ReactJs Intermediate

useEffect

1. `useEffect` kya hai?

- **useEffect = hook for side effects**
- Side effect = **wo kaam jo component ke render ke bahar hota hai ya UI ko directly render nahi karta**
- Examples:
  - API call (fetch data from server)
  - Subscriptions (WebSocket, EventListener)
  - Timer (setInterval, setTimeout)
  - Manual DOM manipulation
  - Logging

> Simple: useEffect = React ka magic room jahan tum side tasks kar sakte ho without breaking UI render

---

2. Syntax

```
useEffect(() => {
  // side effect code here

  return () => {
    // cleanup code here (optional)
  };
}, [dependencies]);
```

- First argument → **function** → side effect code
- Return function → **cleanup** (optional) → component unmount ya effect rerun pe run hota hai

- Dependencies array → **effect kab run hoga**

3. Basic Example: Component Did Mount

```
import React, { useEffect } from 'react';

function Hello() {
  useEffect(() ⇒ {
    console.log("Component mounted"); // runs once
  }, []); // empty array = run only once

  return <h1>Hello World</h1>;
}
```

- Empty array `[]` → **componentDidMount equivalent**
- Side effect sirf **first render pe** run hota hai

4. useEffect with Dependencies

```
import React, { useState, useEffect } from 'react';

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

  useEffect(() ⇒ {
    console.log("Count changed:", count);
  }, [count]); // run whenever count changes

  return <button onClick={() ⇒ setCount(count + 1)}>Increment</button>;
}
```

- `count` change → effect run
- Multiple dependencies → `[count, name, isLoggedIn]`

5. Cleanup Function

- Necessary for **subscriptions, timers, event listeners**
- Prevent memory leaks

```
useEffect(() => {
  const timer = setInterval(() => {
    console.log("Tick");
  }, 1000);

  return () => {
    clearInterval(timer); // cleanup
    console.log("Timer cleared");
  };
}, []);
```

- Component unmount → cleanup run
- Component re-render with new dependencies → cleanup old effect → run new effect

---

6. Real-world Example: API Call

```
import React, { useState, useEffect } from 'react';

function Users() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then((res) => res.json())
      .then((data) => {
        setUsers(data);
        setLoading(false);
      })
      .catch((err) => console.log(err));
  }, []); // run once

  if (loading) return <p>Loading...</p>;
  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

- API call → fetch data → setState → re-render UI
- Empty deps → run once, component mount
- Cleanup not needed here

---

7. Common Patterns

1. **ComponentDidMount:** `useEffect(() => {...}, [])`
2. **ComponentDidUpdate:** `useEffect(() => {...}, [dep])`
3. **ComponentWillUnmount / Cleanup:** return function in `useEffect`

Timers / Subscriptions / Event Listeners → always cleanup

---

8. MeriAnalogy

Socho: **component = kitchen**

- Render = kitchen ready for cooking
- useEffect = **side tasks like fetching ingredients, preheating oven, cleaning utensils**
- Dependencies = "tabhi ye kaam karo jab ingredient list change ho"
- Cleanup = "kaam khatam → clean utensils, stop stove"
- Result = kitchen **organized + efficient** + no wastage

---

9. Key Takeaways

1. useEffect = **side effects handler** in functional components
2. Always **think about dependencies** → prevent infinite loops

3. Cleanup is **mandatory** for timers, subscriptions
4. Can replace **componentDidMount / componentDidUpdate / componentWillUnmount**
5. Common uses: API calls, logging, timers, DOM events
6. Dependencies array → empty = run once, specific values = run on change

useRef

1. `useRef` kya hai?

- `useRef` = hook jo **mutable reference** create karta hai
- Do main use cases:
    1. **Access DOM elements directly**
    2. **Store mutable values** without causing re-render

> Simple shabdon me: useRef = React ka bookmark / storage box, jisme tum data rakh sakte ho ya DOM ko point kar sakte ho.

---

2. Syntax

```
const refContainer = useRef(initialValue);
```

- `refContainer.current` → access ya modify karne ka property
- `initialValue` → shuru me kya value chahiye

---

3. Accessing DOM Example

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

function FocusInput() {
  const inputRef = useRef(null);
  const handleClick = () ⇒ {
    inputRef.current.focus(); // DOM element pe directly kaam
  };

  return (
    <div>
      <input ref={inputRef} type="text" placeholder="Type here..." />
      <button onClick={handleClick}>Focus Input</button>
    </div>
  );
}
```

- `ref={inputRef}` → input element ko point karta hai
- Button click → input pe focus → React ka **direct DOM manipulation**

---

4. Storing Values Without Re-render

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

function Timer() {
  const [count, setCount] = useState(0);
  const intervalRef = useRef(null);

  const start = () ⇒ {
    intervalRef.current = setInterval(() ⇒ {
      setCount((prev) ⇒ prev + 1);
    }, 1000);
  };

  const stop = () ⇒ {
    clearInterval(intervalRef.current);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={start}>Start</button>
      <button onClick={stop}>Stop</button>
    </div>
  );
}
```

- `intervalRef.current` → mutable storage
- Value change → **UI re-render nahi hota** unless state update ho

---

5. Difference between `useState` & `useRef`

Feature	useState	useRef
Causes re-render?	Yes	No
Stores value across renders?	Yes	Yes
Direct DOM access?	No	Yes
Mutable without re-render?	No	Yes

> useRef = perfect for timers, previous value tracking, DOM focus

---

6. Track Previous State Example

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

function PrevCounter() {
  const [count, setCount] = useState(0);
  const prevCountRef = useRef(0);

  useEffect(() => {
    prevCountRef.current = count; // update previous value after render
  }, [count]);

  return (
    <div>
      <p>Current: {count}</p>
      <p>Previous: {prevCountRef.current}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

- `prevCountRef.current` = previous count
- UI re-render → previous value tracked **without extra state**

---

7. Meri Analogy

- Soch **useState = TV ka display** → har change → screen update
- **useRef = remote storage box / sticky note**
  - Tumne kuch likha → reference hamesha available
  - Change karoge → display update nahi hota automatically
- DOM ka example → "pointer to TV channel button" → directly focus/press kar sakte ho

---

8. Key Takeaways

1. `useRef` = mutable reference in functional components
2. Main uses:
   - **Access DOM directly** (`ref={...}`)
   - **Store values without re-render** (`intervals, previous values`)
3. `ref.current` = read/write access
4. State vs Ref → Ref **does not trigger re-render**, State triggers
5. Perfect for timers, previous state, input focus, third-party libraries

useMemo and useCallback

1. Kya problem hai?

- React me **functional components** har render me **poora function run karte hain**
- Agar component me **heavy calculation** ya **child components passing functions** ho → performance slow ho jati hai
- Solution = **memoization** → React ko bolo: "bhai, agar values change nahi hui → phir se calculate mat kar"

---

2. `useMemo` kya hai?

- `useMemo` = hook for **memoizing values**
- Heavy calculations ko **bina re-calculating** reuse karna
- Syntax:

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a,
b]);
```

- `[a, b]` → dependencies
- Agar dependencies change nahi → previous result reuse hota hai

---

Example: Heavy Calculation

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

function HeavyCalc() {
  const [count, setCount] = useState(0);
  const [other, setOther] = useState(0);

  const factorial = (n) => {
    console.log("Calculating factorial...");
    let result = 1;
    for (let i = 1; i <= n; i++) result *= i;
    return result;
  };

  const fact = useMemo(() => factorial(count), [count]);
  return (
    <div>
      <p>Factorial of {count} is {fact}</p>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
      <button onClick={() => setOther(other + 1)}>Increment Other</button>
    </div>
  );
}
```

- **Console log check** → factorial **sirf count change pe calculate hota hai**, `other` change pe nahi
- **Bina useMemo** → har render me factorial recalc → slow

---

3. `useCallback` kya hai?

- `useCallback` = hook for **memoizing functions**
- Useful jab **function child component me props ke through pass** ho
- Prevents **unnecessary re-renders**

```
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

- Function sirf **dependencies change hone par create hota hai**
- Child component me referential equality maintained → React memoization work kare

---

Example: Child Component Re-render

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

const Child = React.memo(({ onClick }) => {
  console.log("Child rendered");
  return <button onClick={onClick}>Click Me</button>;
});
```

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

  const handleClick = useCallback(() => {
    console.log("Button clicked");
  }, []); // function never changes

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

- Without `useCallback` → **Child re-renders on every parent render**
- With `useCallback` → **Child only re-renders if dependencies change**

---

4. Meri Analogy

- Socho: **component = factory**
- Heavy calculation = factory me **machinery ka heavy work**
- useMemo = “bhai, agar raw materials same hai → previous product use kar lo, dobara mat banayo”
- useCallback = “worker ko baar-baar nahi bulana, sirf jab kaam ka input change ho”
- Result → **fast, efficient, smooth production**

---

5. Key Takeaways

1. **useMemo** → memoize **values** / heavy calculations
2. **useCallback** → memoize **functions** / prevent unnecessary child renders
3. Dependencies array → **must** → React ko pata chale **kab recalc ya recreate karna hai**

4. Use in **performance-critical apps**
5. Combine with **React.memo** → perfect optimization

Forms

1. Forms ka funda

- **Forms = user input lene ka tarika**
- React me forms handle karna thoda alag hai → **controlled vs uncontrolled components**
- Controlled → React state me input ka value track hota hai
- Uncontrolled → DOM pe direct value read karte hain (`ref` ke through)

> Simple: controlled forms = React ka boss mode

---

2. Controlled Components

- Input ka value React **state se control hota hai**
- Har change → state update → UI re-render

Example: Text Input

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

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

  const handleChange = (e) => {
    setName(e.target.value);
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    alert(`Hello, ${name}`);
  };
  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={name} onChange={handleChange} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Flow:

1. User type → `onChange` → state update
2. Submit → `onSubmit` → prevent default → alert show

---

3. Multiple Inputs

```
function MultiForm() {
  const [form, setForm] = useState({ email: "", password: "" });

  const handleChange = (e) => {
    setForm({ ...form, [e.target.name]: e.target.value });
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log(form);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input name="email" value={form.email} onChange={handleChange} pl
aceholder="Email" />
      <input name="password" value={form.password} onChange={handleC
hange} placeholder="Password" />
      <button type="submit">Submit</button>
    </form>
  );
}
```

- **Dynamic form handling** using `name` attribute
- Spread operator → previous state preserve

---

4. Checkboxes & Radio Buttons

```
function Preferences() {
  const [checked, setChecked] = useState(false);

  return (
    <div>
      <label>
        <inputtype="checkbox"
          checked={checked}
          onChange={(e) ⇒ setChecked(e.target.checked)}
        />
        Accept Terms
      </label>
      <p>{checked ? "Accepted" : "Not Accepted"}</p>
    </div>
  );
}
```

- Checkbox → `checked` attribute
- Radio → similar, value + state

---

5. Select Dropdown

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

  return (
    <div>
      <select value={fruit} onChange={(e) ⇒ setFruit(e.target.value)}>
        <option value="">Select Fruit</option>
        <option value="apple">Apple</option>
        <option value="mango">Mango</option>
      </select>
      <p>Selected: {fruit}</p>
    </div>
  );
}
```

- Dropdown → `value` controlled by state
- onChange → update state

---

6. Uncontrolled Components (Optional)

- Input value **DOM pe stored**
- `ref` ke through read

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

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

  const handleSubmit = (e) => {
    e.preventDefault();
    alert(inputRef.current.value);
  };

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

- Less React control → simpler but **not recommended for complex forms**

---

7. Meri Analogy

- Socho: **form = kitchen order slip**

- Controlled → React chef knows **exactly kya ingredients user ne choose kiya**, har change track karta hai
- Uncontrolled → chef bas slip dekhke kaam karta hai, React ko pata nahi → inefficient
- Checkbox, radio, select → ingredients ke options
- Submit → React confirms order and processes

---

8. Key Takeaways

1. Forms handle karne ka React way → **controlled components**
2. `value` = state, `onChange` = update state
3. `onSubmit` → prevent default → process data
4. Multiple inputs → use `name` attribute + dynamic state update
5. Checkbox, radio, select → controlled by state
6. Uncontrolled → `ref`, optional, simple forms
7. Forms + state = **reactive + interactive UI**

Controlled and Uncontrolled Components

1. Controlled Components

- **Controlled = React state se input value control hoti hai**
- `value` attribute → React state
- `onChange` → state update
- React **full control** rakhta hai

Example: Controlled Input

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

function ControlledInput() {
  const [name, setName] = useState("");
  return (
    <div>
      <inputtype="text"
        value={name}          // controlled by React state
        onChange={(e) => setName(e.target.value)}  // update state
        placeholder="Enter name"
      />
      <p>Hello, {name}</p>
    </div>
  );
}
```

Flow:

1. User type → `onChange` → state update
2. State change → UI re-render → input value update
3. React ke paas **100% control**

Pros:

- Validation easy
- Predictable
- Easy to reset / prefill forms

Cons:

- Boilerplate code → extra `onChange` handlers

---

2. Uncontrolled Components

- **Uncontrolled = DOM itself manages value**
- React ke paas **direct control nahi**
- Value access → `ref` ke through

Example: Uncontrolled Input

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

function UncontrolledInput() {
  const inputRef = useRef();

  const handleSubmit = () => {
    alert(`Hello, ${inputRef.current.value}`); // direct DOM access
  };

  return (
    <div>
      <input type="text" ref={inputRef} placeholder="Enter name" />
      <button onClick={handleSubmit}>Submit</button>
    </div>
  );
}
```

Flow:

1. User type → DOM stores value
2. Submit → React reads value via `ref`
3. React **state unaware**

Pros:

- Less code, simple forms
- Quick for small inputs

Cons:

- Hard to validate dynamically
- Hard to reset / prefill
- Not recommended for complex forms

---

3. Key Differences

Feature	Controlled	Uncontrolled
Value controlled by	React state	DOM
Updates via	onChange → setState	ref access only
Validation	Easy, real-time	Hard, on submit only
Reset / Prefill	Easy	Hard

Feature	Controlled	Uncontrolled
Code complexity	More boilerplate	Less code
Use case	Complex forms, dynamic UI	Simple / one-off inputs

4. Meri Analogy

- **Controlled = React chef knows exact ingredients user ne choose kiye** → can adjust anytime
- **Uncontrolled = Chef bas slip dekhe → kaam karta hai** → React unaware of changes
- Complex orders = controlled → safe
- Simple single ingredient = uncontrolled → fast

5. Best Practice

- **Always prefer controlled components** for real-world apps
- Uncontrolled → only for **quick, small forms**
- Large forms → combine with **useState + validation + dynamic fields**

Passing Data without Props Drilling

1. Props Drilling ka Problem

- Props drilling = data ko **parent → child → grandchild → ...** pass karna
- Bahut deep components me → messy aur hard to maintain
- Example:

```
function App() {
  const user = "Harsh";
  return <Parent user={user} />;
}

function Parent({ user }) {
  return <Child user={user} />;
}

function Child({ user }) {
  return <GrandChild user={user} />;
}

function GrandChild({ user }) {
  return <p>Hello {user}</p>;
}
```

- Problem: **har level pe props pass karna**
- Agar tree bada hai → nightmare

---

2. Solution: React Context API

- **Context = global state for a subtree**
- Kisi bhi nested component → directly access data
- No props drilling

Steps:

1. **Create Context**

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

export const UserContext = createContext();
```

1. **Provide Context**

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

  return (
    <UserContext.Provider value={user}>
      <Parent />
    </UserContext.Provider>
  );
}
```

1. **Consume Context**

```
import React, { useContext } from "react";
import { UserContext } from "./UserContext";

function GrandChild() {
  const user = useContext(UserContext);
  return <p>Hello {user}</p>;
}
```

- **Parent aur Child ko props pass karne ki zarurat nahi**
- Deep tree → clean, maintainable

Dynamic Data Example

```
function App() {
  const [user, setUser] = React.useState("Harsh");

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

function GrandChild() {
  const { user, setUser } = useContext(UserContext);
  return (
    <div>
      <p>Hello {user}</p>
      <button onClick={() ⇒ setUser("Shweta")}>Change User</button>
    </div>
  );
}
```

- **Context can store state + updater function**
- Nested component → **directly update parent state**

4. Meri Analogy

- Socho: **props drilling = bucket brigade**
  - Water (data) pass karna har person → long chain
- **Context = water tank**
  - Sab directly access kar sakte hain → easy, fast, maintainable
- Nested kitchen = deep tree
- Each chef (component) can directly use the ingredient (data)

5. Key Takeaways

1. Props drilling → inefficient, messy for deep trees
2. Context API → provides **global-ish state for subtree**
3. `createContext()` → context create
4. `Provider` → supply value to subtree
5. `useContext()` → consume anywhere in subtree
6. Can store **values, objects, functions** → powerful for state management
7. Combine with **useReducer / state** → even more pro

Router

1. React Router kya hai?

- React SPA me **ek hi HTML page** hota hai
- Multiple pages → React Router ke through manage

- Browser URL change hota hai → React DOM ke andar component switch hota hai
- **No full page reload** → fast user experience

> Simple shabdon me: React Router = React ka GPS + traffic controller

---

2. Installation

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

- React Router DOM = browser-specific routing

---

3. Basic Setup

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

function Home() { return <h1>Home Page</h1>; }
function About() { return <h1>About Page</h1>; }
function Contact() { return <h1>Contact Page</h1>; }

function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link> |
        <Link to="/about">About</Link> |
        <Link to="/contact">Contact</Link>
      </nav>

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

Explanation:

1. `Router` → wrap the app → enables routing
2. `Link` → SPA navigation → no page reload
3. `Routes` → container for all routes
4. `Route` → path + element → which component render on which URL

---

4. Dynamic Routes (Params)

```
function User({ id }) {
  return <h1>User ID: {id}</h1>;
}

// Setup
<Route path="/user/:id" element={<UserWrapper />} />

function UserWrapper() {
  const { id } = useParams(); // hook to get URL param
  return <User id={id} />;
}
```

- `:id` → dynamic parameter
- `useParams()` → get params from URL

---

5. Nested Routes

```
function Dashboard() {
  return (
    <div>
      <h1>Dashboard</h1>
      <Outlet /> {/* nested component render here */}
    </div>
  );
}

function Stats() { return <p>Stats Page</p>; }
function Reports() { return <p>Reports Page</p>; }

<Routes>
  <Route path="/dashboard" element={<Dashboard />}>
    <Route path="stats" element={<Stats />} />
    <Route path="reports" element={<Reports />} />
  </Route>
</Routes>
```

- Parent → `<Dashboard />`
- Child → `<Outlet />` me render
- URL: `/dashboard/stats` → renders Stats inside Dashboard

---

6. Redirect / Navigation

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

function Login() {
  const navigate = useNavigate();

  const handleLogin = () => {
    // after login success
    navigate("/dashboard");
  };

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

- `useNavigate()` → programmatic navigation
- Redirect after action → common in auth flows

---

7. 404 / Not Found Page

```
<Route path="*" element={<h1>404 Not Found</h1>} />
```

- Any unmatched route → show 404
- Always place **last route**

---

8. Meri Analogy

- **React app = city**
- **Router = GPS system** → decide which road (component) to take based on URL
- **Link = signboards** → click → city doesn't reload, just redirect
- **Dynamic route = personalized address** → /user/101 → direct house
- **Nested route = building floors** → parent dashboard → render child inside
- **Navigate = cab driver** → programmatic redirection

---

9. Key Takeaways

1. React Router = SPA navigation engine
2. Components:
    - `BrowserRouter` → main router
    - `Routes` → container for all routes
    - `Route` → path + element
    - `Link` → client-side navigation
    - `useParams` → dynamic URL params
    - `useNavigate` → programmatic navigation
    - `Outlet` → nested routes placeholder
3. Always handle 404 → `path="*"`
4. Nested routes → maintain UI hierarchy
5. Avoid full page reload → SPA benefit

Single Page Apps (SPA)

1. SPA kya hai?

- SPA = Single Page Application
- **Ek hi HTML page** load hota hai
- User navigate kare → **browser reload nahi hota**, sirf content dynamically update hota hai
- React, Vue, Angular → SPA framework examples

> Simple: SPA = ek hi page + multiple views + dynamic rendering

---

2. Traditional Multi Page App (MPA) vs SPA

Feature	MPA	SPA
Page reload	Full reload on every navigation	No reload, just DOM update
Speed	Slow	Fast, instant navigation
Server load	Higher	Lower, more frontend heavy
URL routing	Server-side	Client-side (React Router)
State management	Harder	Easier, single page state

---

3. SPA ka Flow

1. Browser loads **index.html** once
2. React JS bundle load hota hai → render app
3. User clicks link → React **DOM me component switch**
4. Browser URL change hota hai → page reload nahi hota
5. Backend API → fetch data dynamically
6. React updates UI based on state / route

Flow diagram:

```
Browser → index.html → React App → Routes → Component → API (if need
ed) → Render
```

4. React + SPA

- React naturally SPA friendly hai
- React Router → client-side routing → multiple views without reload
- State + Context → manage global app data
- API calls → dynamically fetch data without refreshing

5. SPA Benefits

1. **Fast Navigation** → no full reload
2. **Better UX** → smooth, app-like feel
3. **Frontend-driven** → server load reduced
4. **Dynamic content** → API-driven, real-time
5. **Reusable components** → maintainable UI

6. SPA Challenges

- SEO → server-side rendering needed for bots → solution: **Next.js**
- Initial load → heavy bundle → solution: **code splitting, lazy loading**
- Browser history → manage with React Router

7. Meri Analogy

- MPA = old-school office → har file open → desk reset → slow
- SPA = modern office → digital dashboard → click tab → content update instantly
- React = **control room operator** → decide which section show without desk reset
- User = employee → fast, smooth experience

8. Key Takeaways

1. SPA = single HTML page + multiple dynamic views
2. Navigation → no reload → React Router handles
3. State management → Context / Redux / useState / useReducer
4. API calls → dynamically fetch data → update UI
5. SEO → consider server-side rendering for marketing
6. Performance → lazy loading, code splitting

Routes, dynamic routes, nested routes

1. Routes kya hai?

- **Route = path + component**
- Browser URL match hota hai → React component render hota hai
- Example: `/about` → About component

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

function Home() { return <h1>Home</h1>; }
function About() { return <h1>About</h1>; }

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

- `Routes` = container for all Route

- `Route` = single URL → component map

3. Dynamic Routes

- **Dynamic route = URL me variable part**
- React Router me `:param` syntax
- Useful for **user profiles, posts, products**

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

function User() {
  const { id } = useParams(); // grab dynamic part
  return <h1>User ID: {id}</h1>;
}

// Route
<Route path="/user/:id" element={<User />} />
```

- URL: `/user/101` → `id = 101`
- URL: `/user/202` → `id = 202`
- Same component → multiple data

3. Nested Routes

- **Nested = child component inside parent route**
- Parent component → `<Outlet />` → child render
- Useful for **dashboard, settings pages**

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

function Dashboard() {
  return (
    <div>
      <h1>Dashboard</h1>
      <nav>
        <Link to="stats">Stats</Link> | <Link to="reports">Reports</Link>
      </nav>
      <Outlet /> {/* child render */}
    </div>
  );
}

function Stats() { return <p>Stats Page</p>; }
function Reports() { return <p>Reports Page</p>; }

<Routes>
  <Route path="/dashboard" element={<Dashboard />}>
    <Route path="stats" element={<Stats />} />
    <Route path="reports" element={<Reports />} />
  </Route>
</Routes>
```

- URL `/dashboard/stats` → Dashboard + Stats
- URL `/dashboard/reports` → Dashboard + Reports
- Parent UI remain constant, child changes

---

4. Meri Analogy

- **Routes = city map** → URL decide karta hai kaunsa building open hoga
- **Dynamic Routes = personalized address** → /user/101 → specific house
- **Nested Routes = building ke floors** → dashboard = ground floor, stats/reports = first floor
- **Outlet = elevator** → child floor me render

---

5. Key Takeaways

1. **Route** = URL → component mapping
2. **Dynamic Route** = variable in URL → multiple data with same component
3. **Nested Route** = parent + child layout → UI hierarchy maintained
4. `useParams()` → grab dynamic part
5. `Outlet` → render nested children

6. Use `Link` for SPA navigation → no reload
7. Clean, maintainable, scalable routing structure

Fetching API'S

1. API Fetching kya hai?

- **API = Application Programming Interface** → backend se data lene ka tarika
- React frontend me → fetch data from server → render UI
- Common examples: JSONPlaceholder, REST APIs, OpenWeather, etc.

> Simple: API fetching = React ko bolo "bhai, mujhe ye data lao aur screen pe dikhao"

---

2. Fetch API (Browser Built-in)

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

function Users() {
  const [users, setUsers] = useState([]);

  useEffect(() ⇒ {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then((res) ⇒ res.json())
      .then((data) ⇒ setUsers(data))
      .catch((err) ⇒ console.log(err));
  }, []); // empty deps → run once on mount

  return (
    <div>
      <h1>Users List</h1>
      <ul>
        {users.map((user) ⇒ (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
    </div>
  );
}
```

Flow:

1. Component mount → `useEffect` run
2. `fetch()` → request backend → response
3. `res.json()` → parse JSON
4. `setUsers(data)` → state update → UI re-render

---

3. Async/Await Version

```
useEffect(() => {
  const fetchUsers = async () => {
    try {
      const res = await fetch("https://jsonplaceholder.typicode.com/users");
      const data = await res.json();
      setUsers(data);
    } catch (err) {
      console.log(err);
    }
  };
  fetchUsers();
}, []);
```

- Cleaner, easier to read
- `async/await` = modern JS best practice

---

4. Loading & Error Handling

```
const [loading, setLoading] = useState(true);
const [error, setError] = useState(null);

useEffect(() => {
  const fetchUsers = async () => {
    try {
      const res = await fetch("https://jsonplaceholder.typicode.com/users");
      if (!res.ok) throw new Error("Network error");
      const data = await res.json();
      setUsers(data);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  };
  fetchUsers();
}, []);

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

- `loading` **→ show spinner / placeholder**
- `error` **→ show friendly message**

---

5. Axios Library (Optional but pro-level)

```
npm install axios
```

```
import axios from "axios";

useEffect(() => {
  const fetchUsers = async () => {
    try {
      const res = await axios.get("https://jsonplaceholder.typicode.com/user
s");
      setUsers(res.data);
    } catch (err) {
      console.log(err);
    }
  };
  fetchUsers();
}, []);
```

- Axios → cleaner syntax, automatic JSON parsing, error handling easier

---

6. Meri Analogy

- **API call = wait staff in restaurant**
- React = customer (UI) → "bhai menu lao"
- fetch/Axios = waiter → kitchen (backend) se order laata hai
- `loading` = waiter abhi aa raha hai → placeholder
- `data/state update` = waiter le aaya dish → render table

---

7. Key Takeaways

1. `useEffect` → component mount / dependency change pe API call
2. `fetch()` → browser built-in, `res.json()` → parse data
3. `async/await` → modern JS readability
4. `axios` → optional library, easy & pro
5. Always handle **loading + error** states
6. Update state → UI re-render with new data
7. Map over array → render dynamic list

Fetach and Axios

1. Fetch API

- **Browser built-in** → no installation
- Returns **Promise**
- Must use `res.json()` to parse JSON
- Syntax:

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.log(err));
```

Pros:

- Built-in, lightweight
- No extra dependency

Cons:

- Must manually parse JSON
- Error handling tricky → only network errors by default
- No request/response interceptors

---

2. Axios

- Third-party library → `npm install axios`
- Automatically **parses JSON**
- Supports **interceptors, cancel tokens, timeout, default headers**

```
import axios from "axios";

axios.get("https://jsonplaceholder.typicode.com/users")
  .then(res => console.log(res.data))
  .catch(err => console.log(err));
```

Pros:

- Cleaner syntax
- Automatic JSON parsing
- Interceptors → pro-level feature
- Timeout, request cancellation
- Works in older browsers

Cons:

- Extra dependency
- Slightly bigger bundle

3. Async/Await Example

Fetch

```
try {
  const res = await fetch("https://jsonplaceholder.typicode.com/users");
  if (!res.ok) throw new Error("Network Error");
  const data = await res.json();
  console.log(data);
} catch (err) {
  console.log(err);
}
```

Axios

```
try {
  const res = await axios.get("https://jsonplaceholder.typicode.com/user
s");
  console.log(res.data);
} catch (err) {
  console.log(err);
}
```

- Axios = less boilerplate, automatic error handling & JSON parsing

4. When to Use Which

Feature	Fetch	Axios
Dependency	None	Needs installation
JSON Parsing	Manual (`res.json()`)	Automatic (`res.data`)
Error Handling	Manual (`res.ok`)	Automatic (`catch`)
Interceptors	❌	✅

Feature	Fetch	Axios
Timeout	Manual	Built-in
Old browser support	Modern only	Wide support

---

6. Meri Analogy

- **Fetch = bare hands** → you can do everything, but need extra effort
- **Axios = Swiss Army Knife** → ready-made tools → easy, fast, safe
- Fetch = small tasks → fine
- Axios = pro-level projects → recommended

---

6. Key Takeaways

1. **Fetch = native, lightweight, manual control**
2. **Axios = library, cleaner, auto JSON + error handling + interceptors**
3. Async/Await → modern JS practice → use with both
4. For **small/simple apps** → fetch fine
5. For **real-world, pro apps** → Axios recommended

Handling loading and Eroor States

Concept

- **Loading state** → jab API call ya heavy operation chal raha ho
- **Error state** → jab API fail ho ya exception aaye
- UI me **feedback** dena → better UX
- React me → **useState** + conditional rendering

> Simple: Loading = "wait a sec bhai", Error = "oops kuch gadbad ho gaya"

---

2. Basic Example

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

function Users() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(res => {
        if (!res.ok) throw new Error("Network error");
        return res.json();
      })
      .then(data => setUsers(data))
      .catch(err => setError(err.message))
      .finally(() => setLoading(false));
  }, []);

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

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

Flow:

1. `loading = true` → show spinner/placeholder
2. API success → `setUsers` → `loading = false`
3. API fail → `setError` → `loading = false`
4. Conditional rendering → show correct UI

---

3. Async/Await Version

```
useEffect(() => {
  const fetchUsers = async () => {
    try {
      const res = await fetch("https://jsonplaceholder.typicode.com/users");
      if (!res.ok) throw new Error("Network error");
      const data = await res.json();
      setUsers(data);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  };
  fetchUsers();
}, []);
```

- `try/catch/finally` → clean and readable
- `finally` → always stop loading

---

4. Loading Spinner

```
if (loading) return <div className="spinner">Loading...</div>;
```

- Better UX → animated spinner or skeleton UI

---

5. Error Retry Option

```
if (error) return (
  <div>
    <p>Error: {error}</p>
    <button onClick={() => window.location.reload()}>Retry</button>
  </div>
);
```

- Users can retry API call

- Pro-level user experience

---

6. Meri Analogy

- **Loading = waiter abhi order le ke aa raha hai** → table empty, user wait
- **Error = kitchen me mistake ho gayi** → user ko polite message show karo
- `finally` → waiter ja chuka, table ready
- Conditional rendering → user ko **always proper feedback**

---

7. Key Takeaways

1. **Loading state** → track API/in-progress operation
2. **Error state** → catch exceptions → show friendly UI
3. `useState` → track `loading` + `error`
4. `useEffect` → fetch API + update states
5. Conditional rendering → loading → error → data
6. Optional → **retry button, skeleton UI, spinner**
7. Always set `loading = false` in `finally`