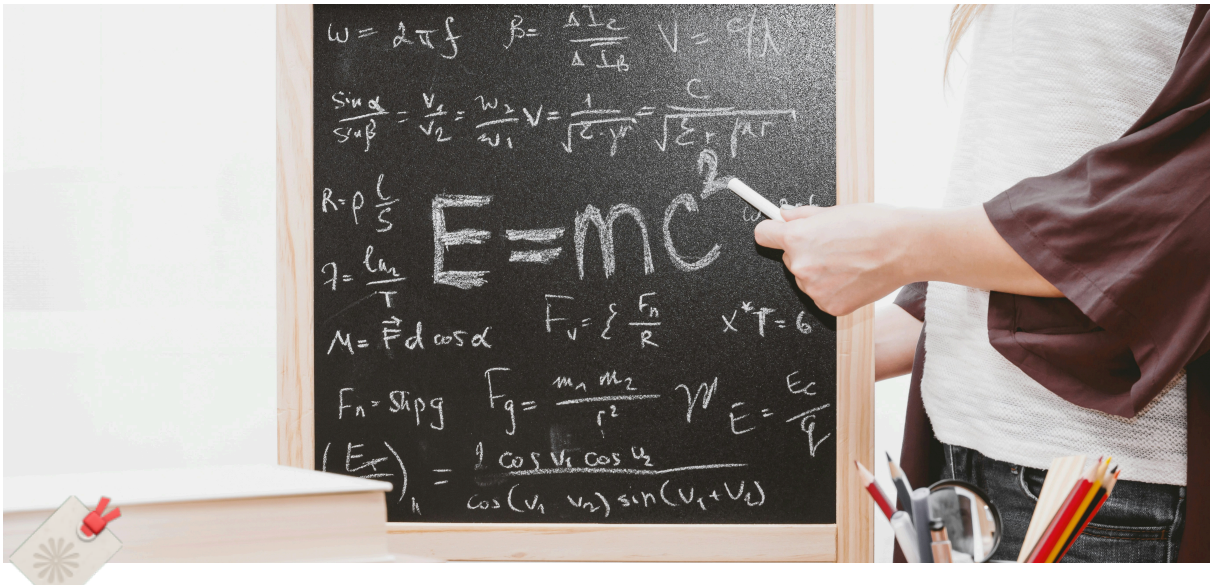


React

About the React

- **React is a JavaScript Library** developed by **Meta (Facebook)** to build **User Interfaces (UI)**, especially for **single-page applications (SPA)**.
- React follows **Component-Based Architecture**, meaning the UI is **divided into independent, reusable pieces** (components).
- A **Component** is:
  - A **reusable piece of UI**
  - **Technically**, it's just a **JavaScript function** that returns **JSX** (custom HTML-like code).
  - Example: `<Header />` is a custom tag (component).
- Think of Components as **building blocks** of any React application.
- React uses **Virtual DOM** for efficient rendering. (Explained below in edge points.)

Why we need React?

🧠 Problem with Vanilla JavaScript:

- JS follows an **Imperative approach** → You have to **manually tell the browser step-by-step what to do.**
  - More lines of code
  - Harder to manage logic as app grows
  - DOM manipulation is error-prone and slow

💡 React's Solution:

- React follows a **Declarative approach** → You just tell **what the UI should look like** for a given state.
  - React handles the DOM updates for you
  - Cleaner, shorter, and more maintainable code
  - React automatically handles **state-to-UI** mapping

🔍 Important Edge Points (Must-Know)

- ✅ React uses **Virtual DOM**:
  - A lightweight in-memory copy of the real DOM.
  - On state change, React compares (diffing) Virtual DOM vs actual DOM and updates only the changed parts. **(Efficient)**
- ✅ JSX (JavaScript XML):
  - It lets you write HTML-like code in JavaScript.
  - JSX gets **transpiled to** `React.createElement()` internally.
- ✅ SEO Concern:
  - React apps are not SEO-friendly by default (due to SPA nature), but can be fixed using **Next.js** (a React framework).
- ✅ One-Way Data Binding:
  - Data flows from **parent → child** component via **props**.
  - This ensures **better control** over UI state.

UseState Hook

🔷 What is a React Hook?

- **React Hook** is simply a **utility function** provided by React to manage **state** and **lifecycle features** in **function components**.
- React hooks **start with "use"** (e.g., `useState`, `useEffect`, `useRef`, etc.).

🧠 Why `useState`?

- In React, if you want **dynamic data to reflect on the UI**, you need a **state**.
- But function components don't have `this.state` (like class components).
- That's why we use `useState` **Hook** to create and manage state in function components.

✅ How `useState` Works:

```
const [stateVariable, setStateFunction] = useState(initialValue);
```

- `stateVariable` → The current value of your state.
- `setStateFunction` → A function used to update the state.
- `initialValue` → The starting value of the state.

Lifting State Up in React

❓ Problem Statement

In React:

- Data normally flows **from Parent → Child** using `props`.
- But what if you want **Child → Parent communication**?
    - Like a child component sending data to its parent?

➡️ That's where **Lifting State Up** comes in.

🔄 What is Lifting State Up?

> Lifting state up means:
>
> "Move the shared state from the child to the parent component so that the parent can manage it, and pass it back down via props."

UseEffect Hook

🔷 What is `useEffect`?

- `useEffect` is a **React Hook** that allows you to **perform side effects** in function components.
- It runs **after the component renders**.

🧠 What is a Side Effect?

> Side effects are operations that happen outside the normal UI rendering flow, such as:

- ✅ Fetching data from an API
- ✅ Directly interacting with the DOM
- ✅ Setting or updating `document.title`
- ✅ Setting up subscriptions or timers
- ✅ Cleaning up before the component is removed

🧪 useEffect Syntax:

```
useEffect(() => {
  // your side effect code here
}, [dependencies]);
```

- 🔷 This function runs **after rendering**.
- 🔷 The **dependency array** controls **when** the effect runs.

🧨 Types of useEffect Execution

```
// 1️⃣ Runs on every render
useEffect(() => {
  console.log("Component rendered or re-rendered");
});

// 2️⃣ Runs only once on initial render (componentDidMount)
useEffect(() => {
  console.log("Component mounted");
}, []);
```

```
// 3️⃣ Runs when a specific state/prop changes
useEffect(() => {
  console.log("Text changed");
}, [text]);

// 4️⃣ Cleanup function - runs when component unmounts
useEffect(() => {
  console.log("Component mounted");

  return () => {
    console.log("Component unmounted or before re-running");
  };
}, []);
```

🧠 useState vs useEffect

Feature	`useState`	`useEffect`
Purpose	Create and manage state (data)	Perform side effects after rendering
Returns	`[value, setValue]` → value & updater function	Nothing returned (just runs the effect code)
Triggered When	State is updated using `setState`	Runs after component renders, based on dependencies
Common Use	UI updates, input values, toggles, counters	API calls, DOM changes, timers, event listeners

🧠 Important Terms:

- **Mounting** → When component renders on the screen (initial time)
- **Unmounting** → When component is removed from the DOM
- **Effect Cleanup** → You return a function from `useEffect()` to **clean up resources**, e.g., timers, subscriptions

🗒️ Interview Tips:

> Q: What is the purpose of the dependency array in useEffect?
>
> ✅ It tells React **when to re-run the effect**.

- `[]` → Run only once
- `[state]` → Run when state changes
- No array → Run on every render

> Q: Can I use async function directly inside useEffect?
>
> ❌ No. Instead, define async function inside and call it.

```
useEffect(() => {
  const fetchData = async () => {
    const res = await fetch('/api/data');
    // handle res
  };
  fetchData();
}, []);
```

📢Toastify Setup in React

React-Toastify is a popular library to display toast notifications in React applications. Here's how to set it up step-by-step:

1. Install React-Toastify:

First, make sure you have installed **React-Toastify** using npm or yarn:

```
npm install react-toastify
```

2. Import Required Components:

In your **App.js** (or your main component), you need to import **ToastContainer** and the required CSS file for styling the toasts:

```
// Importing ToastContainer component
import { ToastContainer } from "react-toastify";

// Importing the Toastify CSS file for styling
import "react-toastify/dist/ReactToastify.css";
```

3. Add `ToastContainer` in the Render Method:

In your **ReactDOM.render** or **React 18** method, wrap your application with the **ToastContainer** component, which will display the toasts globally.

```
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <div>
    <App />
    {/* ToastContainer to display toasts */}
    <ToastContainer />
  </div>
);
```

4. Triggering Toast Notifications:

Now, to trigger a toast message in your application, use the `toast` function provided by **React-Toastify**. Here's how you can do it:

```
import { toast } from "react-toastify";

// Triggering a success toast
toast.success("This is a success message!");

// Triggering an error toast
toast.error("Something went wrong!");
```

5. Customizing Toasts:

You can also customize the appearance and behavior of the toasts, such as position, auto-close time, etc., by passing options to **ToastContainer** or each toast:

```
// Example of a customized toast notification
toast.success("Success!", {
  position: toast.POSITION.TOP_CENTER,
  autoClose: 5000, // Close after 5 seconds
  hideProgressBar: true,
});
```

📡API Calling in React (Using useEffect)

1. Create State to Store API Data

```
const [courses, setCourses] = useState(null);
```

Here, `courses` is a state variable to store fetched data, and `setCourses` is used to update it.

2. useEffect for Side Effects (API Calls)

```
async function fetchData(){
  try{
    let response = await fetch(apiUrl);   // 1. Fetch data from API
    let output = await response.json();   // 2. Convert response to JSON
    setCourses(output.data);              // 3. Store data in state
  }
  catch(err){
    toast.error("Network Issue");         // 4. Handle error
  }
}

useEffect(()⇒{
  fetchData();   // Function Call
},[]);          //First Render Pe Sirf Call Hoga
```

📌 Explanation:

Step	What Happens?
`useEffect()`	Executes after component mounts (because of empty `[]` dependency array).
`fetchData()`	Async function that performs the actual API call.
`fetch(apiUrl)`	Sends HTTP GET request to the given API endpoint.
`res.json()`	Converts the response body to JSON format.
`setCourses()`	Updates state with fetched data → triggers re-render.

Step	What Happens?
`catch` block	Handles errors and displays a toast notification (if using `react-toastify`).

🧠 How to Convert Key-Value Based JSON to Array in React

📦 Problem Statement (Simple Hinglish Explanation)

Humare paas jo data aata hai wo **key-value format** mein hota hai:

```
{
  "Development": [ {}, {}, {} ],
  "Business": [ {}, {}, {} ],
  "Design": [ {}, {}, {} ],
  ...
}
```

Har key (jaise `"Development"`, `"Business"`) ek category ko represent karti hai aur uski value ek **array of course objects** hoti hai.

🙋‍♂️ But hum kya chahte hain?

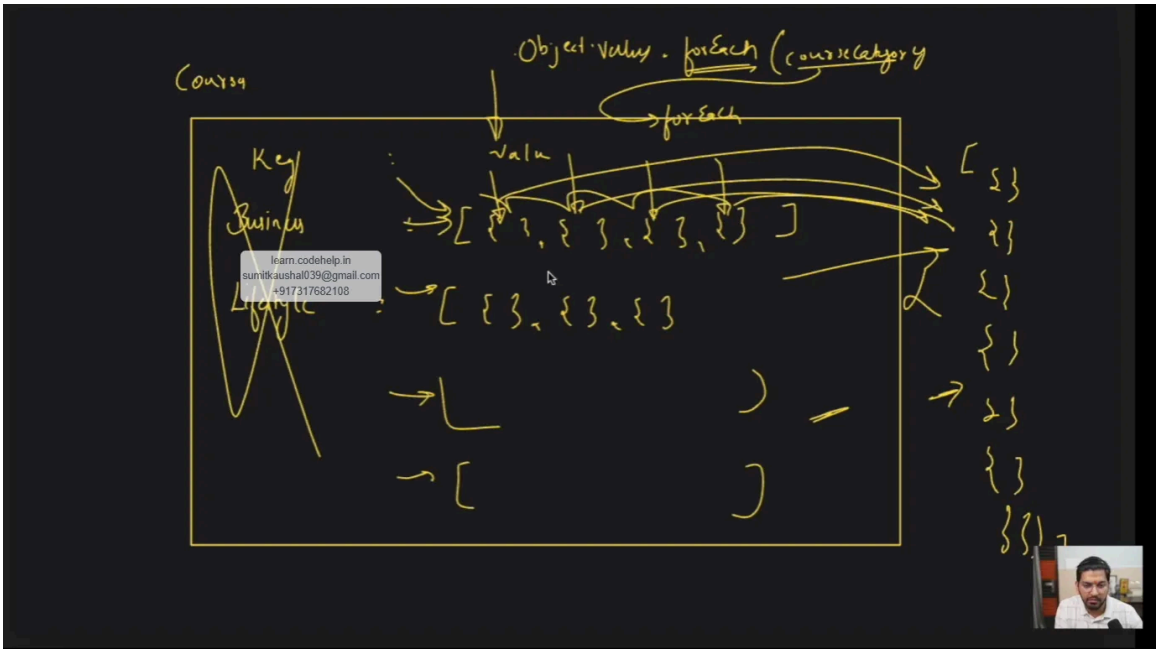
Hume chahiye:

🔷 Sare category ke courses ko ek **hi array** mein combine karna —

taaki hum `map()` loop lagake easily sab course dikhayein UI par.

🔧 Solution Approach

1. JSON object ke andar sirf values extract karni hai.
2. Har ek category ke andar jo courses ka array hai usko loop karke collect karna hai.
3. Final result hoga → ek **flat array** of all courses.



✅ Final Code (Reusable Format)

```
let allCourses = [];

const getCourses = () => {
  // Object.values() → sabhi key ke values ko extract karega (array of arrays)
  Object.values(courses).forEach((courseCategory) => {
    // Har category ke andar ke course objects ko allCourses me push karo
    courseCategory.forEach((course) => {
      allCourses.push(course);
    });
  });

  return allCourses; // Return flat array of all courses
};
```

🧪 Example Input Data (from your JSON)

```
{
  "Development": [
    { "title": "Web Dev 101", "description": "..." },
    { }, { }, ...
```

```
  ],
  "Business": [ { }, { } ],
  "Design": [ { }, { } ]
}
```

✅ Output After Conversion:

```
[
  { "title": "Web Dev 101", "description": "..." },
  { }, { }, { }, ...
]
```

🎯 React Category Filter System (with Buttons)

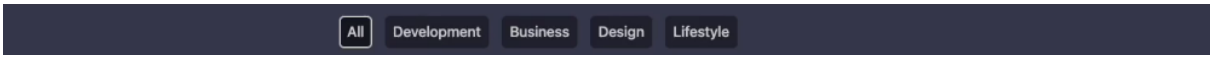
We are building a system where users can filter courses by categories like:

🔷 Development, Business, Design, Lifestyle, All

This is how it looks visually:

✅ Goal:

- Show all courses when **“All”** is selected.
- Show filtered courses when a specific category is clicked.



🔧 Step-by-Step Implementation:

🧩 Step 1: Create State for Selected Category

We need to track which category is selected by the user.

```
const [category, setCategory] = useState(filterData[0].title);
```

🔶 This is usually defined in the **App.js** file.

🔶 `filterData[0].title` means initially "All" is selected.

🧩 Step 2: Pass Props to Filter Component

Now pass the category state to the Filter component (where the buttons exist):

```
<Filter
  filterData={filterData}
  category={category}
  setCategory={setCategory}
/>
```

🧩 Step 3: Filter Component (Button Click Logic)

Here, we'll display all the filter buttons and update the selected category on click.

```
export const Filter = ({ filterData, category, setCategory }) => {

  function filterHandler(title) {
    setCategory(title); // update selected category
  }

  return (
    <div className="w-11/12 flex flex-wrap max-w-max space-x-4 gap-y-4
mx-auto py-4 justify-center">
      {filterData.map((data) => {
        return (
          <buttononClick={() => filterHandler(data.title)}
            className={`text-lg px-2 py-1 rounded-md font-medium text-white
bg-black hover:bg-opacity-50 border-2 transition-all duration-300`}
            key={data.id}
          >
            {data.title}
          </button>
        );
      })}
    </div>
  );
};
```

📌 **filterHandler**: Updates the category using `setCategory`.

🧩 Step 4: Show Filtered Courses Based on Selected Category

Now that we know which category is selected, use that to display relevant courses:

```
const getCourses = () => {
  if (category === "All") {
    let allCourses = [];

    Object.values(courses).forEach((courseCategory) => {
      courseCategory.forEach((course) => {
        allCourses.push(course);
      });
    });

    return allCourses; // show all combined courses
  } else {
    return courses[category]; // show only selected category courses
  }
};
```

📝 Creating Forms in React – Step-by-Step Guide

1. Introduction

- **Problem**:
  - Brute approach me har input ke liye separate state variable aur handler function likhna padta hai.
  - Code repetitive ho jata hai, aur form ko maintain karna mushkil ho jata hai.
- **Solution (Optimized Approach)**:
  - Ek state object banate hain jisme saare form inputs (like first name, last name, email, etc.) store honge.
  - Ek single change handler use karke, hum state update kar sakte hain.
  - Controlled components ka concept follow karte hain, jisme each input ka value attribute hota hai jo state se linked hota hai.

2. Brute Approach (Non-Optimized)

Code Sample:

```
import { useState } from "react";
import "./App.css";

function App() {
  const [firstName, setFirstName] = useState("");
  const [lastName, setlastName] = useState("");

  console.log(firstName);
  console.log(lastName);

  function changeFirstNameHandler(event) {
    setFirstName(event.target.value);
  }

  function changeLastNameHandler(event) {
    setlastName(event.target.value);
  }

  return (
    <div className="App">
      <form>
        <input type="text"
          placeholder="Enter your first name"
          onChange={changeFirstNameHandler}
        />
        <br /><br />
        <input type="text"
          placeholder="Enter your last name"
          onChange={changeLastNameHandler}
        />
      </form>
    </div>
  );
}
```

```
export default App;
```

Key Issues:

- Multiple state variables and handler functions likhne padte hain.
- Code ka duplication aur unoptimized structure.

3. Optimal Approach Using an Object

Steps & Explanation:

1. **Create a Single State Object**:
   - Instead of multiple state variables, define one object to hold all form data.
   - **Note**: *State ke initial values ko object ke properties ke through set karo.*
2. **Name Attribute**:
   - Har input field me `name` attribute same hona chahiye jaise state object ki keys.
   - Yeh help karta hai single change handler ko exactly pata chalne me ki kis field ka update karna hai.
3. **Single Change Handler**:
   - Event ko destructure kar ke `{ name, type, value, checked }` nikalte hain.
   - Check karo agar input type `checkbox` hai; uske liye `checked` property use karo, nahin toh `value` use karo.
   - State ko update karne ke liye spread operator (`...prevFormData`) ka use karo.
4. **Controlled Components**:
   - Har input element ka value attribute state se linked hona chahiye.
   - Isse hum ensure karte hain ki UI aur state hamesha sync me rahein.
5. **Connecting Input with Label**:
   - Input me `id` aur label me `htmlFor` attribute same rakho for accessibility aur clear association.

Code Sample (Improved & Generalized Form):

```
import { useState } from "react";
import "./App.css";

function App() {
  // Using a single object to maintain state for the entire form
  const [formData, setFormData] = useState({
    firstName: "",
    lastName: "",
    email: "",
    comments: "",
    isVisible: true,
    mode: "",
    favCar: ""
  });

  // Single handler for all form fields
  function changeHandler(event) {
    const { name, type, value, checked } = event.target;
    setFormData((prevFormData) => ({
      ...prevFormData,
      [name]: type === "checkbox" ? checked : value,
    }));
  }

  // Form submit handler
  function submitHandler(event) {
    event.preventDefault(); // Prevent default form submission behavior
    console.log("Printing Form Data");
    console.log(formData);
  }

  return (
    <div className="App">
      <form onSubmit={submitHandler}>
        {/* First Name Field */}
        <input type="text"
```

```
        placeholder="Enter your first name"
        onChange={changeHandler}
        name="firstName"
        value={formData.firstName}
      />
      <br /><br />

      {/* Last Name Field */}
      <input type="text"
        placeholder="Enter your last name"
        onChange={changeHandler}
        name="lastName"
        value={formData.lastName}
      />
      <br /><br />

      {/* Email Field */}
      <input type="email"
        placeholder="Enter your email here"
        onChange={changeHandler}
        name="email"
        value={formData.email}
      />
      <br /><br />

      {/* Comments Field */}
      <textarea placeholder="Enter your comments"
        name="comments"
        onChange={changeHandler}
        value={formData.comments}
      />
      <br /><br />

      {/* Checkbox Field */}
      <input type="checkbox"
        onChange={changeHandler}
        name="isVisible"
        id="isVisible"
```

```
        checked={formData.isVisible}
      />
      <label htmlFor="isVisible">Am I visible?</label>
      <br /><br />

      {/* Radio Buttons for Mode */}
      <fieldset>
        <legend>Mode:</legend>
        <input type="radio"
          onChange={changeHandler}
          name="mode"
          value="Online-Mode"
          id="Online-Mode"
          checked={formData.mode === "Online-Mode"}
        />
        <label htmlFor="Online-Mode">Online Mode</label>
        <input type="radio"
          onChange={changeHandler}
          name="mode"
          value="Offline-Mode"
          id="Offline-Mode"
          checked={formData.mode === "Offline-Mode"}
        />
        <label htmlFor="Offline-Mode">Offline Mode</label>
      </fieldset>
      <br />

      {/* Select Dropdown */}
      <label htmlFor="favCar">Tell me your favourite Car:</label>
      <select id="favCar"
        name="favCar"
        value={formData.favCar}
        onChange={changeHandler}
      >
        <option value="fortuner">Fortuner</option>
        <option value="tharr">Tharr</option>
        <option value="legender">Legender</option>
        <option value="defender">Defender</option>
```

```
        </select>
        <br /><br />

        {/* Submit Button */}
        <button>Submit</button>
      </form>
    </div>
  );
}

export default App;
```

4. Important Notes

- **Single State Object**:
  - Helps in maintaining all form data in one place.
  - Easier to update and manage when forms grow complex.
- **Name Attribute**:
  - Har input ka `name` attribute state ke keys ke saath match karna chahiye.
- **Controlled Components**:
  - Link each input field's `value` to the state to maintain a single source of truth.
- **Checkbox Handling**:
  - Checkbox ke liye, `value` attribute ke bajaye `checked` attribute ko use karo.
- **Label and Input Association**:
  - Input me `id` aur label me `htmlFor` same value dene se accessibility improve hoti hai.
- **Single Change Handler**:
  - Use destructuring (`{ name, type, value, checked }`) to update state based on input type.
  - Reduce repetition and make the code more maintainable.

⚛️ React Router – Navigate Without Refreshing the Page

React Router allows you to move from one page to another **without reloading** the entire page (Single Page Application behavior).

✅ Why Use React Router?

- Move between pages/components without refreshing the browser.
- Maintain state and performance in a **Single Page Application** (SPA).
- Replace traditional HTML `<a>` tags with React-specific routing.

🧱 Step-by-Step Setup

🛠️ Step 1: Install React Router DOM

Use the command below to install React Router:

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

🌐 Step 2: Setup `BrowserRouter`

`BrowserRouter` connects the React app with the browser URL.

📍 **Wrap your `App` component in `index.js`:**

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

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

🧭 Step 3: Define Routes using `Routes` and `Route`

📍 Go to `App.js`, and set up multiple pages:

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

```
function App() {
  return (
    <div className="App">
      <Routes>
        <Route path="/" element={<div>Home Page</div>} />
        <Route path="/support" element={<div>Support Page</div>} />
        <Route path="/labs" element={<div>Labs Page</div>} />
        <Route path="/about" element={<div>About Page</div>} />
      </Routes>
    </div>
  );
}
```

🔗 Navigating Between Pages

🧵 Using `<Link />` for Navigation

React uses `<Link>` instead of anchor `<a>` for navigating:

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

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

🌟 `NavLink` – Advanced Navigation

`NavLink` works like `Link` but gives the clicked link an **active class** automatically (helpful in highlighting the current page):

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

<NavLink to="/">Home</NavLink>
<NavLink to="/support">Support</NavLink>
```

👨‍👧 Nested (Child) Routes / Parent-Child Routing

🧩 Structure Example:

```
<Route path="/" element={<Layout />}>
  <Route index element={<Home />} />
  <Route path="about" element={<About />} />
  <Route path="labs" element={<Labs />} />
  <Route path="support" element={<Support />} />
  <Route path="*" element={<Error />} />
</Route>
```

✅ `Layout` component is the parent.

✅ Inside `Layout` component, use `<Outlet />` to render children routes.

🧯 What is `<Outlet />`?

When you use nested routes, the children routes need a placeholder in the parent to be shown.

📍 Example inside `Layout.js`:

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

function Layout() {
  return (
    <div>
      <h1>This is a Layout</h1>
      <Outlet /> {/* Renders the child component */}
    </div>
  );
}
```

🚨 Special Routes

1. Index Route

Used when no additional path is provided (default child route):

```
<Route index element={<Home />} />
```

2. Wildcard Route

Used to catch undefined URLs (Error Page):

```
<Route path="*" element={<Error />} />
```

🚀 Programmatic Navigation with `useNavigate()`

`useNavigate` is used to navigate from one page to another using a button click or programmatically.

📍 Example:

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

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

  function navigateHandler() {
    navigate("/about");
  }

  return (
    <div>
      <h2>This is Labs</h2>
      <button onClick={navigateHandler}>Move to About</button>
    </div>
  );
}
```

🛠️ React Custom Hook (Step-by-Step Guide)

📌 What is a Custom Hook?

A **custom hook** is a **reusable function** in React that starts with the word `use` and lets you extract and share logic between components **without repeating code**.

💡 Why Create a Custom Hook?

In real projects, like a **GIF Generator App**, you might need to fetch data multiple times (e.g., random GIFs and tagged GIFs).

If you repeat the same fetching logic in multiple components:

- Your code becomes **messy** and **hard to maintain**.
- Your app becomes **less scalable**.
- So, custom hooks help to **write clean, DRY (Don't Repeat Yourself) code**.

🔌 Axios – Making API Calls Simplified

We use **Axios** instead of the native `fetch`:

✅ Axios Benefits:

- Automatically parses JSON (no need for `.json()`)
- Easier error handling
- Shorter syntax

📦 Basic API Call using Axios:

```
async function fetchData() {
  setLoading(true);
  try {
    const url = `https://api.giphy.com/v1/gifs/random?api_key=${API_KEY}`;
    const { data } = await axios.get(url);
    const imageSource = data.data.images.downsized_large.url;
    setGif(imageSource);
  } catch (error) {
    toast.error("API Error");
  }
  setLoading(false);
}
```

🔁 Steps to Create a Custom Hook

✅ Step 1: Decide What Logic to Extract

We are repeating:

- API call logic
- Setting loading
- Storing image/gif

We'll move this into a custom hook.

✅ Step 2: Naming the Hook

➡️ Must start with the word `use` (React's rule)

📝 We'll name our hook: `useGif`

✅ Step 3: Create the Custom Hook File

📁 Create a file like: `useGif.js`

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

const API_KEY = process.env.REACT_APP_GIPHY_API_KEY;
const BASE_URL = `https://api.giphy.com/v1/gifs/random?api_key=${API_KEY}`;
```

✅ Step 4: Define the Hook Logic

```
export const useGif = (tag) => {
  const [gif, setGif] = useState("");
  const [loading, setLoading] = useState(false);

  async function fetchData(tag) {
    setLoading(true);
    try {
      const { data } = await axios.get(
        tag ? `${BASE_URL}&tag=${tag}` : BASE_URL
      );
      const imageSource = data.data.images.downsized_large.url;
      setGif(imageSource);
    } catch (error) {
      console.error("GIF API Error:", error);
      toast.error("Failed to fetch GIF");
    } finally {
      setLoading(false);
    }
```

```
  }

  useEffect(() => {
    fetchData(tag); // Will run on mount and whenever tag changes
  }, [tag]);

  return { gif, loading, fetchData };
};
```

📦 What this hook returns?

```
{
  gif,       // the image URL
  loading,   // true/false for spinner
  fetchData  // method to refetch manually (on button click)
}
```

🔁 What is Prop Drilling?

➤ Simple Definition:

Prop Drilling means passing data from a **parent component** down to multiple **child components**, even if some of them don't need it — just to get it to the component that does.

🧠 Real-Life Example:

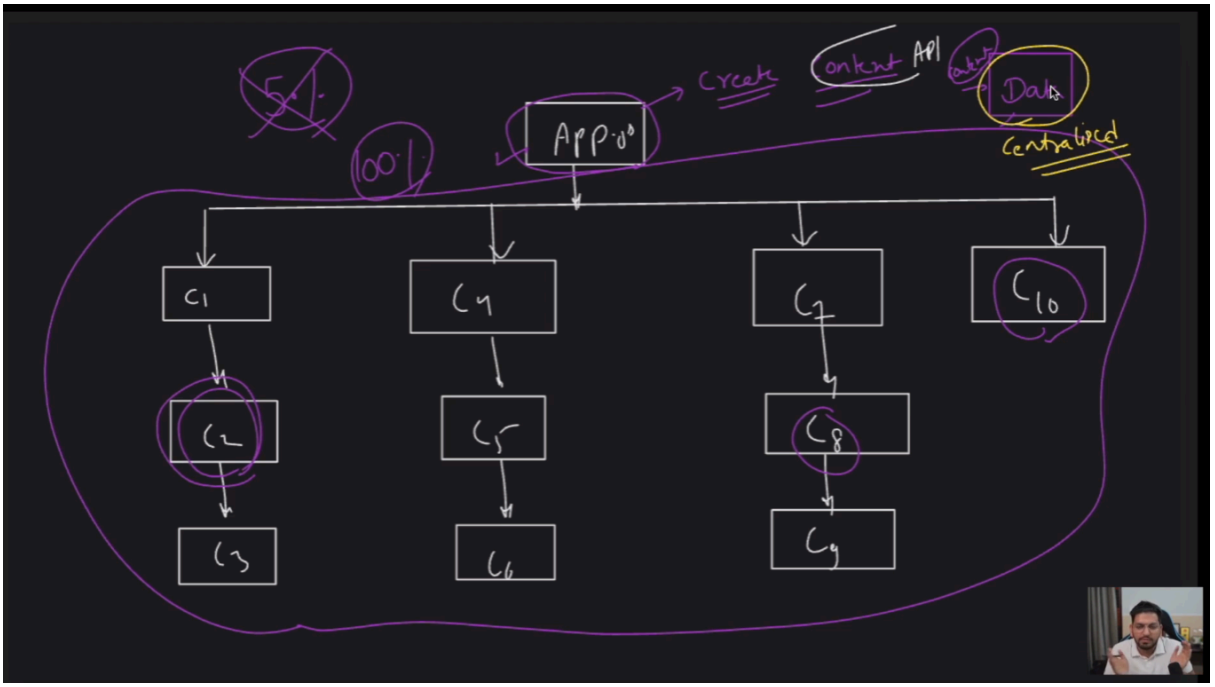
Imagine a family tree — a grandparent (App) wants to give ₹100 to the great-grandchild (a nested component). But there's no direct connection. So:

- Grandparent → Parent → Child → Great-Grandchild

  Each one just **passes the money** down even if they **don't use it**.

⚠️ Problem with Prop Drilling:

- Code becomes **messy** and **difficult to manage**.
- Performance goes **down** because unnecessary components re-render.
- Hard to **track the flow** of data.



🧰 Solution: Context API

🔍 What is Context API?

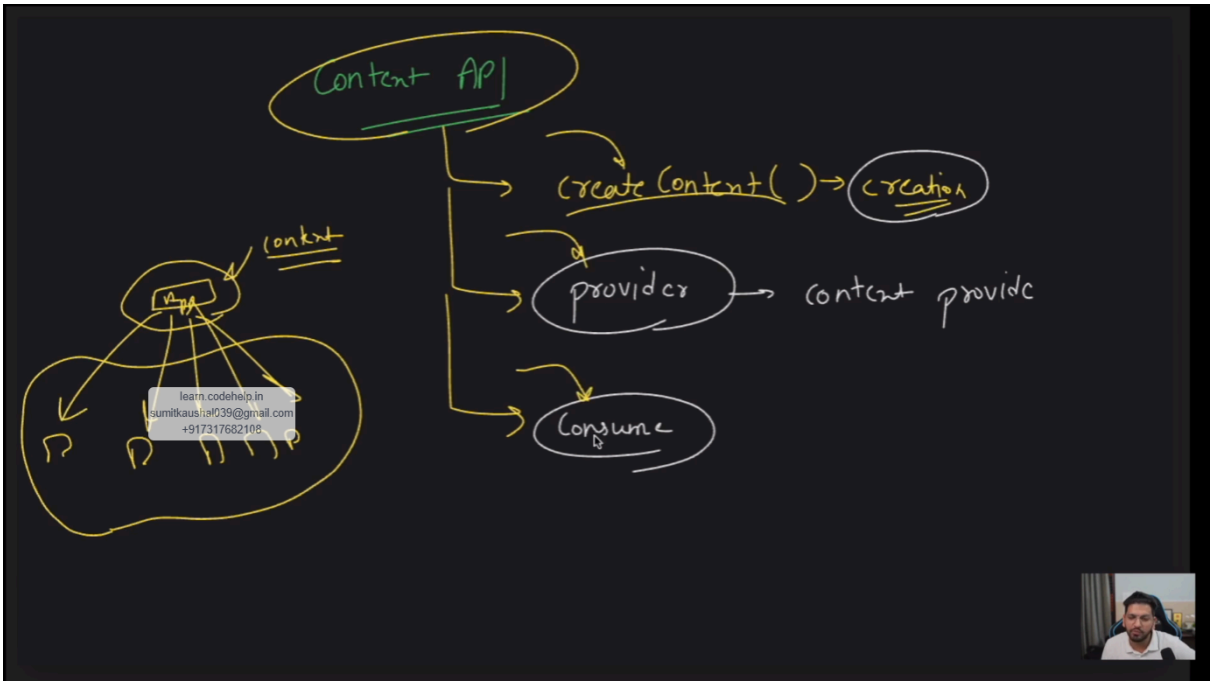
React's Context API helps you to **share data globally** (like a global state) across components **without passing props manually** at every level.

📦 Think of it like:

A **data box (context)** where you keep your important data and any component that needs it can **open the box and use the data** — no need to pass it hand-to-hand.

✅ When to Use Context API?

- When **too many props are passed** across levels.
- When multiple components need access to the **same data or functions**.
- To handle things like: `dark mode`, `language`, `user data`, `API state`, `authentication`, etc.



🛠️ Step-by-Step: How to Use Context API in React?

1. Create a `context/` folder in your project

Inside it, create a file: `AppContext.js`

2. Inside `AppContext.js` : Create the context

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

// Step 1: Create the context (the box)
export const AppContext = createContext();
```

3. Create the Provider Function

This provider will **wrap your entire App** and give access to data inside the box.

```
export default function AppContextProvider({ children }) {
  // All the shared state and functions go here
  const [loading, setLoading] = useState(false);
  const [posts, setPosts] = useState([]);
  const [page, setPage] = useState(1);
  const [totalPages, setTotalPages] = useState(null);
```

```
// Function to fetch blog posts
async function fetchBlogPosts(page = 1) {
  setLoading(true);
  let url = `${baseUrl}?page=${page}`;

  try {
    const result = await fetch(url);
    const data = await result.json();
    setPosts(data.posts);
    setPage(data.page);
    setTotalPages(data.totalPages);
  } catch (error) {
    console.log("Error in fetching data");
    setPage(1);
    setPosts([]);
    setTotalPages(null);
  }
  setLoading(false);
}

// Function to handle page change
function handlePageChange(page) {
  setPage(page);
  fetchBlogPosts(page);
}

// Step 2: Store all values/functions in an object
const value = {
  loading,
  setLoading,
  posts,
  setPosts,
  page,
  setPage,
  totalPages,
  setTotalPages,
  fetchBlogPosts,
  handlePageChange,
```

```
  };

  // Step 3: Provide the value to children
  return <AppContext.Provider value={value}>{children}</AppContext.Provi
der>;
}
```

4. Wrap your App with the Provider

Go to `index.js`:

```
import ReactDOM from "react-dom/client";
import App from "./App";
import AppContextProvider from "./context/AppContext"; // 👈 import the p
rovider

const root = ReactDOM.createRoot(document.getElementById("root"));

root.render(
  <AppContextProvider>
    <App />
  </AppContextProvider>
);
```

5. Use the Context in any Component

Wherever you want to access the context data:

```
import { useContext } from "react";
import { AppContext } from "../context/AppContext";

function Blog() {
  const { posts, loading, fetchBlogPosts } = useContext(AppContext);

  return (
    <div>
      {loading ? <p>Loading...</p> : posts.map((post) => <p>{post.title}</p
>)}
    </div>
```

```
  );
}
```

📘 Understanding `useLocation` and `useSearchParams` in React Router

🔷 What are these hooks?

Hook Name	Purpose
`useLocation()`	Used to get the **current URL path** and **search parameters**
`useSearchParams()`	Used to **read or update** query parameters from the URL like `?page=2`

🧠 Real-World Use Case

Imagine you're building a **blog website** where:

- Blogs can be filtered by **tags** (like `react`, `javascript`)
- Blogs can also be filtered by **categories** (like `web-development`, `machine-learning`)
- You are using **pagination** — `?page=1`, `?page=2` in the URL

✅ To handle all this **just from the URL**, we use these two hooks.

🔎 1. `useLocation()` – For Pathname

➤ What it does:

It gives you the **current URL path**, like:

```
"/tags/react-js?page=2"
```

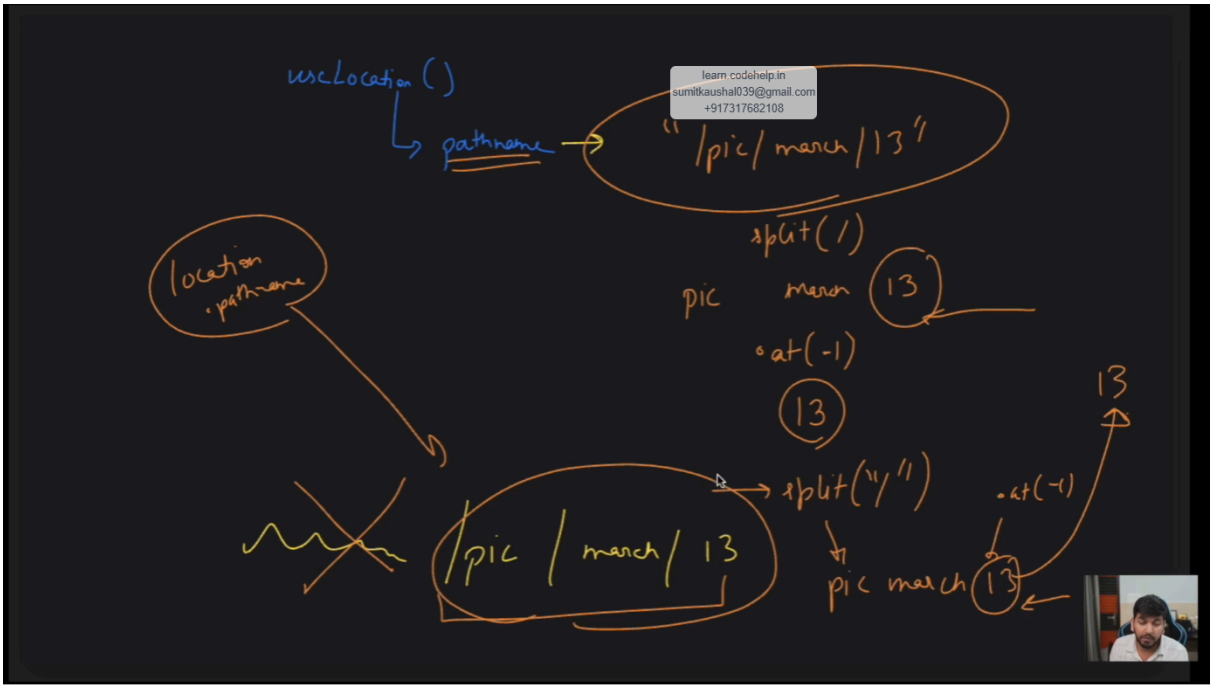
You can then:

- Get the route part: `"tags/react-js"`
- Know whether you're on a **tag page** or a **category page**

🔧 Example:

```
const location = useLocation();
console.log(location.pathname); // /tags/react-js
```

```
console.log(location.search);   // ?page=2
```



🔎 2. `useSearchParams()` – For Query Strings

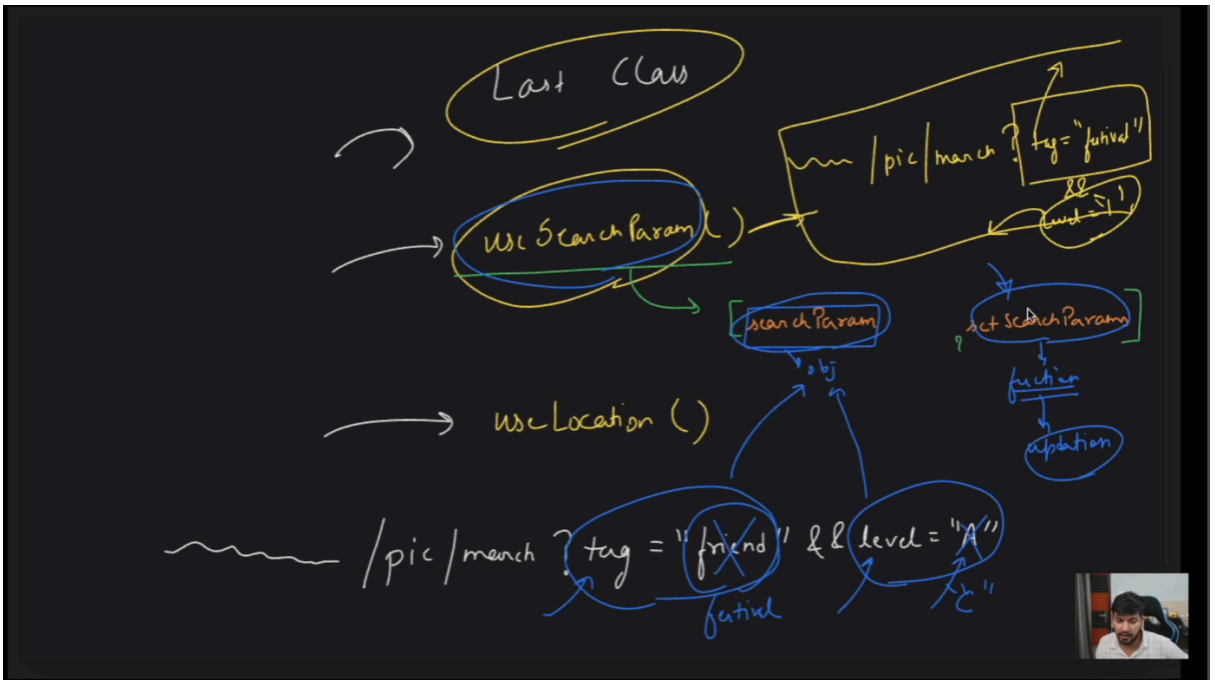
➤ What it does:

Lets you **get or set** query parameters in the URL like:

```
?page=2
```

🔧 Example:

```
const [searchParams, setSearchParams] = useSearchParams();
const page = searchParams.get("page") ?? 1; // If not found, default to 1
```



✅ Complete Breakdown of Your Code

```
const [searchParams, setSearchParams] = useSearchParams();
const location = useLocation();

useEffect(() => {
  const page = searchParams.get("page") ?? 1;

  if(location.pathname.includes("tags")){
    const tag = location.pathname.split('/').at(-1).replaceAll("-", " ");
    fetchBlogPosts(Number(page), tag)
  }
  else if(location.pathname.includes("categories")){
    const category = location.pathname.split('/').at(-1).replaceAll("-", " ");
    fetchBlogPosts(Number(page), null, category)
  }
}, [location.pathname, location.search]);
```

📘 Redux Toolkit Concept

🔷 Redux Kya Hai?

➡️ **Redux ek JavaScript library hai** jo hum **React apps me global state manage karne ke liye use karte hain**.

👉 Jab app me bahut saari states ho jaati hain (user info, theme, cart items, etc.) toh unko manage karna mushkil ho jaata hai. Redux is problem ka solution hai.

🔷 Redux Toolkit Kya Hai?

➡️ **Redux Toolkit** is the official, modern way to write Redux code.

Ye Redux ka **simplified version** hai jisme boilerplate kam likhna padta hai.

✅ Redux Toolkit ke saath:

- Code readable hota hai
- State ka structure clear hota hai
- Error chances kam ho jaate hain

🔷 Redux Main Concepts:

Concept	Explanation in Simple Hinglish
Slice	App ke state ka ek hissa. Jaise `counter`, `user`, `cart`, etc.
Reducers	Wo functions jo state ko change karte hain (e.g., increase count)
Actions	Reducer functions ko call karne ke triggers
Store	Redux ka global container jisme sab state store hoti hai
useSelector	Store se state **read** karne ke liye hook
useDispatch	Store me actions **dispatch** karne ke liye hook

🔨 Step-by-Step Guide with Counter Example

✅ Step 1: Install Redux Toolkit

```
npm install @reduxjs/toolkit react-redux
```

✅ Step 2: `createSlice()` ka Use

➡️ Redux Toolkit me state + actions + reducers ko milakar ek **slice** banate hain.

```
import { createSlice } from "@reduxjs/toolkit";

// Initial state
const initialState = {
  value: 0,
};

// Slice banana
export const counterSlice = createSlice({
  name: "counter", // slice ka naam
  initialState,    // starting value
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
  },
});

// Actions ko export karna
export const { increment, decrement } = counterSlice.actions;

// Reducer ko export karna
export default counterSlice.reducer;
```

🔎 Explanation:

- `name` : Ye slice ka naam hai.
- `initialState` : Ye state ki starting value hai.
- `reducers` : Yaha hum state change karne wale functions likhte hain.
- `state.value += 1` : Direct mutate kar sakte ho kyunki Redux Toolkit internally **Immer.js** use karta hai.

✅ Step 3: Store Create Karna (configureStore)

```
import { configureStore } from "@reduxjs/toolkit";
import counterReducer from "./slices/CounterSlice"

export const store = configureStore({
   reducer : {
      counter : counterReducer
   }
})
```

💡 Yahaan `counterReducer` ko hum `counter` naam ke under store me store kar rahe hain.

✅ Step 4: React App ko Redux Store se Connect Karna

```
import ReactDOM from "react-dom/client";
import App from "./App";
import { Provider } from "react-redux";
import { store } from "./store";

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

💡 `Provider` ek wrapper hai jo React app ko Redux store ka access deta hai.

✅ Step 5: Data Ko Use Karna – `useSelector` & `useDispatch`

🔷 `useSelector` – State Read Karne ke liye

```
import { useSelector } from "react-redux";
```

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

➡️ Yahaan hum `counter` slice ka `value` read kar rahe hain.

🔷 `useDispatch` – Action Dispatch Karne ke liye

```
import { useDispatch } from "react-redux";
import { increment, decrement } from "../slices/CounterSlice";

const dispatch = useDispatch();

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

➡️ `dispatch()` ke through hum slice ke actions call karte hain.

📘 Reducer Functions in Depth

🔷 Reducer Functions in Redux Toolkit

💡 Definition:

In Redux Toolkit, **reducer functions** are those functions that define **how the state should change** in response to an action.

➡️ **Each reducer takes 2 input parameters**:

1. `state` – current value of the slice state
2. `action` – object that contains information about what change is requested

🔶 What is `action.payload`?

Whenever we dispatch an action with some data (like an item, user, ID, etc.),

that **data travels inside** `action.payload`.

✅ So, if you want to access that passed data inside the reducer, use:

```
action.payload
```

🔨 Example: Cart Slice (Add / Remove Item)

Let’s say we’re building a shopping cart.

We want to:

- Add items to the cart
- Remove items from the cart

✅ Code Example

```
import { createSlice } from "@reduxjs/toolkit";

// Step 1: Create the slice
export const CartSlice = createSlice({
  name: "cart",           // Name of the slice
  initialState: [],       // Cart starts as an empty array

  reducers: {
    // Step 2: Add item to cart
    add: (state, action) => {
      // action.payload contains the new item (like {id, name, price})
      state.push(action.payload);  // Directly push into the array
    },

    // Step 3: Remove item from cart using its ID
    remove: (state, action) => {
      // action.payload contains the id to be removed
      return state.filter((item) => item.id !== action.payload);
    },
  },
});

// Step 4: Export actions
export const { add, remove } = CartSlice.actions;

// Step 5: Export reducer to use in store
export default CartSlice.reducer;
```

🧠 Deep Explanation:

🔷 `add` Reducer

```
add: (state, action) ⇒ {
  state.push(action.payload);
}
```

- `state`: current cart array
- `action.payload`: item object (like `{ id: 101, title: "Shoes", price: 1200 }`)
- `state.push(...)`: directly pushes the item into the cart

🔍 Redux Toolkit allows us to **mutate** the state directly using Immer.js, but behind the scenes it keeps it immutable.

🔷 `remove` Reducer

```
remove: (state, action) ⇒ {
  return state.filter((item) ⇒ item.id !== action.payload);
}
```

- `action.payload` contains the ID of the item to remove
- `filter()` returns a **new array** excluding that item
- This is a **pure function** (no mutation), which is also a valid Redux pattern

📘 `reduce()` Function Usage in React Example

Topic: Using `.reduce()` inside `useEffect` to calculate a total value (e.g. total cart amount)

✅ Code Snippet

```
useEffect(() ⇒ {
  setTotalAmount(cart.reduce((acc, curr) ⇒ acc + curr, 0));
}, [cart]);
```

🧠 Purpose of This Code

This piece of code **automatically calculates the total amount of a shopping cart** every time the `cart` changes, and updates the state (e.g. to display total on the UI).

🔍 Dissecting Step-by-Step

1. `useEffect(() => { ... }, [cart])`

- React hook that runs side-effects.
- Runs every time the `cart` array is updated.
- This ensures the total amount is always **up-to-date** with latest cart data.

Analogy: Like a calculator that runs automatically every time your `cart` changes.

2. `.reduce((acc, curr) => acc + curr, 0)`

➕ `reduce()` in JavaScript

- **Purpose**: To reduce an array into a single value.
- **Syntax**:

```
jsarr.reduce((accumulator, currentItem) => { ... }, initialValue);
```

🧱 Applied to Totaling Values

In this context:

```
jscart.reduce((acc, curr) => acc + curr, 0)
```

Parameter	Description
`acc` (accumulator)	Running total being calculated
`curr` (current)	Current item from cart array
`0`	Initial value of accumulator

Important:

- Your `cart` should be an array of numbers like `[100, 250, reduce function goes one by one:

```
textacc = 0
first item = 100 → acc = 0 + 100 = 100
next item = 250 → acc = 100 + 250 = 350
next item = 90  → acc = 350 + 90 = 440
→ Final result: 440
```

🛒 Realistic Example: Cart with Products

If `cart` is:

```
jsconst cart = [
 { id: 1, name: "Item A", price: 100 },
 { id: 2, name: "Item B", price: 250 },
 { id: 3, name: "Item C", price: 90 }
];
```

Then reduce should be:

```
jsxcart.reduce((acc, curr) ⇒ acc + curr.price, 0)
```

Updated useEffect:

```
useEffect(() ⇒ {
  const total = cart.reduce((acc, curr) ⇒ acc + curr.price, 0);
  setTotalAmount(total);
}, [cart]);
```

🔍 useRef Hook – Simple Explanation

🔶 Why useRef? (Kyu use karte hai?)

- Jab hum koi **aisa variable** banana chahte hai **jo value store karke rakhe** even after **re-render**, tab hum `useRef` use karte hai.
- Re-render hone par bhi `useRef` ki value **same rehti hai**, aur **component re-render nahi hota** agar `useRef` ka value change ho.

🔶 useState vs useRef

useState	useRef
Value change → Re-render hota hai	Value change → Re-render **nahi** hota
UI ke liye use hota hai	Mostly background ya DOM control ke liye use hota hai

✅ useRef Returns an Object

```
const btnRef = useRef();
```

Ye `btnRef` ek **object** return karta hai like this:

```
{ current: null }
```

- Aap `btnRef.current` se value **access** bhi kar sakte ho aur **change** bhi kar sakte ho.
- `current` property hi main hoti hai.

🧠 Major 2 Use Cases

1️⃣ Persist value across renders

- Aap kisi bhi value ko `useRef` me store karke rakh sakte ho.
- Wo value **re-render me lost nahi hoti**.

2️⃣ Access or Manipulate DOM directly

- React me directly DOM access nahi karte, but useRef se kar sakte ho.
- Isko `document.getElementById()` ka React version samajh lo.

🧪 Real Example: Button ka Color Change Karna (DOM Manipulation)

✅ 3 Step Process

1. **Reference Create Karna**

```
const btnRef = useRef(); // Step 1: Create Ref Krna
```

2. **Reference Link Karna**

```
<button ref={btnRef}>...</button> //Step 2: Refference Link Krna
```

3. **Reference Use Karna**

```
function changeColor() {
  btnRef.current.style.backgroundColor = "red"; // Step 3: DOM Manipul
ate
```

```
  btnRef.current.style.color = "black";
}
```

Mobbin for UI Blueprint and Dribble also code write for claude

GSPL7949

Ga@12345

OTP Screen BackPress logic aur jb hum otp enter kr rhe hai ek number enter ki toh keyboard band ho jaa rha hu

3teen issue aa rhe hai

first login failed hone par welcome screen pr navigate ho jaa rha tha aur back btn proper work nhi kr rha tha and then jb data fetch ho rha tha tb tk white screen loading show ho rha hai jo User experience bekaar kr rha tha so in issue ko fixed kia