal="green"`, it shows "✅ Go".

🔗 Real-Time Example: Online Shopping Stock Availability

```
function Product({ name, inStock }) {
  return (
    <div>
      <h2>{name}</h2>
      {inStock ? <p>✅ In Stock</p> : <p>❌ Out of Stock</p>}
    </div>
  );
}

export default function App() {
  return (
    <div>
      <Product name="Samsung Galaxy S24" inStock={true} />
      <Product name="iPhone 15" inStock={false} />
    </div>
  );
}
```

★ What Happens?

- If `inStock=true`, it shows ✅ In Stock.
- If `false`, it shows ❌ Out of Stock.

🔗 Summary of Conditional Rendering Methods

Method	Use Case
if statement	When you have a larger conditional block.
Ternary ? :	When you need to show one of two options .
&& (Short-circuit)	When you only show content if the condition is <code>true</code> .
switch statement	When there are multiple conditions to check.

★ Routing in React (React Router Explained)

Routing in React is handled using **React Router**, which allows navigation between different pages (components) without reloading the entire app.

◆ 1. Install React Router

Before using routing, install **React Router** using:

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

◆ 2. Basic Example with `react-router-dom`

Let's create a simple **React Router setup**.

✓ **App.js (Main Component)**

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

export default function App() {
  return (
    <Router>
      <nav>
        <ul>
          <li><Link to="/">🏠 Home</Link></li>
          <li><Link to="/about">📄 About</Link></li>
          <li><Link to="/contact">📞 Contact</Link></li>
        </ul>
      </nav>

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

◆ 3. Create Separate Page Components

✓ **Home.js**

```
import React from "react";

export default function Home() {
  return <h1>🏠 Welcome to Home Page</h1>;
}
```

✓ **About.js**

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

```
export default function About() {
  return <h1>📄 About Us Page</h1>;
}
```

✓ **Contact.js**

```
import React from "react";

export default function Contact() {
  return <h1>📞 Contact Us Page</h1>;
}
```

🔑 4. Explanation of Key Elements

Feature	Explanation
<Router>	Wraps the entire app and enables routing.
<Routes>	Contains all the routes (pages).
<Route path="/" element={<Home />} />	Defines a route (URL / will load Home.js).
<Link to="/about">About</Link>	Replaces <a href> to navigate without page reload.

🔑 5. Redirecting and 404 Page

✓ **Redirect Example**

If a user visits an unknown route, we can redirect them to a **404 Page**.

✓ **Modify App.js**

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

export default function App() {
  return (
    <Router>
      <nav>
        <ul>
          <li><Link to="/">🏠 Home</Link></li>
          <li><Link to="/about">📄 About</Link></li>
          <li><Link to="/contact">📞 Contact</Link></li>
        </ul>
      </nav>
    </Router>
  );
}
```

```

    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
      <Route path="/contact" element={<Contact />} />
      <Route path="*" element={<NotFound />} /> { /* 404 Page */}
    </Routes>
  </Router>
);
}

```

✓ Create NotFound.js

```

import React from "react";

export default function NotFound() {
  return <h1>✖ 404 - Page Not Found</h1>;
}

```

★ Now, if a user enters an unknown route like /random, it will show "404 - Page Not Found".

🔗 6. Nested Routes Example

Sometimes, we need **nested pages** (e.g., /products and /products/laptop).

✓ Modify App.js

```

import { BrowserRouter as Router, Routes, Route, Link } from "react-router-dom";
import Home from "./Home";
import Products from "./Products";
import Laptop from "./Laptop";

export default function App() {
  return (
    <Router>
      <nav>
        <ul>
          <li><Link to="/">🏠 Home</Link></li>
          <li><Link to="/products">📦 Products</Link></li>
        </ul>
      </nav>

      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/products" element={<Products />}>
          <Route path="laptop" element={<Laptop />} /> { /* Nested Route */}
        </Route>
      </Routes>
    </Router>
  );
}

```


✔ Create Products.js

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

export default function Products() {
  return (
    <div>
      <h1>📦 Products Page</h1>
      <Link to="/products/laptop">View Laptops</Link>
      <Outlet /> { /* Renders nested routes */ }
    </div>
  );
}
```

✔ Create Laptop.js

```
export default function Laptop() {
  return <h1>📁 Laptop Products</h1>;
}
```

★ Now, visiting `/products` shows the "Products Page". Clicking "View Laptops" loads `/products/laptop`.

💡 7. Programmatic Navigation (`useNavigate`)

Instead of using `<Link>`, we can **redirect users dynamically**.

✔ Example Using `useNavigate`

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

export default function Home() {
  const navigate = useNavigate();

  return (
    <div>
      <h1>🏠 Welcome to Home</h1>
      <button onClick={() => navigate("/about")}>Go to About</button>
    </div>
  );
}
```

★ Clicking the button **redirects to the About page** (`/about`).

🔗 Summary of React Router Features

Feature	Explanation
BrowserRouter	Enables routing in React apps.

Feature	Explanation
Routes	Holds multiple <code>Route</code> components.
<code>Route path="/" element={<Component />}</code>	Defines a specific route.
<code>Link to="/about"</code>	Navigates between pages without refreshing.
<code>useNavigate()</code>	Redirects programmatically.
<code>Route path="*" element={<Component />}</code>	Handles unknown routes (404 page).
Outlet	Used for rendering nested routes .

🔗 React Router Example: Indian Food Delivery App IN

Let's build a **basic food delivery app** with **React Router**. Users can navigate between **Home, Menu, Order Details, and Cart**.

🔗 1. Install React Router

Run the following command:

```
sh
CopyEdit
npm install react-router-dom
```

🔗 2. Folder Structure

```
/src
├── /components
│   ├── Navbar.js
│   ├── Home.js
│   ├── Menu.js
│   ├── Cart.js
│   ├── Order.js
│   └── NotFound.js
├── App.js
└── index.js
```

🔗 3. Create App.js (Main Component)

```
import React from "react";
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import Navbar from "./components/Navbar";
import Home from "./components/Home";
import Menu from "./components/Menu";
import Cart from "./components/Cart";
import Order from "./components/Order";
import NotFound from "./components/NotFound";
```

```
export default function App() {
  return (
    <Router>
      <Navbar />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/menu" element={<Menu />} />
        <Route path="/cart" element={<Cart />} />
        <Route path="/order" element={<Order />} />
        <Route path="*" element={<NotFound />} />
      </Routes>
    </Router>
  );
}
```

★ What This Does:

- Uses `BrowserRouter` for routing.
- Shows `Navbar` on all pages.
- Uses `Routes` to define paths (`/`, `/menu`, `/cart`, `/order`).
- Shows `NotFound` for unknown routes.

🔗 4. Create `Navbar.js` (Navigation)

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

export default function Navbar() {
  return (
    <nav>
      <ul>
        <li><Link to="/">🏠 Home</Link></li>
        <li><Link to="/menu">🍽 Menu</Link></li>
        <li><Link to="/cart">🛒 Cart</Link></li>
        <li><Link to="/order">📦 Order</Link></li>
      </ul>
    </nav>
  );
}
```

★ What This Does:

- Uses `<Link>` instead of `<a>` for smooth navigation.
- Navigates between **Home, Menu, Cart, and Order** pages.

🔗 5. Create `Home.js`

```
import React from "react";

export default function Home() {
  return (
    <div>
      <h1>🏠 Welcome to Indian Delights 🍽</h1>
      <p>Order delicious Indian food delivered to your home!</p>
    </div>
  );
}
```

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

🔗 6. Create Menu.js (Food Items)

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

const menuItems = [
  { id: 1, name: "Paneer Butter Masala", price: 250 },
  { id: 2, name: "Chicken Biryani", price: 300 },
  { id: 3, name: "Masala Dosa", price: 150 },
  { id: 4, name: "Dal Tadka", price: 180 },
];

export default function Menu() {
  return (
    <div>
      <h1>🍽️ Our Menu</h1>
      <ul>
        {menuItems.map((item) => (
          <li key={item.id}>
            {item.name} - ₹{item.price}{" "}
            <Link to="/cart">🛒 Add to Cart</Link>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

🔗 What This Does:

- Displays a **list of Indian dishes** with prices.
- Each dish has an **"Add to Cart"** button.

🔗 7. Create Cart.js

```
import React from "react";
import { useNavigate } from "react-router-dom";

export default function Cart() {
  const navigate = useNavigate();

  return (
    <div>
      <h1>🛒 Your Cart</h1>
      <p>Items in your cart will appear here.</p>
      <button onClick={() => navigate("/order")}>📦 Place Order</button>
    </div>
  );
}
```

★ What This Does:

- Displays the **Cart page**.
- Uses `useNavigate()` to **redirect** users to the Order page.

🔗 8. Create `Order.js`

```
import React from "react";

export default function Order() {
  return (
    <div>
      <h1>🍽️ Order Placed Successfully! 🎉</h1>
      <p>Your delicious food is on the way. 🚚</p>
    </div>
  );
}
```

★ What This Does:

- Shows a **success message** after placing an order.

🔗 9. Create `NotFound.js` (404 Page)

```
jsx
CopyEdit
import React from "react";

export default function NotFound() {
  return <h1>❌ 404 - Page Not Found</h1>;
}
```

★ What This Does:






- Shows **404 error** for unknown URLs.

🔗 10. Run the App

Start the React app:

🚀 Final Features of the App

Feature	Description
🏠 Home Page	Shows app introduction.
🍽️ Menu Page	Displays a list of Indian dishes.

Feature	Description
 Cart Page	Users can view selected items.
 Order Page	Shows order confirmation.
 404 Page	Handles invalid routes.
 Link Navigation	Uses <code><Link></code> for fast transitions.
 <code>useNavigate()</code>	Redirects users programmatically.

✦ Steps Covered:

1. **Setup Express & MongoDB**
2. **Create Signup & Login Routes**
3. **Hash Passwords with `bcryptjs`**
4. **Generate JWT Token for Authentication**
5. **Store User Data in MongoDB**

✦ 1. Install Dependencies

Run the following command in your project folder:

```
npm init -y
npm install express mongoose bcryptjs jsonwebtoken dotenv cors body-parser
```

✦ 2. Create `server.js`

This is the main Express app file.

```
require('dotenv').config();
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const bodyParser = require('body-parser');
const authRoutes = require('./routes/authRoutes');

const app = express();

// Middleware
app.use(cors());
app.use(bodyParser.json());

// MongoDB Connection
mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log("MongoDB Connected"))
```

```
    .catch(err => console.error(err));

// Routes
app.use('/auth', authRoutes);

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

✦ 3. Create `.env` File

This file stores your MongoDB connection string and JWT secret.

```
MONGO_URI=mongodb://localhost:27017/authdb
JWT_SECRET=mysecretkey
```

✦ 4. Create `models/User.js`

This defines the User schema for MongoDB.

```
const mongoose = require('mongoose');

const UserSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true }
});

module.exports = mongoose.model('User', UserSchema);
```

✦ 5. Create `routes/authRoutes.js`

This file contains **Signup** and **Login** routes.

```
const express = require('express');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const User = require('../models/User');

const router = express.Router();

// Signup Route
router.post('/signup', async (req, res) => {
  try {
    const { name, email, password } = req.body;

    // Check if user already exists
    const existingUser = await User.findOne({ email });
    if (existingUser) return res.status(400).json({ message: "User already exists" });

    // Hash the password
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(password, salt);
```

```

        // Save user in DB
        const newUser = new User({ name, email, password: hashedPassword
    });

    await newUser.save();

    res.status(201).json({ message: "User registered successfully" });

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

// Login Route
router.post('/login', async (req, res) => {
    try {
        const { email, password } = req.body;

        // Find user in DB
        const user = await User.findOne({ email });
        if (!user) return res.status(400).json({ message: "User not found"
    });

        // Compare password
        const isMatch = await bcrypt.compare(password, user.password);
        if (!isMatch) return res.status(400).json({ message: "Invalid
credentials" });

        // Generate JWT token
        const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, {
expiresIn: "1h" });

        res.status(200).json({ token });

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

module.exports = router;

```

✦ 6. Run the Server

Start the Express server using:

```
node server.js
```

or if using Nodemon:

```
npx nodemon server.js
```

✦ 7. Testing API with Postman

✓ Sign Up API (POST /auth/signup)

- **Request Body:**


```
{
  "name": "Rahul Sharma",
  "email": "rahul@example.com",
  "password": "123456"
}
```

- **Response:**

```
{
  "message": "User registered successfully"
}
```

✓ Login API (POST /auth/login)

- **Request Body:**

```
{
  "email": "rahul@example.com",
  "password": "123456"
}
```

- **Response:**

```
{
  "token": "eyJhbGciOiJIUz..."
}
```

✦ 8. Using JWT for Protected Routes

If you want to **protect routes** (only accessible with a valid token), create **middleware**:

Create `middleware/authMiddleware.js`

```
const jwt = require('jsonwebtoken');

const authMiddleware = (req, res, next) => {
  const token = req.header("Authorization"); // Get token from header

  if (!token) return res.status(401).json({ message: "Access Denied" });

  try {
    const verified = jwt.verify(token, process.env.JWT_SECRET); //
    Verify token
    req.user = verified; // Attach user info to request
    next(); // Continue to the next middleware or route handler
  } catch (error) {
    res.status(400).json({ message: "Invalid Token" });
  }
};

module.exports = authMiddleware; // Export the function
```

★ Apply Middleware to Protected Route (`authRoutes.js`)

Now, use `authMiddleware` in your **protected routes**:

```
const authMiddleware = require('../middleware/authMiddleware');

// Example of a protected route
router.get('/dashboard', authMiddleware, (req, res) => {
  res.json({ message: "Welcome to Dashboard", user: req.user });
});
```

★ Forms in React

Forms in React are slightly different from regular HTML forms because React handles form elements using **controlled components**. This means form inputs are controlled by React state.

1 Controlled Components

In a controlled component, the form elements (like `<input>`, `<textarea>`, `<select>`) are controlled by React state.

Example: Controlled Input

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

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

  const handleChange = (event) => {
    setName(event.target.value); // Updating state on every keystroke
  };

  const handleSubmit = (event) => {
    event.preventDefault();
    alert("Submitted Name: " + name);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input type="text" value={name} onChange={handleChange} />
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}

export default ControlledForm;
```

★ Key Takeaways:

- The value of `<input>` is tied to name state.
- `onChange` updates state on user input.

2 Handling Multiple Inputs

For forms with multiple fields, use an object in state.

Example: Multi-field Form

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

function 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;
```

★ Key Takeaways:

- The `[event.target.name]` dynamically updates the state based on input field names.

3 Handling Select, Checkbox, and Radio Inputs

Dropdown (`<select>`)

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

  return (
    <form>
      <label>
        Pick a fruit:
        <select value={fruit} onChange={(e) => setFruit(e.target.value)}>
          <option value="apple">Apple</option>
        </select>
      </label>
    </form>
  );
}
```

```

        <option value="banana">Banana</option>
        <option value="mango">Mango</option>
      </select>
    </label>
    <p>Selected: {fruit}</p>
  </form>
);
}

```

Checkbox

```

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

  return (
    <form>
      <label>
        Accept Terms:
        <input type="checkbox" checked={isChecked} onChange={() =>
setIsChecked(!isChecked)} />
      </label>
      <p>{isChecked ? "Accepted" : "Not Accepted"}</p>
    </form>
  );
}

```

Radio Buttons

```

function RadioForm() {
  const [gender, setGender] = useState("");

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

```

4 Uncontrolled Components

In uncontrolled components, the form data is accessed using **refs**, rather than state.

Example: Uncontrolled Input using `useRef`

```

import { useRef } from "react";

function UncontrolledForm() {
  const inputRef = useRef(null);

```

```

const handleSubmit = (event) => {
  event.preventDefault();
  alert("Input Value: " + inputRef.current.value);
};

return (
  <form onSubmit={handleSubmit}>
    <input type="text" ref={inputRef} placeholder="Enter something..." />
    <button type="submit">Submit</button>
  </form>
);
}

```

★ When to use?

Uncontrolled components are useful for non-React libraries (e.g., integrating with third-party forms).

5 Form Validation

Example: Basic Validation

```

function FormValidation() {
  const [email, setEmail] = useState("");
  const [error, setError] = useState("");

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

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

```

6 Handling File Uploads

```

function FileUpload() {
  const [file, setFile] = useState(null);

  const handleFileChange = (event) => {
    setFile(event.target.files[0]);
  };

  const handleSubmit = (event) => {
    event.preventDefault();

```

```

    alert(`File Selected: ${file ? file.name : "No file"}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="file" onChange={handleFileChange} />
      <button type="submit">Upload</button>
    </form>
  );
}

```

7 Submitting Forms Using `fetch()`

After handling the form, you often need to send data to a backend.

Example: Submit to API

```

function SubmitToAPI() {
  const [name, setName] = useState("");

  const handleSubmit = async (event) => {
    event.preventDefault();
    const response = await fetch("https://api.example.com/submit", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ name }),
    });
    const result = await response.json();
    alert("Response: " + result.message);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={name} onChange={(e) =>
setName(e.target.value)} placeholder="Enter Name" />
      <button type="submit">Submit</button>
    </form>
  );
}

```

Summary:

Feature	Example Hook Used
Controlled Inputs	<code>useState</code>
Multiple Fields	<code>useState</code> with an object
Checkbox, Select, Radio	<code>useState</code>
Uncontrolled Form	<code>useRef</code>
Form Validation	<code>useState</code> and basic logic

Feature	Example Hook Used
File Upload	useState and onChange
API Submission	fetch()

🔗 useMemo Hook in React

The `useMemo` hook is used for **performance optimization** in React. It **memoizes** the result of a computation and **only recalculates it when dependencies change**, reducing unnecessary re-renders.

◆ Syntax

```
const memoizedValue = useMemo(() => computeExpensiveValue(dep1, dep2),
[dep1, dep2]);
```

- ✓ **First argument:** A function that returns the computed value.
- ✓ **Second argument:** Dependency array (`[dep1, dep2]`). The function runs **only if dependencies change**.
- ✓ Returns the **memoized value**.

🔗 Example: Without `useMemo` (Recalculates Every Render)

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

function ExpensiveCalculation() {
  const [count, setCount] = useState(0);
  const [items, setItems] = useState([1, 2, 3]);

  // Expensive computation (runs on every render)
  const sum = items.reduce((acc, num) => {
    console.log("Calculating sum...");
    return acc + num;
  }, 0);

  return (
    <div>
      <h3>Count: {count}</h3>
      <h3>Sum: {sum}</h3>
      <button onClick={() => setCount(count + 1)}>Increase Count</button>
      <button onClick={() => setItems([...items, items.length + 1]}>Add
Number</button>
    </div>
  );
}

export default ExpensiveCalculation;
```

● Issue:

- The sum calculation runs **every time** count updates, which is unnecessary.

🔗 Optimized with `useMemo`

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

function ExpensiveCalculationMemoized() {
  const [count, setCount] = useState(0);
  const [items, setItems] = useState([1, 2, 3]);

  // Memoizing the sum calculation
  const sum = useMemo(() => {
    console.log("Calculating sum...");
    return items.reduce((acc, num) => acc + num, 0);
  }, [items]); // Runs only when 'items' changes

  return (
    <div>
      <h3>Count: {count}</h3>
      <h3>Sum: {sum}</h3>
      <button onClick={() => setCount(count + 1)}>Increase Count</button>
      <button onClick={() => setItems([...items, items.length + 1]}>Add
Number</button>
    </div>
  );
}

export default ExpensiveCalculationMemoized;
```

✓ Benefits of `useMemo`

- Now, sum **only recalculates when items change**, avoiding unnecessary re-renders when updating count.
- Improves **performance** for expensive computations.

🔗 `useMemo` with Filtering

It is useful for filtering large lists without recalculating on every render.

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

function FilterList() {
  const [query, setQuery] = useState("");
  const items = ["Apple", "Banana", "Mango", "Orange", "Grapes"];

  // Memoized filtered list
  const filteredItems = useMemo(() => {
    console.log("Filtering items...");
    return items.filter((item) =>
item.toLowerCase().includes(query.toLowerCase()));
  }, [query]); // Runs only when 'query' changes
```



```

    return (
      <div>
        <input type="text" placeholder="Search..." value={query}
onChange={ (e) => setQuery(e.target.value)} />
        <ul>
          {filteredItems.map((item, index) => (
            <li key={index}>{item}</li>
          ))}
        </ul>
      </div>
    );
  }
}

export default FilterList;

```

✓ Benefit:

- The filter function **only runs when the user types** instead of re-running on every render.

💡 When to Use `useMemo`?

✓ Use `useMemo` for **expensive computations**, such as:

- Heavy calculations** (e.g., sorting, filtering, mathematical operations).
- Derived state** (e.g., extracting values from large datasets).
- Preventing unnecessary renders** of child components.

⊗ Don't use `useMemo` unnecessarily!

- If the computation is **fast**, avoid `useMemo`, as it adds complexity.

💡 Summary

Feature	Without <code>useMemo</code>	With <code>useMemo</code>
Performance	Slower (Recalculates Every Render)	Faster (Only Recalculates When Dependencies Change)
Usage	Every time component renders	Only when dependencies update
Best For	Expensive Computations	Optimizing Performance

💡 `useMemo` VS. `useEffect` – When to Use What?

Feature	<code>useMemo</code>	<code>useEffect</code>
Purpose	Memoizes a value (avoids recalculating it on every render)	Runs side effects (e.g., API calls, DOM manipulations, subscriptions)

Feature	<code>useMemo</code>	<code>useEffect</code>
Returns	A memoized value	Nothing (you handle side effects inside)
Executes When	Dependencies change	Component renders + dependencies change
Best For	Performance optimizations (e.g., expensive calculations, filtering, sorting)	Side effects (e.g., fetching data, event listeners, subscriptions)

`useContext` in React

◆ What is `useContext`?

- `useContext` is a **React Hook** that allows **components to consume context** without passing props down manually (prop drilling).
- It helps **share state** between components efficiently.

◆ How to Use `useContext`?

✓ 1. Create a Context

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

// Create a Context
const ThemeContext = createContext("light");

export default ThemeContext;
```

✓ 2. Provide Context Value

Wrap your components inside a **Provider** to share state.

```
import ThemeContext from "../ThemeContext";

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <ChildComponent />
    </ThemeContext.Provider>
  );
}

export default App;
```

✓ 3. Consume Context using `useContext`

Now, any child component can **access the context value** without prop drilling.

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

function ChildComponent() {
  const theme = useContext(ThemeContext); // ✓ Access context value

  return <h1>The current theme is {theme}</h1>;
}

export default ChildComponent;
```

◆ When to Use `useContext`?

- ✓ **Avoid Prop Drilling** – If a value needs to be passed to multiple nested components.
 - ✓ **Global State Management** – Example: Theme, Authentication, User Data, etc.
-

◆ Real-world Example: Theme Toggle using `useContext`

✦ Step 1: Create a `ThemeContext`

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

const ThemeContext = createContext();

export const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState("light");

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};

export default ThemeContext;
```

✦ Step 2: Wrap App with `ThemeProvider`

```
import { ThemeProvider } from "../ThemeContext";
import ThemeSwitcher from "../ThemeSwitcher";

function App() {
  return (
    <ThemeProvider>
      <ThemeSwitcher />
    </ThemeProvider>
  );
}

export default App;
```

✦ Step 3: Use `useContext` to Toggle Theme

```

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

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

  return (
    <div>
      <h2>Current Theme: {theme}</h2>
      <button onClick={() => setTheme(theme === "light" ? "dark" :
"light")}>
        Toggle Theme
      </button>
    </div>
  );
}

export default ThemeSwitcher;

```

🔗 [useContext VS. useState VS. Redux](#)

Hook	Use Case
useState	Local component state (e.g., input fields, toggles)
useContext	Global state shared across multiple components (e.g., theme, auth)
Redux / Zustand	Large-scale state management with advanced features