Karyvio
LEARN | BUILD | GROW | GET HIRED

React Complete Notes

Beginner to Advanced

A modern JavaScript library for building user interfaces

Small Steps Everyday Make You a Better Developer !

1 What is React ?

Ans →

- React is a JavaScript library for building user interfaces, mainly for web applications.
- It helps us create reusable UI components and manage the UI efficiently.
- React is maintained by Meta (Facebook).
- It uses a declarative approach (we tell React what to render, not how to render).
- React makes it easy to build fast, interactive and scalable applications.

Example :-

```
function App() {
  return (
    <h1>Hello React!</h1>
  );
}
```

↳ Simple React component

Output :-

Hello React!

2 Why React ?

Ans →

- Component based architecture.
- Reusable and maintainable code.
- Fast and efficient with Virtual DOM.
- Large ecosystem and community support.
- Used by top companies (Meta, Netflix, Google, etc.).
- Flexible and easy to integrate with other libraries or frameworks.
- Great for building single page applications (SPAs).

Key Features :-

- Component based
- Declarative UI
- Virtual DOM
- Reusable components
- Rich ecosystem
- Strong community
- Flexible and scalable
- Great developer tools

Used By :-

- Meta (Facebook)
- Instagram
- Netflix
- Google
- Microsoft
- Airbnb
- Uber
- Discord
- Shopify
- and many more ...

3 History of React

Ans →

- React was created by Jordan Walke, a software engineer at Facebook.
- It was first released in 2013.
- Initially used internally at Facebook, then open-sourced in 2013.
- Over time, React has become one of the most popular JavaScript libraries in the world.
- Now it is maintained by Meta and a large community of developers.

Timeline :-

2011	→	Initial development (at Facebook)
2013	→	Open sourced
2015	→	React Native (for mobile apps)
2016	→	Major updates (v15, v16)
2018	→	React Hooks (v16.8)
2020	→	Concurrent Mode (v18)
2023+	→	Continuous improvements

4 React vs JavaScript (Vanilla DOM)

Ans →

Feature	Vanilla JavaScript (DOM)	React
Approach	Imperative (how to do)	Declarative (what to do)
DOM Updates	Manual DOM manipulation	Virtual DOM (automatic)
Code Structure	Can become complex	Component based
Reusability	Harder to achieve	Easier with components
Performance	Slower for large apps	Faster with Virtual DOM
Learning Curve	Steeper for big apps	Easier and modern
Tools & Ecosystem	Limited	Rich (Vite, Next.js, etc.)

Important Note :-

- React is a library, not a full framework (like Angular).
- It focuses only on the UI layer.
- You can use React with other libraries (e.g. React Router, Redux, etc.).
- React can be used for web, mobile (React Native), desktop (Electron) and more.

5 Where is React Used ?

Ans →

- Web applications (single page apps)
- Mobile applications (React Native)
- Desktop applications (Electron)
- Server-side rendering (Next.js)
- Progressive Web Apps (PWAs)
- Real-time applications (chat, collab tools)
- E-commerce, dashboards, SaaS products, etc.

Real World Examples :-

- Facebook → Main web application
- Instagram → Web and mobile (React Native)
- Netflix → Web application
- Airbnb → Web application
- Discord → Web application
- Shopify → Admin dashboard (Polaris)
- Many startups and big companies use React !

Same Technology Different Dreams Keep Learning

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Setup & Project Structure

Good Setup Leads to Great Projects !

Set up your first React project and understand the folder structure

1 Requirements Before You Start ?

Ans →
- A computer (Windows / macOS / Linux)
- Node.js (v18 or higher)
- npm (comes with Node.js)
- A code editor (VS Code recommended)
- Basic knowledge of HTML, CSS and JavaScript

Check Versions :-

```
node -v    // e.g. v20.11.0
npm -v     // e.g. 10.2.4
npx -v     // e.g. 10.2.4
```

Make sure Node.js is installed properly.

2 Create a New React Project (Using Vite)

Ans →
- Vite is a modern and fast build tool for React.
- It provides a better development experience compared to older tools like Create React App.
- Open your terminal and run the following commands:

Commands :-

```
# 1. Create a new React project
npm create vite@latest my-react-app -- --template react
# 2. Go to the project folder
cd my-react-app
# 3. Install dependencies
npm install
# 4. Start the development server
npm run dev
```

Explanation :-
- **npm create vite@latest** → creates a new React project using Vite.
- **my-react-app** → your project name (you can use any name).
- **--template react** → tells Vite to use the React template.
- **npm install** → installs all required dependencies.
- **npm run dev** → starts the local development server.

3 Project Structure Overview

Ans →
- After creating the project, you will see a folder structure like this:

```
my-react-app/
├── node_modules/    # Installed packages
├── public/          # Static files (e.g. images, favicon)
├── src/             # Main source code
│   ├── assets/      # Images, fonts, etc.
│   ├── App.jsx      # Main component
│   ├── main.jsx     # Entry point
│   └── index.css    # Global styles
├── .gitignore       # Git ignore file
├── index.html       # HTML template
├── package.json     # Project config and scripts
└── vite.config.js   # Vite configuration
```

Key Files Explained :-
- index.html → The main HTML file.
- main.jsx → Renders the App component to the DOM.
- App.jsx → Your main React component (where you write your UI).
- package.json → Contains project details, dependencies and scripts.
- vite.config.js → Vite configuration file (advanced settings).

Note :-
- node_modules folder contains all installed packages.
- You usually don't modify files inside node_modules.
- All your code will be inside the src/ folder.

4 Run Your First React App

Ans →
- After running **npm run dev**, you will see a URL in the terminal (usually http://localhost:5173).
- Open that URL in your browser.
- You will see a default React app running.
- Now you can start editing src/App.jsx and see the changes live (Hot Module Replacement).
- This makes development fast and productive.

Default Output :-

Next Steps :-
- Understand JSX (next page)
- Learn about components
- Explore props and state
- Build small projects
- Customize the project structure as needed

Karyvio
LEARN | BUILD | GROW | GET HIRED

React JSX

Write HTML in JavaScript

Good Syntax Leads to Clean Components and Better Apps ! ☺

1 What is JSX ?

Ans →
- JSX (JavaScript XML) is a syntax extension for JavaScript.
- It allows us to write HTML-like code inside JavaScript.
- JSX is not a separate language, it's transformed into regular JavaScript by tools like Babel (in Vite).
- It makes the code more readable and easier to write UI components.

Example :-
```
const element = (
   <h1>Hello Karyvio!</h1>
);
// Output :-
Renders a heading: Hello Karyvio!
```

Key Point :-
- JSX looks like HTML but it's actually JavaScript.
- It gets converted to JavaScript using Babel.
- Browser can't understand JSX directly.

2 JSX Expressions

Ans →
- In JSX, you can use JavaScript expressions inside curly braces { }.
- You can include variables, function calls, objects, arrays, and even ternary operators.
- JSX expressions are evaluated and the result is rendered to the UI.
- You cannot use statements (like if, for, switch) directly inside JSX.

Example :-
```
const name = "Karyvio";
const age = 21;

function App() {
  return (
    <div>
      <h2>My name is {name}</h2>
      <p>I am {age} years old.</p>
    </div>
  );
}
// Output :-
My name is Karyvio
I am 21 years old.
```

You can use :-
- Variables
- Math operations
- Function calls
- Ternary operator
- Array .map()
- Object properties

Not Allowed :-
- if statements
- for loops
- switch cases (use alternatives like .map() or helper functions)

3 JSX Attributes

Ans →
- In JSX, we use attributes to provide additional information to elements (similar to HTML).
- Some attributes are the same as HTML, but some are different.
- In React, we use camelCase for most attributes (like className, htmlFor, onClick, etc.).
- String values can be written with double quotes (") and JavaScript expressions with { }.

Common JSX Attributes

HTML Attribute	JSX Attribute	Example
class	className	
for	htmlFor	<label htmlFor="name">
onclick	onClick	<button onClick={handle}>
style	style (object)	<div style={{ color: 'red'}}>
tabindex	tabIndex	<input tabIndex={1} />
readonly	readOnly	<input readOnly />

4 JSX with Functions

Ans →
- You can call functions directly inside JSX using { }.
- Functions can return JSX elements.
- This is useful for dynamically rendering content.

Example :-
```
function getGreeting(name) {
  return <h3>Hello, {name}!</h3>;
}
function App() {
  return <div>{getGreeting("Sovit")}</div>;
}
```

Output :-
Hello, Sovit!

Function call inside JSX ☺

5 JSX Fragments

Ans →
- In JSX, a component must return a single parent element.
- To avoid adding extra unnecessary , we can use Fragments.
- Fragments let you group multiple elements without adding extra nodes to the DOM.

Example :- Using Fragment
```
function App() {
  return (
    <>
      <h1>Welcome to Karyvio</h1>
      <p>Learn React with us!</p>
    </>
  );
}
```

Alternative (Using)
```
function App() {
  return (
    <div>
      <h1>Welcome to Karyvio</h1>
      <p>Learn React with us!</p>
    </div>
  );
}
```

6 JSX Rules (Important)

Ans →
1. Must return a single parent element (use Fragment if needed).
2. Use camelCase for attributes (className, onClick, etc.).
3. Close all tags (even self-closing tags like
).
4. Use { } for JavaScript expressions.
5. We can write multiline JSX using parentheses ( ).

Quick Tips :-
- Write clean and readable JSX.
- Use meaningful component and variable names.
- Prefer Fragments to avoid unnecessary .
- Use consistent indentation.
- Let your code be beautiful and simple !

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Components

The Building Blocks of React Applications

Small Components Big Possibilities !

1 What is a Component ?

Ans →
- A component is a reusable piece of UI in React.
- It returns some JSX that describes what should appear on the screen.
- Components make the code modular, reusable and easy to maintain.
- Everything in React is a component (even App).
- Components can be small (button) or large (entire page).

Example :-

```
function Welcome() {
  return (
    <h1>Hello, React Components!</h1>
  );
}
// Usage
<Welcome />
```

Reusable
Maintainable
Scalable
Code

2 Types of Components

Ans →
- Functional Components (modern way)
- Class Components (older way, rarely used now)
- In modern React, we mostly use Functional Components with Hooks.

Functional Component (Recommended)

```
function Greeting() {
  return <h2>Hello Karyvio!</h2>;
}
```

Class Component (Old)

```
class Greeting extends React.Component {
  render() {
    return <h2>Hello Karyvio!</h2>;
  }
}
```

3 Component Structure

Ans →
- A typical component looks like this:

```
function Button() {
  return (
    <button className="btn">
      Click Me
    </button>
  );
}
```

Function name (PascalCase)
JSX to render UI
Return single parent element

Key Points :-
- Component names must start with a capital letter (e.g. Button, UserCard).
- A component must return a single parent element (use or Fragment).
- We use camelCase for component names.
- Keep components small and focused (single responsibility).
- Use meaningful names.
- Components can accept data using props.

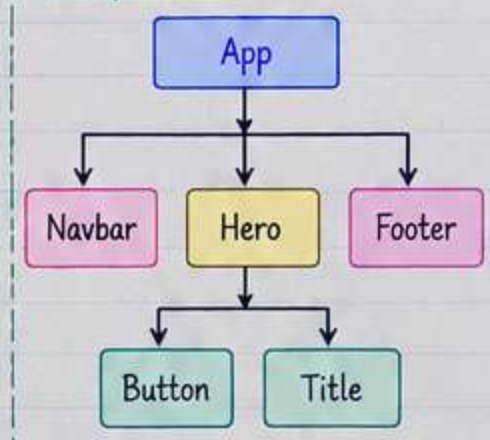
4 Component Composition

Ans →
- We can use one component inside another.
- This is called component composition.
- It helps in building complex UIs from smaller components.
- It makes the code more readable and reusable.

Example :-

```
function App() {
  return (
    <div>
      <Navbar />
      <Hero />
      <Footer />
    </div>
  );
}
```

Component Tree :-



- App is the parent component.
- Navbar, Hero, Footer are child components.
- Hero also has its own child components.
- This forms a component tree.

5 Reusable Components

Ans →
- Components can be reused multiple times across the application.
- We can pass different data (props) to make them dynamic.
- Reusable components save time and reduce duplication.
- Examples: Button, Card, Input, Navbar, etc.

Example :- (Reusing Button)

```
function App() {
  return (
    <div>
      <Button text="Login" />
      <Button text="Signup" />
      <Button text="Learn More" />
    </div>
  );
}
```

6 Best Practices

Ans →
1. Use meaningful and descriptive names.
2. Keep components small and focused.
3. Use props for dynamic data.
4. Follow consistent folder structure.
5. Extract reusable components.
6. Use comments for better readability.

Folder Structure (example) :-

```
+ src/
    ├── components/
    │     ├── Button.jsx
    │     ├── Navbar.jsx
    │     └── Card.jsx
    ├── App.jsx
    └── main.jsx
```

Quick Tips :-
- Use PascalCase for component names.
- Keep UI and logic separated.
- Reuse components as much as possible.
- Prefer functional components.
- Use proper folder structure.
- Make your components flexible using props.

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Props

Pass data to make components dynamic

Good Components Communicate Using Props !

1 What are Props ?

Ans →
- Props (short for properties) are used to pass data from a parent component to a child component.
- Props make components reusable and dynamic.
- Props are read-only (cannot be modified by child component).
- We can pass any type of data using props – string, number, boolean, array, object, function, etc.

Example :-

```
function Welcome(props) {
   return <h2>Hello, {props.name}!</h2>;
}
function App() {
   return <Welcome name="Karyvio" />;
}
// Output :-
  Hello, Karyvio!
```

Passing data using props

2 Props Syntax

Ans →
- We pass props like attributes in JSX.
- We can access props using the parameter in functional components.
- You can use any name (commonly props) or directly destructure.

Syntax :-

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

Like HTML attributes but for custom components

Important Note :-
- Props are read-only (immutable).
- Child component cannot change props (it can only use them).

3 Destructuring Props

Ans →
- Instead of using props.name, we can destructure props directly.
- It makes the code cleaner and easier to read.

Example (Without Destructuring) :-

```
function UserCard(props) {
  return (
    <div>
      <h3>{props.name}</h3>
      <p>{props.role}</p>
    </div>
  );
}
```

Same Output

Example (With Destructuring) :-

```
function UserCard({ name, role }) {
  return (
    <div>
      <h3>{name}</h3>
      <p>{role}</p>
    </div>
  );
}
```

Clean and Recommended ✓

4 Default Props

Ans →
- We can provide default values if no prop is passed.
- Useful to avoid undefined values.

```
function Button({ text = "Click Me", type = "primary"}) {
  return <button className={type}>{text}</button>;
}

// Usage
  <Button />                  // shows: Click Me
  <Button text="Submit" />    // shows: Submit
```

5 Children Prop

Ans →
- props.children allows us to pass JSX content between the opening and closing tags.
- Useful for building wrapper components like Card, Modal, etc.

```
function Card({ children }) {
  return <div className="card">{children}</div>;
}
// Usage
  <Card>
    <h3>Hello</h3>
    <p>This is inside Card</p>
  </Card>
```

Children can be any valid JSX (including other components).

6 Function Props (Callback Props)

Ans →
- We can pass functions as props (useful for event handling).
- Child component can call the function passed from parent.

```
function Button({ onClick, label }) {
  return <button onClick={onClick}>{label}</button>;
}
function App() {
  const handleClick = () => {
    alert("Button Clicked!");
  };
  return <Button onClick={handleClick} label="Click Me" />;
}
```

Used for handling events from child to parent.

7 Types of Data We Can Pass

Ans

Data Type	Example	JSX Syntax
String	"Karyvio"	<Comp name="Karyvio" />
Number	25	<Comp age={25} />
Boolean	true	<Comp isActive={true} />
Array	[1, 2, 3]	<Comp items={[1,2,3]} />
Object	{ name: "Sovit" }	<Comp user={{ name: "Sovit" }} />
Function	() => {}	<Comp onClick={handle} />
JSX element	<h1 />	<Comp header={ Hello } />

Small Props, Big Possibilities ! ☺

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Conditional Rendering

Show different UI based on conditions

Same Component Different UI That's the Power of React !

1 What is Conditional Rendering ?

Ans →
- Conditional rendering means showing different UI elements based on a condition (true/false).
- In React, we can use regular JavaScript conditions inside JSX.
- It helps to create dynamic and interactive UIs.
- Commonly used for authentication, permissions, loading states, error handling, etc.

Real World Example :-
- Show "Login" or "Logout" button based on user login status.
- Show loading spinner while fetching data.
- Show error message if API fails.
- Show different menu for admin vs user.

2 Using if-else (Outside JSX)

Ans →
- We can use regular if-else statements before the return, and store the result in a variable.
- This is useful for complex conditions.

Example :-
```
function Message({ isLoggedIn }) {
    let content;
    if (isLoggedIn) {
        content = <h2>Welcome Back!</h2>;
    } else {
        content = <h2>Please Login</h2>;
    }
    return <div>{content}</div>;
}
```

Output (isLoggedIn = true) :-
Welcome Back!

Output (isLoggedIn = false) :-
Please Login

Key Point :-
- if-else can't be used directly inside JSX.
- We use it outside JSX or with ternary / logical operators.
- Useful when we have multiple conditions (not just true/false).

3 Using Ternary Operator

Ans →
- Ternary operator is a short and simple way to conditionally render elements.
- Syntax: condition ? <TrueComponent /> : <FalseComponent />

Example :-
```
function LoginButton({ isLoggedIn }) {
    return (
        <div>
            {isLoggedIn ? (
                <button>Logout</button>
            ) : (
                <button>Login</button>
            )}
        </div>
    );
}
```

Output :-
If isLoggedIn = true → Logout
If isLoggedIn = false → Login

When to use ?
- Use ternary for simple conditions.
- If the UI is too complex, use if-else outside JSX instead.

4 Using Logical AND (&&)

Ans →
- The && operator is useful when we only want to render something if the condition is true.
- If the condition is false, it renders nothing.

Example :-
```
function Notification({ unread }) {
    return (
        <div>
            {unread && (
                <p>You have {unread} new messages!</p>
            )}
        </div>
    );
}
```

Output (unread = 5) :-
You have 5 new messages!

Output (unread = 0) :-
(Nothing is rendered)

5 Conditional Rendering with Multiple Conditions

Ans →
- We can use else if, nested ternary operators, or separate components for multiple conditions.

Example :-
```
function Status({ status }) {
    if (status === "loading") return <p>Loading....</p>;
    if (status === "error") return <p> style={{color: 'red'}}>Error!</p>;
    if (status === "success") return <p style={{color: 'green'}}>Data Loaded!</p>;
    return null;
}
```

Output :-
Loading...
Error!
Data Loaded!

Different UI based on status

6 Common Use Cases

Ans →
- Authentication (Login / Logout)
- Loading and error states (API calls)
- Showing/hiding elements based on user role (Admin/User)
- Displaying empty states (e.g. no data found)
- Feature toggles
- Responsive UI (mobile/desktop)

Quick Tips :-
- Keep conditions simple and readable.
- Avoid nested ternary (hard to read).
- Use separate components for complex UI.
- Use meaningful variable names.
- Return null to render nothing.
- Combine with state, props or context.

Right UI at the Right Time Makes a Better User Experience !

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Lists & Keys

Render multiple elements efficiently

Lists Turn Data into Beautiful UIs !

1 Why Do We Need Lists ?

Ans →
- In real applications, we often work with arrays of data (e.g. users, products, posts, etc.).
- We can use the map() function to render a list of elements in React.
- Each item in a list must have a unique key to help React identify which items have changed.

Example Data (Array) :-

```
const students = [
  { id: 1, name: "Sovit" },
  { id: 2, name: "Smruti" },
  { id: 3, name: "Karyvio" }
];
```

We will render this array using map()

2 Rendering a List using map()

Ans →
- The map() function creates a new array by calling a function for each element.
- In React, we use map() to transform data into JSX elements.
- Always provide a unique key for each item.

Example :-

```
function StudentList() {
  const students = [
    { id: 1, name: "Sovit" },
    { id: 2, name: "Smruti" },
    { id: 3, name: "Karyvio" }
  ];

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

Output :-

- Sovit
- Smruti
- Karyvio

Key Point :-
- We use a unique key (here id) for each list item.
- Without a key, React will show a warning.

3 JSX with List of Objects

Ans →
- We can render more complex UI for each item (like cards, tables, etc.).
- Each element returned by map() can be a full component or a block of JSX.

Example :-

```
function UserList() {
  const users = [
    { id: 1, name: "Sovit", role: "Admin" },
    { id: 2, name: "Smruti", role: "Editor" },
    { id: 3, name: "Karyvio", role: "Learner" }
  ];
  return (
    <div>
      {users.map((user) => (
        <div key={user.id} className="user-card">
          <h3>{user.name}</h3>
          <p>Role: {user.role}</p>
        </div>
      ))}
    </div>
  );
}
```

Output (UI) :-

Sovit
Role: Admin

Smruti
Role: Editor

Karyvio
Role: Learner

4 Keys in React

Ans →
- Keys help React identify which items have changed, are added, or removed.
- Keys should be unique and stable (i.e. not changing on every render).
- Keys are only required when rendering lists (array of elements).

Best Practices for Keys :-
- ✅ Use a unique ID from your data (like user.id).
- ✅ Avoid using array index as key (only if no unique ID is available).
- ✅ Keys should be stable (not randomly generated).
- ✅ Do not use non-unique values.

Bad Example (Using Index as Key) :-

```
{users.map((user, index) => (
  <li key={index}>{user.name}</li>
))}
```

✗ Not Recommended !

5 Rendering a List of Components

Ans →
- Instead of writing JSX inside map(), we can also create a separate component for each item.
- This makes the code cleaner and reusable.

Separate Component for each item → Reusable & Clean ✓

```
function UserCard({ user }) {
  return (
    <div className="user-card">
      <h3>{user.name}</h3>
      <p>Role: {user.role}</p>
    </div>
  );
}
```

Using the Component :-

```
function UserList({ users }) {
  return (
    <div>
      {users.map((user) => (
        <UserCard key={user.id} user={user} />
      ))}
    </div>
  );
}
```

6 Common Use Cases

Ans →
- Rendering list of users / products
- Displaying comments
- Showing notifications
- Rendering menu items
- Displaying search results
- Building tables and data grids
- Rendering dynamic forms

Quick Tips :-
- Always use a unique and stable key.
- Keep list items as separate components when possible.
- Use meaningful variable names.
- Combine with conditional rendering for empty states.

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Events

Make your components interactive

User Actions
Make Your UI Alive !

1 What are Events in React ?

Ans →
- Events are actions that happen in the browser (like clicks, typing, submitting a form, etc).
- In React, we use event handlers to respond to these actions.
- Event handlers are written in camelCase (onClick, onChange, etc).
- We pass a function that will be called when the event occurs.

Common Events :-
- onClick
- onChange
- onSubmit
- onFocus
- onBlur
- onKeyDown / onKeyUp
- onMouseEnter / onMouseLeave
- onDoubleClick

Listen
Handle
Respond
Build Better
Experiences !

2 Handling a Click Event

Ans →
- Use the onClick event to handle button clicks.
- Pass a function (either a named function or an arrow function).
- We can also pass arguments to the function.

Example :-
```
function ButtonExample() {
  const handleClick = () => {
    alert("Hello from Karyvio!");
  };
  return (
    <button onClick={handleClick}>
      Click Me
    </button>
  );
}
```

Output (alert) :-

Key Point :-
- Event names are written in camelCase.
- We pass a function, not a function call.
- onClick={handleClick} ✓
  onClick={handleClick()} ✗
  (this will call the function immediately).

Passing Arguments :-
```
function Greet({ name }) {
  const sayHello = (user) => {
    alert(`Hello, ${user}!`);
  };
  return (
    <button onClick={() => sayHello(name)}>
      Greet {name}
    </button>
  );
}
<Greet name="Sovit" />
```

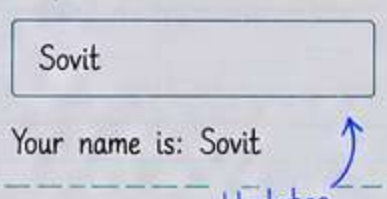
You can pass data to event handlers !

3 Handling Input Change (onChange)

Ans →
- Use onChange to handle input field changes.
- Commonly used in forms to update state.
- The event object (e) gives access to the input value.

Example :-
```
import { useState } from "react";
function InputExample() {
  const [name, setName] = useState("");
  const handleChange = (e) => {
    setName(e.target.value);
  };
  return (
    <div>
      <input
        type="text"
        pleceholder="Enter your name"
        value={name}
        onChange={handleChange}
      />
      p>Your name is: {name}</p>
  ); </div>
}
```

Output :-



Updates in real-time as you type !

Key Point :-
- e.target.value gives the current input value.
- onChange works with input, textarea, select, etc.

4 Form Submit Event (onSubmit)

Ans →
- Use onSubmit to handle form submission.
- We can prevent the default page reload using e.preventDefault().
- Useful for handling forms in React.

Example :-
```
function LoginForm() {
  const handleSubmit = (e) => {
    e.preventDefault();
    alert("Form Submitted!");
  };
  return (
    <form onSubmit={handleSubmit}>
      <input type="text" plecholder="Email" />
      <button type="submit">Submit</button>
    </form>
  );
}
```

5 Some More Useful Events

Ans

Event	Description	Example
onFocus	Triggered when input gets focus	<input onFocus={...} />
onBlur	Triggered when input loses focus	<input onBlur={...} />
onKeyDown	Triggered when a key is pressed	<input onKeyDown={...} />
onKeyUp	Triggered when a key is released	<input onKeyUp={...} />
onMouseEnter	Triggered when mouse enters	
onMouseLeave	Triggered when mouse leaves	
onDoubleClick	Triggered on double click	<button onDoubleClick={...} />

6 Event Object (e)

Ans →
- The event handler receives an event object (usually called e).
- It contains useful information about the event.
- Common properties:
  - e.target -> the element that triggered the event
  - e.target.value => current value (for inputs)
  - e.preventDefault() => prevents default behavior
  - e.stopPropogation() => stops event bubbling
  - e.key => key pressed (for keyboard events)
  - e.type => type of the event (e.g., "click")

Code Interacts, Ideas Come to Life !

Karyvio
LEARN | BUILD | GROW | GET HIRED

React useState

Add state to your components

State Makes Your UI Dynamic !

1 What is useState ?

Ans →

- useState is a built-in React Hook that lets us add state (variables) to functional components.
- It returns an array with two values:
  1. Current state value
  2. A function to update the state
- When the state updates, React re-renders the component with the new value.
- Syntax: const [state, setState] = useState(initialValue);

Syntax :-

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

- `count` → state variable (current value)
- `setCount` → function to update state

Update Re-render Repeat That's React !

2 Simple Counter Example

Ans →

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

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

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick<() => setCount(count + 1)}>
          Increment
      </button>
    </div>
  );
}
```

Output :-

Count: 0
[Increment]
↓
Count: 1
[Increment]

Key Points :-

- useState can be used multiple times in the same component.
- Each state is independent.
- Updating state triggers a re-render.
- Initial value can be any data type (string, number, boolean, array, object, etc).
- State updates are asynchronous.
- We should not directly modify the state value (it won't re-render).

3 Different Data Types

Ans

Data Type	Example	Code
Number	0	const [num, setNum] = useState(0)
String	"Hello"	const [text, setText] = useState("")
Boolean	false	const [isOpen, setIsOpen] = useState(false)
Array	[1, 2, 3]	const [items, setItems] = useState([])
Object	{ name: "Sovit" }	const [user, setUser] = useState({})

4 Updating State

Ans →

- Use the setter function returned by useState.
- You can update state based on the previous value using a callback function.

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

// Direct update
setCount(count + 1);

// Functional update (recommended when
// new state depends on previous state)
setCount(prev => prev + 1);
```

Why use functional update ?

- Avoids issues with stale state.
- Useful when multiple updates happen quickly.

5 Multiple States in One Component

Ans

```
function UserProfile() {
  const [name, setName] = useState("Sovit");
  const [age, setAge] = useState(25);
  const [isStudent, setIsStudent] = useState(true);

  return (
    <div>
      <p>Name: {name}</p>
      <p>Age: {age}</p>
      <p>Student: {isStudent ? "Yes" : "No"}</p>
    </div>
  );
}
```

Output :-

Name: Sovit
Age: 25
Student: Yes

Note :-

- You can have as many useState hooks as needed in a component.

6 Rules & Best Practices

Ans →

- ✅ Always call useState at the top level of the component (not inside loops or conditions).
- ✅ Use meaningful state names.
- ✅ Do not mutate the state directly.
- ✅ Use functional updates when new state depends on the previous state.
- ✅ Keep state as simple as possible.
- ✅ If state becomes complex, consider using useReducer.
- ✅ Avoid unnecessary state (derive values when possible).

7 Common Mistakes

Ans →

- ❌ Directly modifying state (e.g. count = 5) → won't re-render.
- ❌ Not using functional update when needed.
- ❌ Using state inside loops or conditions.
- ❌ Storing derived values in state (can be calculated instead).
- ❌ Forgetting that state updates are asynchronous.

8 Real World Example (Toggle)

Ans

```
function Toggle() {
  const [isOn, setIsOn] = useState(false);

  return (
    <button onClick<() => setIsOn(!isOn)}>
      {isOn ? "ON" : "OFF"}
    </button>
  );
}
```

Output :-

[OFF]
↓
[ON]

Simple but Powerful !

Karyvio.com

Keep Learning, Keep Building !

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Objects & Arrays in State

Manage complex data in a better way

Immutable Updates Make Your App Predictable !

1 Why Use Objects & Arrays in State ?

Ans →
- Real world applications deal with complex data (users, product lists, forms, settings, etc.).
- Objects help us store related data (e.g. user details).
- Arrays help us manage collections of data (e.g. todo list).
- We should always update them immutably (create new copy) instead of mutating the original state.

2 Important Rules

Ans →
- ✓ Never mutate state directly.
- ✓ Always create a new object/array using spread operator or methods.
- ✓ State updates are asynchronous.
- ✓ Keep your state structure simple and normalized.
- ✓ Use meaningful keys and IDs for array items.

Immutable Data = More Reliable UI

3 Working with Objects in State

Ans →

Example :- User Profile (Object)

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

function UserProfile() {
  const [user, setUser] = useState({
    name: "Sovit",
    age: 25,
    role: "Developer"
  });
  const updateName = () => {
    // create a new object with updated name
    setUser({ ...user, name: "Smruti" });
  };

  return (
    <div>
      <h3>User: {user.name}</h3>
      <p>Age: {user.age}</p>
      <p>Role: {user.role}</p>
      <button onClick={updateName}>Change Name
    </div>
  );
}
```

Output :-

Before Click:
User: Sovit
Age: 25
Role: Developer

After Click:
User: Smruti
Age: 25
Role: Developer

Key Point :-
- Use spread operator (....) to copy the previous object.
- Only update the field you want to change.

Don't mutate directly !

4 Updating Nested Objects

Ans →

Example :- Nested Object Update

```
const [user, setUser] = useState({
    name: "Sovit",
    address: { city: "Bhubaneswar", pin: "751001" }
});

const updateCity = () => {
  setUser({
    ...user,
    address: {
      ...user.address,
      city: "Puri"
    }
  });
};
```

Output :-

```
address: {
  city: "Puri",
  pin: "751001"
}
```

Key Point :-
- For nested objects, spread each level.
- Always create a new copy (don't mutate nested data).

5 Working with Arrays in State

Ans →

Example :- Todo List (Array)

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

function TodoList() {
  const [tasks, setTasks] = useState(["Learn React",
  "Build Projects"]);

  const addTask = () => {
    const newTask = prompt("Enter new task:");
    if (newTask) {
      setTasks([...tasks, newTask]); // add to array
  };

  return (
    <div>
      <h3>Tasks)>
      <ul>
        {tasks.map((task, index) => (
          <lii key={index}>{task}</li>
        ))}
      </ulo>
    <button onClick={addTask}>Add Task}>
  );
}
```

Output :-

Tasks
- Learn React
- Build Projects
- Make Portfolio

Key Point :-
- Use spread operator to add new items.
- Never use array index as key (if data can change).
- Always create a new array (don't mutate the original).

6 Common Array Operations

Ans →

Operation	Example Code	Description
Add item	setItems([...items, newItem])	Add to end
Remove item	setItems(items.filter(i => i.id !== id))	Remove by id
Update item	setItems(items.map(i => i.id === id ? {...i, name:"New"} : i))	Update specific item
Toggle items	setItems(items.map(i => i.id === id ? {...i, done: !i.done} : i))	Toggle a value
Clear all	setItems([])	Remove all items

7 Updating Arrays of Objects

Ans →

Example :- Toggle Complete

```
const [todos, setTodos] = useState([
    { id: 1, text: "Learn React", done: false },
    { id: 2, text: "Build Project", done: true },
]);
const toggleTodo = (id) => {
  setTodos(todos.map(todo =>
    todo.id === id ? {...todo, done: !todo.done} : todo
  ));
};
```

8 Common Mistakes

Ans →
- ✗ Mutating object directly (e.g. user.name = "John").
- ✗ Using push() to add items in array.
- ✗ Forgetting to create new object/array,
- ✗ Using index as key in dynamic lists.
- ✗ Shallow copying nested objects (can cause unexpected updates).

Best Practices :-
- ✓ Keep state immutable.
- ✓ Use spread operator or array methods.
- ✓ Keep state structure as simple as possible.
- ✓ Use unique and stable IDs.
- ✓ Normalize data for large applications.
- ✓ Update only what's necessary.
- ✓ Use meaningful variable names.

Small Steps Build Big Projects !

Karyvio.com

Better Code. Brighter Future !

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Forms

Handle user input like a pro

Forms Connect Users to Your App !

1 What are Forms in React ?

Ans →

- Forms allow users to input data like text, numbers, select options, etc.
- In React, we use controlled components to handle form data (using state).
- We capture the input values using onChange and handle form submission using onSubmit.
- Forms are commonly used for login, signup, contact forms, feedback forms, etc.

2 Controlled Components

Ans →

- In React, form inputs are controlled by state.
- The input value is always synced with the state variable.
- We update the state using onChange event.
- This gives us full control over the form data.
- It makes validation and error handling easier.

Collect Validate Process Build Real Apps !

3 Text Input Example

Ans

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

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

  return (
    <div>
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Enter your name"
      />
      <p>Your name is: {name}</p>
    </div>
  );
}
```

Output :-

Sovit

Your name is: Sovit

Key Point :-

- value binds the input to state.
- onChange updates the state.
- Always use controlled components in React.

4 Textarea Example

Ans →

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

function TextArea() {
  const [message, setMessage] = useState("");

  return (
    <div>
      <textarea
        value={message}
        onChange={(e) => setMessage(e.target.value)}
        rows={4}
        placeholder="Write your message..."
      />
      <p>Message: {message}</p>
    </div>
  );
}
```

Used for multi-line input like comments, feedback, messages, etc.

5 Select Dropdown Example

Ans

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

function SelectExample() {
  const [course, setCourse] = useState("");

  return (
    <div>
      <select value={course} onChange={(e)=> setCourse(e.target.value)}>
        <option value="">-- Select Course --</option>
        <option value="react">React</option>
        <option value="javascript">JavaScript</option>
        <option value="python">Python</option>
      </select>
      <p>Selected Course: {course}</p>
    </div>
);};
```

Output :-

React

Selected Course: React

Key Point :-

- Use value for each option.
- Keep a default "-- Select --" option.
- Handle the selected value using onChange.

6 Checkbox Example

Ans →

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

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

  return (
    <div>
      <label>
        <input
          type="checkbox"
          checked={isChecked}
          onChange={(e) => setIsChecked(e.target.checked)}
        /> I agree to the terms and conditions
      </label>
      <p>{isChecked ? "Checked" : "Not Checked"}</p>
    </div>
);};
```

7 Radio Button Example

Ans

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

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

  return (
    <div>
      <label>
        <input type="radio" value="male"
          checked={gender === "male"}
          onChange={(e) => setGender(e.target.value)}
        /> Male
      </label>
      <label>
        <input type="radio" value="female"
          checked={gender === "female"}
          onChange={(e) => setGender(e.target.value)}
        /> Female
      </label>
      <p>Selected: {gender}</p>
    </div>
  );
}
```

Output :-

Male Female

Selected: Male

Note :-

- Use the same name for radio buttons so that only one can be selected at a time.

Small Details Make Better Apps !

8 Form Submission (onSubmit)

Ans

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

function LoginForm() {
  const [email, setEmail] = useState("");
  const handleSubmit = (e) => {
    e.preventDefault(); // prevent page reload
    console.log("Form Submitted:", email);
    setEmail("");
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="email" value={email}
        onChange={(e) => setEmail(e.target.value)}
        placeholder="Enter your email" />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Output :-

test@example.com

Submit

(Check console)

Key Point :-

- Use onSubmit on the <form>.
- Always use e.preventDefault().
- Collect and validate data before sending to server.
- You can call an API here (like login/signup).

Form Inputs Today, Real Users Tomorrow !

Keep Practicing ☆

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Form Validation

Validate user input for better user experience

Good Validation = Happy Users

1 Why Form Validation ?

Ans →

- Prevents invalid or incorrect data from being submitted.
- Improves user experience with helpful error messages.
- Reduces errors and saves time (both for users and developers).
- Can validate on the client side (frontend) and also on the server side (backend).
- Ensures data is in the correct format (e.g., email, password, required fields).

2 Types of Validation

Ans →

- Required Field Validation (e.g. name, email)
- Format Validation (e.g. email, phone)
- Length Validation (e.g. password min 6 chars)
- Pattern Validation (e.g. only numbers)
- Real-time Validation (validate while typing)
- On Submit Validation (validate when form is submitted)
- Custom Validation (e.g. password match)

Validate Early
Avoid Errors Later

3 Basic Validation Example (Required Fields)

Ans

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

function SimpleForm() {
  const [name, setName] = useState("");
  const [error, setError] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    if (!name.trim()) {
      setError("Name is required!");
    } else {
      setError("");
      alert("Form Submitted: " + name);
    }
  };

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

Output :-

Enter your name

Name is required!

Submit

Key Point :-

- Use state to track input value and error message.
- Validate before submitting.
- Show clear and user-friendly error messages.

4 Email Validation Example

Ans

```
const [email, setEmail] = useState("");
const [error, setError] = useState("");

const validateEmail = (value) => {
  const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  if (!value) {
    return "Email is required!";
  } else if (!regex.test(value)) {
    return "Please enter a valid email!";
  }
  return "";
};
const handleChange = (e) => {
  setEmail(e.target.value);
  setError(validateEmail(e.target.value));
};

<input
  type="email"
  value={email}
  onChange={handleChange}
  placeholder="Enter your email"
/>
{error && <p className="error">{error}</p>}
```

Output :-

Enter your email

Please enter a valid email!

Tip :-

- Use regex for format validation.
- Show error in real-time (onChange).
- Also validate again on form submit.

5 Password Validation Example

Ans

```
const [password, setPassword] = useState("");
const [error, setError] = useState("");

const validatePassword = (value) => {
  if (value.length < 6) {
    return "Password must be at least 6 characters!";
  }
  return "";
};

<input
  type="password"
  value={password}
  onChange={(e) => {
    setPassword(e.target.value);
    setError(validatePassword(e.target.value));
  }}
/>
{error && <p className="error">{error}</p>}
```

Output :-

••••

Password must be at least 6 characters!

Key Point :-

- Use length check or regex.
- Don't show actual password.
- Combine multiple validations if needed (e.g. uppercase, number, special char).

Stronger Password
Stronger Security !

6 Multiple Field Validation Example

Ans

```
const [form, setForm] = useState({
  name: "",
  email: "",
  password: "",
});
const [errors, setErrors] = useState({});

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

const handleSubmit = (e) => {
  e.preventDefault();
  const newErrors = {};
  if (!form.name) newErrors.name = "Name is required!";
  if (!form.email) newErrors.email = "Email is required!";
  if (form.password.length < 6)
    newErrors.password = "Password must be 6+ chars!";

  setErrors(newErrors);
  if (Object.keys(newErrors).length === 0) {
    console.log("Form Submitted:", form);
  }
};
```

Form Fields :-

- name
- email
- password

Validation Flow :-

1. Check all fields
2. Store errors in state
3. Show error messages
4. Submit only if no errors

Validate All Fields for a Better User Experience

7 Client Side vs Server Side Validation

Ans

Feature	Client Side (Frontend)	Server Side (Backend)
Purpose	Quick feedback to user	Final data validation
Speed	Faster	Slower (network request)
Security	Can be bypassed	More secure
Implementation	React (JavaScript)	Node.js, Express, etc.
Best Practice	Use both	Use both

8 Best Practices & Common Mistakes

Ans

Best Practices :-

- ✓ Show clear and user-friendly error messages.
- ✓ Validate on both client and server.
- ✓ Use real-time validation (onChange).
- ✓ Disable submit button if form is invalid.
- ✓ Keep validation logic clean and reusable.
- ✓ Use meaningful field names and labels.
- ✓ Provide visual feedback (error styles).
- ✓ Handle edge cases (empty spaces, etc.).

Common Mistakes :-

- ✗ Not validating input fields.
- ✗ Using only client side validation.
- ✗ Giving unclear error messages.
- ✗ Not handling empty spaces (e.g. " ").
- ✗ Letting users submit invalid data.
- ✗ Overcomplicating validation logic.
- ✗ Forgetting to reset errors.
- ✗ Not validating on form submit.

Better Validation. Better Applications !

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Component Communication

Share data between components effectively

Components Talk Make Your App Interactive !

1 Why Component Communication ?

Ans →
- Real world applications have multiple components.
- Components need to share data and communicate with each other.
- Helps in building dynamic and interactive UIs.
- Makes components reusable and maintainable.
- There are different ways to communicate depending on the relationship between components.

Different Ways to Communicate :-

1. Parent → Child (using props)
2. Child → Parent (using callbacks)
3. Sibling Components (lifting state up)
4. Context API (for global communication)
5. External state management (Redux, Zustand, etc.)

Right Data Right Component Better Apps !

2 Parent → Child Communication (Props)

Ans → Parent passes data to child using props.

```
// Parent Component
function App() {
  const name = "Sovit";
  return (
    <div>
      <Welcome name={name} />
    </div>
  );
}

// Child Component
function Welcome({ name }) {
  return <h2>Hello, {name}!</h2>;
}
```

Output :-

Hello, Sovit!

Key Point :-
- Pass data using props.
- Child component receives it as a parameter.
- Props are read-only (cannot be modified).

3 Child → Parent Communication (Callback)

Ans → Child can send data to parent using a function passed via props.

```
// Parent Component
function App() {
  const handleMessage = (msg) => {
    alert("Message from child: " + msg);
  };
  return <Child onSend={handleMessage} />;
}

// Child Component
function Child({ onSend }) {
  return (
    <button onClick={() => onSend("Hello!")}>
      Send Message to Parent
    </button>
  );
}
```

What Happens ?
- Parent passes a function.
- Child calls the function (e.g. onClick).
- Parent receives the data.

Output :-

(On button click)

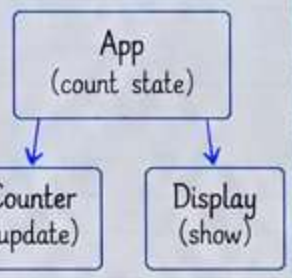
Message from child: Hello!

4 Sibling Component Communication (Lifting State Up)

Ans →
- Sibling components cannot directly communicate.
- We lift the state to their common parent.
- Then pass data and callbacks to both siblings.

```
function App() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <Counter count={count} setCount={setCount} />
      <Display count={count} />
    </div>
  );
}

function Counter({ count, setCount }) {
  return (
    <button onClick={() => setCount(count + 1)}>
      Increment
    </button>
  );
}
function Display({ count }) {
  return <p>Count: {count}</p>;
}
```

Component Tree :-



Key Point :-
- Common parent manages the state.
- Pass state and functions to siblings.
- Keeps components in sync.

5 Using Context API (Global Communication)

Ans →
- Useful when many components need the same data.
- Avoids prop drilling (passing props through many levels).
- Commonly used for theme, authentication, user data, etc.

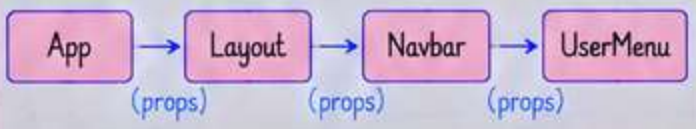
```
import { createContext, useContext, useState } from "react";
const UserContext = createContext();

function App() {
  const [user, setUser] = useState("Sovit");
  return (
    <UserContext.Provider value={user}>
      <Navbar />
      <Profile />
    </UserContext.Provider>
  );
}

function Navbar() {
  const user = useContext(UserContext);
  return <h3>Welcome, {user}</h2>;
}
```

When to use ?
- Authenticated user info
- Theme (light/dark)
- Language settings
- Global app state

Advantage :-
- No prop drilling
- Easy to manage global state
- Clean and simple code

6 Prop Drilling Problem

Ans →
- Passing props through many nested components can become difficult to maintain.
- Context API or state management libraries help solve this problem.



Too many levels → Hard to maintain → Use Context

7 Best Practices

Ans →
- ✓ Keep components small and focused.
- ✓ Use props for parent → child communication.
- ✓ Use callbacks for child → parent communication.
- ✓ Use Context API for global data.
- ✓ Avoid unnecessary prop drilling.
- ✓ Choose the right method based on the use case.
- ✓ Keep data flow predictable.
- ✓ Follow a clear component structure.

8 Common Mistakes

Ans →
- ✗ Trying to directly access sibling component's state.
- ✗ Mutating props inside child component.
- ✗ Overusing Context API for small apps.
- ✗ Passing too many props (prop drilling).
- ✗ Not using proper component structure.
- ✗ Making components too tightly coupled.

Karyvio
LEARN | BUILD | GROW | GET HIRED

React useEffect

Handle side effects in functional components

Side Effects Make Your App More Powerful !

1 What is useEffect ?

Ans →

- useEffect is a React Hook that lets us perform side effects in functional components.
- Side effects are operations that happen outside the normal rendering process.
- Examples: API calls, timers, event listeners, DOM updates, subscriptions, etc.
- It runs after the component is rendered to the screen.

2 Syntax of useEffect

Ans →

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

useEffect(() => {
  // side effect logic here
  return () => {
    // optional cleanup
  };
}, [dependencies]);
```

Parameters :-

- First argument: a function to run after render.
- Second argument (optional): dependency array.

3 Example: Run Effect on Every Render

Ans →

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

function App() {
  useEffect(() => {
    console.log("Component rendered!");
  }); // no dependency array

  return <h1>Check Console</h1>;
}
```

Output :-

"Component rendered!" (prints after every render)

Note :-

- Runs after every render (initial + re-renders).
- Useful for debugging.

4 Run Effect Only Once (on Mount)

Ans →

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

function App() {
  useEffect(() => {
    console.log("Runs only once!");
  }, []); // empty dependency array

  return <h1>Welcome to React</h1>;
}
```

Output :-

"Runs only once!"

Use Case :-

- API calls to fetch data on page load.
- Initialize third-party libraries.
- Set up event listeners.

5 Run Effect on Dependency Change

Ans →

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

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

  useEffect(() => {
    console.log("Count changed:", count);
  }, [count]); // runs when count changes

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

Output :-

"Count changed: 1"
"Count changed: 2"
... (whenever count updates)

Key Point :-

- useEffect only runs when the values in the dependency array change.
- Keep the dependency array minimal and relevant.

6 Cleanup Function

Ans →

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

function Timer() {
  useEffect(() => {
    const interval = setInterval(() => {
      console.log("Timer running...");
    }, 1000);

    // cleanup function
    return () => {
      clearInterval(interval);
      console.log("Timer stopped!");
    };
  }, []);

  return <h1>Open console and unmount</h1>;
}
```

When does cleanup run?

- Before the component unmounts.
- Before the effect runs again (if dependencies change).
- Useful to clean up timers, subscriptions, event listeners, etc.

Real World Use Case :-

- Clean up prevents memory leaks and unnecessary background processes.

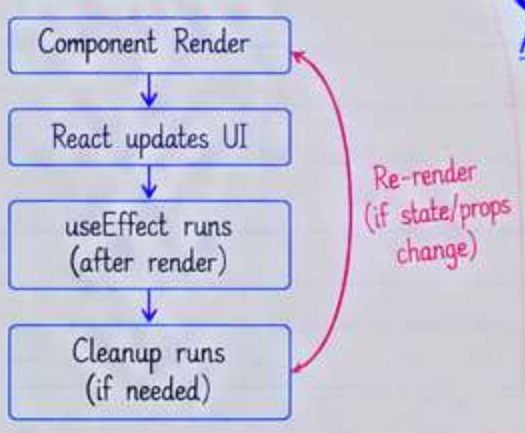
7 Common Use Cases

Ans →

- ✅ Fetching data from an API.
- ✅ Setting up event listeners (scroll, resize, etc.).
- ✅ Timers and intervals.
- ✅ Subscribing to WebSocket or other services.
- ✅ Updating the document title.
- ✅ Working with browser localStorage.
- ✅ Integrating third-party libraries (charts, maps, etc.).

8 useEffect Flow

Ans →



9 Best Practices

Ans →

- ✅ Use the right dependency array.
- ✅ Keep effects focused and small.
- ✅ Clean up when necessary.
- ✅ Avoid unnecessary re-renders.
- ✅ Use functional updates if needed.
- ✅ Handle errors in async effects.
- ✅ Keep side effects outside of rendering logic.

10 Common Mistakes

Ans →

- ❌ Forgetting the dependency array (causes infinite renders).
- ❌ Adding unnecessary dependencies.
- ❌ Not cleaning up timers or event listeners.
- ❌ Updating state inside useEffect without condition.
- ❌ Using useEffect for things that don't need side effects (e.g., simple calculations).

Use useEffect Wisely. It's Powerful, but Easy to Misuse !

"Good developers handle side effects. Great developers handle them cleanly."
- Karyvio

Karyvio
LEARN | BUILD | GROW | GET HIRED

React useEffect Deep Dive

Understand behavior, dependencies, cleanup and common pitfalls

Small Effects Big Impact !

1 How Dependency Array Works ?

Ans →
- React compares the values in the dependency array using shallow comparison (===).
- If any value changes, the effect runs again.
- If the array is empty [], it runs only once.
- If no array is provided, it runs after every render.
- For objects, arrays and functions, the reference must be the same to avoid re-runs.

2 Different Behaviors with Dependencies

Dependency Array	When it Runs	Use Case
No array	After every render	Logging, debugging
[]	Only once (on mount)	Initial data fetch
[value]	When value changes	React to state/prop change
[a, b]	When a or b changes	Multiple dependencies
[object/function]	When reference changes	Use useMemo/useCallback to prevent unnecessary runs

3 Stale Values Problem

Ans →
- Effects capture the values from the render in which they were created.
- If you don't include a variable in the dependency array, the effect will use a stale (old) value.
- This can lead to bugs, especially with timers, subscriptions and async code.

Example :- Stale Value (Wrong)

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

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

  useEffect(() => {
    const id = setInterval(() => {
      console.log("Count:", count); // always 0
    }, 1000);
    return () => clearInterval(id);
  }, []); // ✗ count not in dependencies

  return <h1>{count}<h1>;
}
```

Problem :-

✗ count is not updated inside effect. It always logs the initial value (0).

This is called a stale closure !

4 Fixing Stale Values

Solution 1: Add to dependencies

```
useEffect(() => {
  const id = setInterval(() => {
    console.log("Count:", count);
  }, 1000);
  return () => clearInterval(id);
}, [count]); // ✓ now logs latest count
```

Solution 2: Use functional update (for state updates in timers)

```
useEffect(() => {
  const id = setInterval(() => {
    setCount(c => c + 1);
  }, 1000);
  return () => clearInterval(id);
}, []); // ✓ no dependency needed
```

When to use each ?
- Use dependencies when you need the latest value inside the effect.
- Use functional updates when you only need to update state based on previous value.
- For complex logic, consider useRef to store mutable values.

5 Infinite Loop Problem

Ans →
- Occurs when the effect updates a state that is also in the dependency array, causing continuous re-renders.

Example :- Infinite Loop (Wrong)

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

useEffect(() => {
  setCount(count + 1);
}, [count]); // ✗ runs infinitely
```

How to Fix ?
- Check if state update is really needed.
- Use condition inside the effect.
- Use functional update or ref if appropriate.

6 Cleanup Timing & Behavior

Ans →
- Cleanup function runs before the next effect execution (when dependencies change).
- Also runs when the component unmounts.
- Used to remove event listeners, clear timers, cancel API requests, close subscriptions, etc.

Example :- Cleanup with Event Listener

```
useEffect(() => {
  const handleResize = () => {
    console.log("Window resized");
  };
  window.addEventListener("resize", handleResize);
  return () => {
    window.removeEventListener("resize", handleResize);
    console.log("Cleanup: removed event listener");
  };
}, []);
```

Key Point :-

Cleanup runs before re-run and on unmount (to avoid memory leaks).

Clean code Happy app !

7 Real World Example (Fetch with Cleanup)

Ans →

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

function UserData({ userId }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    const controller = new AbortController();

    fetch("https://jsonplaceholder.typicode.com/users/${userId}", {
      signal: controller.signal,
    })
      .then(res => res.json())
      .then(data => setUser(data))
      .catch(err => {
        if (err.name !== "AbortError") console.error(err);
      });

    return () => {
      controller.abort(); // cancel fetch if component unmounts
    };
  }, [userId]);

  return <div>{user ? <h3>{user.name}</h3> : "Loading..."}</div>;
}
```

Why use AbortController ?
- If component unmounts before the API response arrives, we cancel the request.
- Prevents memory leaks and unwanted state updates.
- Important for real applications (search, admin panels, etc.).

Handle async effects carefully !

8 Best Practices & Common Mistakes

Ans →

Best Practices :-
- ✓ Always include all necessary dependencies.
- ✓ Use cleanup for timers, listeners, subscriptions, etc.
- ✓ Use useCallback / useMemo for stable references.
- ✓ Avoid unnecessary effects (keep logic simple).
- ✓ Handle errors in async effects.
- ✓ Use AbortController for API requests.
- ✓ Think about unmounting scenarios.
- ✓ Use ESLint (react-hooks/exhaustive-deps) to catch missing dependencies.

Common Mistakes :-
- ✗ Missing dependencies (causes stale values).
- ✗ Unnecessary dependencies (causes extra re-renders).
- ✗ Updating state inside effect without condition (infinite loop).
- ✗ Not cleaning up (memory leaks).
- ✗ Using objects/functions directly in dependency array.
- ✗ Ignoring errors in async code.
- ✗ Using useEffect for logic that doesn't need side effects.

"Understand useEffect deeply, and you control how your app lives !"

Better Understanding → Better React Apps !

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Hooks

Powerful tools to use state and other React features

Hooks Make Functional Components Powerful !

1 What are React Hooks ?

Ans →
- Hooks are special functions that let you use React features (like state, effects, context, etc.) in functional components.
- Introduced in React 16.8 (2019).
- Hooks allow you to "hook into" React's internal features from function components.
- Hooks follow specific rules.
- They make code more reusable, clean and easier to manage.

2 Why Use Hooks ?

Ans →
- Use state and lifecycle features without class components.
- Cleaner and shorter code.
- Better reusability through custom hooks.
- Easier logic sharing between components.
- Actively maintained by the React team.
- More composable and testable code.
- Works well with modern React features (like Concurrent Mode).

Remember :-
Hooks don't replace React, they make React better !

3 Rules of Hooks

Ans →
1. Only call hooks at the top level (not inside loops, conditions or nested functions).
2. Only call hooks from React functional components or custom hooks.
3. Hooks must be called in the same order on every render.
4. You cannot call hooks inside regular JavaScript functions.

❌ Invalid Example (Don't Do This)

```
function BadComponent() {
  if (Math.random() > 0.5) {
    const [count, setCount] = useState(0); // ❌
  }
  return <div>Bad</div>;
}
```

✅ Valid Example

```
function GoodComponent() {
  const [count, setCount] = useState(0);
  if (count > 0) {
    return <div>{count}</div>;
  }
  return <div>0</div>;
}
```

Always call hooks at the top level !

4 Built-in Hooks (Overview)

Ans

Hook	Purpose
useState	Manage state in functional components
useEffect	Handle side effects (API calls, timers, etc.)
useContext	Access context values
useRef	Reference DOM elements or store mutable values
useMemo	Memoize expensive calculations
useCallback	Memoize function references
useReducer	Manage complex state with reducer pattern
useImperativeHandle	Customize the instance value exposed by ref
useLayoutEffect	Like useEffect but runs before paint
useId	Generate unique IDs (React 18+)
useTransition	Handle non-urgent state updates (React 18+)
useDeferredValue	Defer re-rendering for better performance (React 18+)

5 Example: Using Multiple Hooks

Ans

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

function Profile() {
  const [name, setName] = useState("Sovit");
  const [count, setCount] = useState(0);
  const inputRef = useRef(null);

  useEffect(() => {
    document.title = `Hello ${name}`;
  }, [name]);

  return (
    <div>
      <h2>{name}</h2>
      <input ref={inputRef} placeholder="Type here" />
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
    </div>
  );
}
```

Output :-

Hello Sovit
Type here
Count: 0
Increment

Key Point :-
- useState => state
- useEffect => side effect
- useRef => reference
- Multiple hooks can be used together in one component.

Combine Hooks to Build Powerful Apps !

6 Custom Hooks (Introduction)

Ans →
- Custom hooks are JavaScript functions that use other hooks.
- They help you reuse stateful logic across components.
- Custom hooks must start with "use" (e.g., useFetch, useAuth).
- Useful for sharing logic like API calls, authentication, form handling, etc.

Example: useCounter.js

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

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

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

  return { count, increment, decrement, reset };
}

export default useCounter;
```

Usage :-

```
import useCounter from
  "./useCounter";

function App() {
  const { count, increment } =
    useCounter(0);
  return (
    <div>
      <p>{count}</p>
      <button onClick={increment}>
        +1
      </buttoun>
    </div>
  );
}
```

7 When to Use Custom Hooks ?

Ans →
- When you have reusable logic (API calls, forms, auth, etc.).
- When the same logic is used in multiple components.
- To keep components clean and focused.
- To separate business logic from UI.
- To make code more maintainable and testable.

Same Logic Different Components = Custom Hook !

8 Best Practices

Ans →
- ✅ Follow the rules of hooks.
- ✅ Keep custom hooks small and focused.
- ✅ Use meaningful names (starting with "use").
- ✅ Prefer built-in hooks before creating custom ones.
- ✅ Extract complex logic into custom hooks.
- ✅ Keep UI and logic separate.
- ✅ Write reusable and well-documented hooks.
- ✅ Test your custom hooks.

9 Common Mistakes

Ans →
- ❌ Calling hooks inside conditions or loops.
- ❌ Not following the naming convention (for custom hooks).
- ❌ Using hooks in regular JavaScript functions.
- ❌ Creating overly large custom hooks.
- ❌ Not handling dependencies properly.
- ❌ Misusing useEffect for non-side effects.
- ❌ Forgetting cleanup in effects.
- ❌ Not reusing hooks when possible.

Write Less Code Do More with Hooks !

Karyvio
LEARN | BUILD | GROW | GET HIRED

React useRef

Access and persist values without re-rendering

Small Hooks Big Possibilities

1 What is useRef ?

Ans →

- useRef is a React Hook that lets you create a mutable reference to a value or a DOM element.
- The value inside a ref persists across re-renders.
- Updating the ref value does NOT cause a re-render.
- Commonly used to access DOM elements, store mutable values, or keep previous values.
- Syntax: const ref = useRef(initialValue);

2 When to Use useRef ?

Ans →

- To access and manipulate DOM elements.
- To store mutable values (like timers, IDs, previous values).
- When you need a value to persist across renders but don't want to trigger a re-render.
- To integrate with third-party libraries.
- To keep track of previous values.
- To avoid re-creating values on every render.

Quick Note :-

useRef is like a "box" that holds a value, and React doesn't watch the box for changes

3 Basic Example (DOM Reference)

Ans →

```jsx
import { useRef } from "react";

function App() {
  const inputRef = useRef(null);
  const handleClick = () => {
    inputRef.current.focus();
  };
  return (
    <div>
      <input ref={inputRef} type="text"
        placeholder="Click the button" />
      <button onClick={handleClick}>
        Focus Input
      </button>
    </div>
  );
}
```

Output :-

Click the button [Focus Input]

Key Point :-

- ref={inputRef} attaches the ref to the DOM element.
- inputRef.current gives access to the actual DOM node.
- We can call DOM methods like focus(), scrollIntoView(), etc.

4 Storing Mutable Values (No Re-render)

Ans →

```jsx
import { useRef } from "react";

function App() {
  const countRef = useRef(0);

  const handleClick = () => {
    countRef.current += 1;
    console.log("Count:", countRef.current);
  };
  return (
    <div>
      <button onClick={handleClick}>
        Increment (Check Console)
      </button>
    </div>
  );
}
```

Console Output :-

```
Count: 1
Count: 2
Count: 3
...
```

Note :-

- Updating countRef.current does NOT re-render the component.
- Useful for values like timers, counters, or previous values.

5 useRef vs useState

Ans →

Feature	useRef	useState
Purpose	Store mutable value	Store state value
Triggers re-render	No	Yes
Syntax	useRef(initialValue)	useState(initialValue)
Value access	ref.current	state variable
Best for	DOM access, timers, previous values	UI data that changes
Example	const ref = useRef(0)	const [count, setCount] = useState(0)

6 Keeping Previous Value

Ans →

```jsx
import { useRef, useEffect } from "react";

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

  useEffect(() => {
    prevCount.current = count;
  }, [count]);

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

Output (after clicks) :-

```
Current: 2
Previous: 1
```

How it works ?

- useRef stores the previous value.
- We update it inside useEffect after each render.
- On the next render, it shows the last value.

Useful for comparing values !

7 Other Useful Examples

Ans →

1. Storing a Timer ID

```jsx
import { useRef, useEffect } from "react";
function App() {
  const timerRef = useRef();
  const startTimer = () => {
    timerRef.current = setInterval(() => {
      console.log("Tick....");
    }, 1000);
  };

  const stopTimer = () => {
    clearInterval(timerRef.current);
  };

  return (
    <div>
      <button onClick={startTimer}>Start</button>
      <button onClick={stopTimer}>Stop</button>
    </div>
  );
}
```

2. Integrating with Third-party Library

```jsx
import { useRef, useEffect } from "react";

function App() {
  const divRef = useRef();

  useEffect(() => {
    // Example: Initialize a library
    // SomeLibrary.init(divRef.current);
  }, []);

  return <div ref={divRef}>Map goes here</div>;
}
```

Use Cases :-

- Focus input fields
- Scroll to elements
- Store timers / intervals
- Keep previous values
- Integrate third-party libraries (charts, maps, etc.)
- Cache values without re-render

8 Best Practices

Ans →

- ✓ Use refs only when needed (not for everything).
- ✓ Prefer useState for UI-related data.
- ✓ Use meaningful names (e.g., inputRef, timerRef).
- ✓ Avoid reading/writing refs during render.
- ✓ Use refs to store mutable values like timers, IDs, previous values.
- ✓ Clean up timers or subscriptions when component unmounts.

9 Common Mistakes

Ans →

- ✗ Expecting re-render after updating ref.
- ✗ Using refs for UI data (useState instead).
- ✗ Accessing ref.current before the element is mounted (can be null).
- ✗ Forgetting to clean up timers.
- ✗ Modifying refs inside render method.
- ✗ Using the same ref for multiple elements.

"Refs keep values in memory, not in the UI. Use them wisely !"

Less Re-renders, More Control !

Karyvio
LEARN | BUILD | GROW | GET HIRED

React useMemo & useCallback

Optimize performance with memoization

Optimize Only When Needed Not Prematurely

1 Why Optimization is Needed ?

Ans →
- React re-renders components when state or props change.
- Sometimes, expensive calculations or function re-creations can affect performance.
- useMemo and useCallback help to avoid unnecessary re-computations and re-renders.
- Use them for performance optimization, not as a default for every component.

2 What are useMemo and useCallback ?

Ans →
- useMemo → memoizes the result of an expensive calculation.
- useCallback → memoizes a function reference.
- Both hooks store values in memory and only re-compute when dependencies change.
- Helps in preventing unnecessary re-renders, especially in child components (e.g., with React.memo).

Better Performance Better User Experience

3 useMemo - Memoize Values

Ans →
- useMemo is used to memoize the result of a computation.
- It only recalculates the value when its dependencies change.
- Useful for expensive calculations (e.g., filtering large lists, complex math, etc.).

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

function App() {
  const [count, setCount] = useState(0);
  const [items, setItems] = useState([1, 2, 3, 4, 5]);

  // Expensive calculation
  const sum = useMemo(() => {
    console.log("Calculating sum...");
    return items.reduce((acc, curr) => acc + curr, 0);
  }, [items]);

  return (
    <div>
      <p>Sum: {sum}</p>
      <button onClick={() => setCount(count + 1)}>
        Re-render ({count})
      </button>
    </div>
  );
}
```

Output (Console) :-

```
Calculating sum...
(only runs when
items change,
not on every render)
```

Key Point :-
- Syntax: useMemo(() => value, [dependencies])
- Returns the memoized value.
- Recomputes only when dependencies change.
- Helps in avoiding expensive calculations during re-renders.

4 useCallback - Memoize Functions

Ans →
- useCallback is used to memoize a function reference.
- It returns the same function instance until its dependencies change.
- Useful when passing functions to child components (e.g., with React.memo) to prevent unnecessary re-renders.

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

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

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

  const handleClick = useCallback(() => {
    console.log("Button clicked");
  }, []); // function instance remains same

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

Output (Console) :-

```
Child rendered
(only once, not
on every re-render)
```

Key Point :-
- Syntax: useCallback(() => { ....}, [dependencies])
- Returns a memoized function.
- Keeps the same function reference until dependencies change.
- Useful to prevent unnecessary re-renders in child components.

5 When to Use useMemo vs useCallback ?

Ans →

Feature	useMemo	useCallback
Purpose	Memoize a value (result of computation)	Memoize a function reference
Returns	A memoized value	A memoized function
Use case	Expensive calculations (e.g., filtering, sorting)	Passing functions to child components
Syntax	useMemo(() => value, [deps])	useCallback(() => {}, [deps])
Example	const val = useMemo(...)	const fn = useCallback(...)

6 Real World Example - Filtered List (useMemo)

Ans →

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

function App() {
  const [search, setSearch] = useState("");
  const [users] = useState([
    { id: 1, name: "Sovit" },
    { id: 2, name: "Smruti" },
    { id: 3, name: "Karyvio" },
  ]);

  const filteredUsers = useMemo(() => {
    console.log("Filtering...");
    return users.filter(user =>
      user.name.toLowerCase().includes(search.toLowerCase())
    );
  }, [search, users]);

  return (
    <div>
      <input
        type="text"
        value={search}
        onChange={(e) => setSearch(e.target.value)}
        placeholder="Search...";
      />
      <ul>
        {filteredUsers.map(user => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
    </div>
  );
}
```

Try removing useMemo and see the difference in console logs

Use Cases :-
- Search and filtering
- Sorting large lists
- Heavy calculations (charts, analytics)
- Preventing re-renders in child components
- Optimizing event handlers

7 Best Practices

Ans →
- ✅ Use useMemo and useCallback only when needed.
- ✅ Identify performance bottlenecks before optimizing.
- ✅ Use correct dependency arrays.
- ✅ Keep dependencies minimal and relevant.
- ✅ Use React.memo with useCallback for child components.
- ✅ Avoid unnecessary re-rreations of objects/functions.
- ✅ Use meaningful variable names and comments.

8 Common Mistakes

Ans →
- ❌ Using useMemo/useCallback everywhere (premature optimization).
- ❌ Missing dependencies in the array.
- ❌ Using useMemo for simple/cheap calculations.
- ❌ Creating new objects/functions inside the component.
- ❌ Not understanding when re-renders actually happen.

Optimize Smartly, Not Prematurely !

Learn Today, Build Tomorrow !

Karyvio.com

Clean Code → Better Performance = Happier Users !

Karyvio
LEARN | BUILD | GROW | GET HIRED

React — React.memo + Performance

Prevent unnecessary re-renders and build faster React apps

Optimize Renders Build Better Apps !

1 What is React.memo ?

- React.memo is a higher-order component that memoizes a functional component.
- It prevents re-rendering when the props have nat changed.
- It only works for functional components (not class components).

Syntax : const Memoized = React.memo(Component);

2 Why is it Needed ?

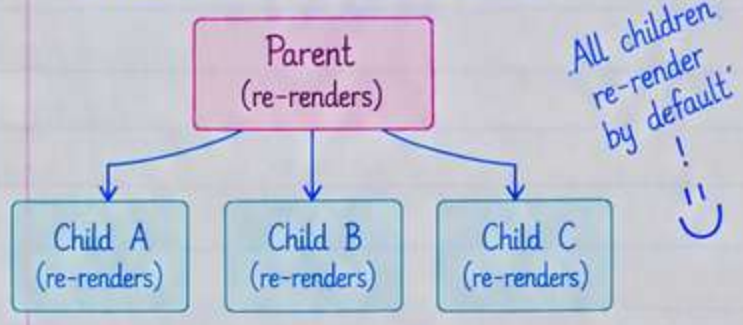
- In React, a component re-renders when its parent re-renders (even if props didn't change).
- Unnecessary re-renders can affect performance, especially in large applications.
- React.memo helps to skip re-rendering if props remain the same (using shallow comparison).

Key Benefit :-

- ✅ Better Performance
- ✅ Faster UI Updates
- ✅ Less Unnecessary Renders
- ✅ Improved User Experience

3 How Re-renders Happen ?

- When a parent component's state changes, all its child components re-render by default.
- Even if a child receives the same props, it still re-renders (normal behavior).



4 Basic Example of React.memo

```jsx
import React from "react";

const UserCard = ({ name, age }) => {
  console.log("UserCard rendered");
  return (
    <div className="card">
      <h3>{name}</h3>
      <p>Age: {age}</p>
    </div>
  );
};

export default React.memo(UserCard);
```

Note :-

- The component will only re-render if name or age actually changes.
- React performs a shallow comparison of props.

Output (Console) :-

UserCard rendered (first render)
(no log when parent updates with same props)
UserCard rendered (when props change)

5 Shallow Comparison of Props

- React.memo uses shallow comparison (to check if props have changed).
- It compares the previous props and new props using === (shallow equality).
- For objects and arrays, it only checks the reference, not the deep values.
- If you pass a new object/array (even with same values), it will still re-render.

Example :-

```jsx
const user = { name: "Sovit" };

return <UserCard
      user={user} />;
```

Above will re-render every time (because a new object is created).

Fix :-

Use useMemo to memoize objects or use primitive props.

```jsx
const user = useMemo(
  () => ({ name: "Sovit" }),
  []
);
return <UserCard user={user} />;
```

6 Custom Comparison (Advanced)

- By default, React.memo does a shallow comparison. You can also provide a custom comparison function.

```jsx
const areEqual = (prevProps, nextProps) => {
  return prevProps.id === nextProps.id;
};

export default React.memo(UserCard, areEqual);
```

- Useful when you want more control over when to re-render.

7 React.memo vs useMemo vs useCallback

Feature	React.memo	useMemo	useCallback
Purpose	Memoize component	Memoize a value	Memoize a function
Used for	Functional components	Expensive calculations	Function references
Syntax	React.memo(Component)	useMemo(() => value, [])	useCallback(() => {}, [])
Returns	Memoized component	Memoized value	Memoized function
When to use	Prevent re-render	Cache expensive result	Pass stable function to child components

8 Profiling Performance

- Use React DevTools Profiler to check which components are re-rendering.
- It shows render times and helps identify performance issues.
- Don't optimize blindly — measure first !

Measure Before Optimizing !

Steps :-

1. Open React DevTools (Profiler tab)
2. Record an interaction
3. Check which components re-rendered
4. Look for unnecessary renders
5. Apply React.memo / useMemo / useCallback

9 When to Use React.memo ?

- ✅ When the component is pure (only depends on props).
- ✅ When it re-renders often with the same props.
- ✅ For large lists or complex UI components.
- ✅ When performance is a concern.
- ✅ Combine with useMemo / useCallback for better results.

10 Best Practices

- ✅ Use React.memo only when needed.
- ✅ Keep components small and focused.
- ✅ Memoize objects, arrays, and functions when passing as props.
- ✅ Use meaningful prop names.
- ✅ Profile before and after optimization.
- ✅ Avoid premature optimization.

11 Common Mistakes

- ❌ Using React.memo everywhere (premature optimization).
- ❌ Passing new object/array/function props on every render.
- ❌ Not using keys in lists (causes unnecessary re-renders).
- ❌ Forgetting to memoize functions (use useCallback).
- ❌ Expecting deep comparison (React.memo only does shallow).
- ❌ Not profiling to check actual performance impact.

Write Clean Components
Optimize Wisely.
Build Faster Apps !

"Performance is not a feature, it's a better user experience."

Karyvio
LEARN | BUILD | GROW | GET HIRED

React useReducer

Manage complex state with a reducer function

Better State Management For Complex Logic

1 What is useReducer ?

Ans →
- useReducer is a React Hook for managing complex state logic.
- It works similar to Redux, but within a component.
- It uses a reducer function and dispatches actions to update the state.
- It is a good alternative to useState when state logic becomes complex (many conditions).
- Returns [state, dispatch].

2 When to Use useReducer ?

Ans →
- When state has multiple related values.
- When state logic is complex (many conditions).
- When next state depends on previous state.
- When you have actions (like in Redux).
- When the same state logic is used in multiple places (can be extracted to a custom hook).
- Not necessary for simple state (useState is enough).

Good Use Cases :-
- To-do apps
- Forms with multiple fields
- Complex UI states
- Cart / e-commerce logic
- Multi-step flows
- Global-like state in a component

3 Basic Syntax

Ans →

```
const [state, dispatch] = useReducer(reducer, initialState);
```

state → current state value
dispatch → function to send actions
reducer → function that defines how state changes
initialState → initial state value

Syntax of Reducer Function :-

```
function reducer(state, action) {
  switch (action.type) {
    case "ACTION_TYPE":
      return newState;
    default:
      return state;
  }
}
```

4 Simple Counter Example

Ans →

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

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1 };
    case "DECREMENT":
      return { count: state.count - 1 };
    case "RESET":
      return { count: 0 };
    default:
      return state;
  }
}

function App() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <h3>Count: {state.count}</h3>
      <button onClick={() => dispatch({ type: "INCREMENT" })}>
        +
      </button>
      <button onClick={() => dispatch({ type: "DECREMENT" })}>
        -
      </button>
      <button onClick={() => dispatch({ type: "RESET" })}>
        Reset
      </button>
    </div>
  );
}
```

Output :-

Count: 0
[+] [-] [Reset]

Note :-
- We use action objects with a type property.
- Reducer function decides how state changes based on action type.
- State is immutable (we return new state).

5 Managing Complex State (Todo App)

Ans →

```
const initialState = {
  todos: [],
  filter: "all"
};

function todoReducer(state, action) {
  switch (action.type) {
    case "ADD_TODO":
      return { ...state, todos: [...state.todos, action.payload] };
    case "REMOVE_TODO":
      return { ...state, todos: state.todos.filter(
        (todo, index) => index !== action.payload)
      )};
    case "SET_FILTER":
      return { ...state, filter: action.payload };
    default:
      return state;
  }
}
```

Dispatching Actions :-

```
dispatch({
  type: "ADD_TODO",
  payload: "Learn React"
});

dispatch({
  type: "REMOVE_TODO",
  payload: 0
});

dispatch({
  type: "SET_FILTER",
  payload: "completed"
});
```

useReducer is great for handling multiple related state values like todos, filters, UI states, etc.

6 useState vs useReducer

Ans →

Feature	useState	useReducer
Best for	Simple state	Complex state
State logic	Direct update	Reducer function
Syntax	const [state, setState]	const [state, dispatch]
Handles multiple values	Can be messy	Easier with actions
Similar to Redux	No	Yes
When to use	Simple UI state	Complex state logic

7 Real World Example (Form with useReducer)

Ans →

```
const initialState = { name: "", email: "", error: "" };

function formReducer(state, action) {
  switch (action.type) {
    case "UPDATE_FIELD":
      return { ...state, [action.field]: action.value };
    case "SET_ERROR":
      return { ...state, error: action.payload };
    case "RESET":
      return initialState;
    default:
      return state;
  }
}
```

useReducer is very useful for forms with multiple fields and complex validation logic.

8 Best Practices

Ans →
- ✓ Use useReducer when state logic is complex.
- ✓ Use meaningful action types (e.g., "ADD_TODO").
- ✓ Keep reducer function pure (no side effects).
- ✓ Use constants for action types (optional but cleaner).
- ✓ Combine with Context API for global state.
- ✓ Keep state structure organized.

9 Common Mistakes

Ans →
- ✗ Using useReducer for simple state (unnecessary complexity).
- ✗ Mutating state directly (always return new state).
- ✗ Forgetting to handle default case in reducer.
- ✗ Using too many action types (keep it simple).
- ✗ Putting side effects inside reducer (use useEffect instead).
- ✗ Not using meaningful action names.

"Good state management leads to better apps !"

"Complex problems need better tools. useReducer gives you control !"

Learn Today, Build Tomorrow !

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Context API

Share data across components without prop drilling

Global State Made Simple! No Prop Drilling Anymore

1 What is Context API ?

Ans→
- Context API is a built-in feature in React that allows us to share data across components without passing props manually at every level (prop drilling).
- It is useful for global data like theme, user authentication, language, etc.
- It is simpler than using state management libraries for small to medium applications.
- Introduced in React 16.3.

2 Why Use Context API ?

Ans→
- Avoids prop drilling.
- Makes data accessible to any component in the component tree.
- Useful for global values (theme, user, auth, etc.).
- Cleaner and more maintainable code.
- Good for medium-sized apps.
- Works well with useReducer for complex state.

Common Use Cases :-
- Theme (dark/light)
- User authentication
- Language settings
- Global UI state
- Shopping cart
- Feature flags

3 Basic Setup (createContext)

Ans→

```jsx
// ThemeContext.js
import { createContext } from "react";

export const ThemeContext = createContext();

// Provider Component
export function ThemeProvider({ children }) {
  const [theme, setTheme] = useState("light");

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

Key Point :-
- createContext() creates a context object.
- Provider makes the data available to all child components.
- value can be any data (state, function, object, etc.).

Wrap your app with Provider (usually in main.jsx or App.jsx)

4 Using Context in a Component

Ans→

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

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

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

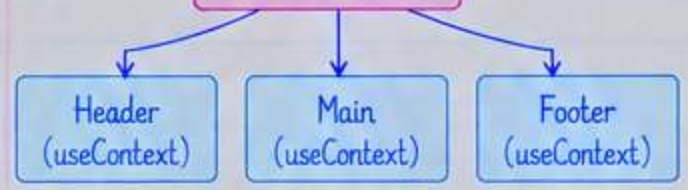
Output :-

Current Theme: light
[Toggle Theme]

Note :-
- We use useContext() to access the context value.
- It returns the value from the nearest Provider.
- When the value changes, the component re-renders automatically.

5 Component Tree with Context

Ans→



All components inside ThemeProvider can access the context value.

No need to pass props manually at every level !

6 Real World Example (User Auth Context)

Ans→

```jsx
// AuthContext.js
import { createContext, useState } from "react";

export const AuthContext = createContext();

export function AuthProvider({ children }) {
  const [user, setUser] = useState(null);

  const login = (userData) => setUser(userData);
  const logout = () => setUser(null);

  return (
    <AuthContext.Provider value={{ user, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
}
```

Use in Component :-

```jsx
import { useContext } from "react";
import { AuthContext } from "../AuthContext";

function Profile() {
  const { user, logout } = useContext(AuthContext);

  return (
    <div>
      {user ? (
        <div>
          <p>Welcome, {user.name}</p>
          <button onClick={logout}>Logout</button>
        </div>
      ) : (
        <p>Please login</p>
      )}
    </div>
  );
}
```

7 Context Limitations

Ans→
- ✗ Not ideal for very large and complex applications (consider Redux, Zustand, etc.).
- ✗ Can cause unnecessary re-renders if not optimized (e.g., when passing new object values).
- ✗ Should not be used for frequently changing state (like input fields).
- ✗ Debugging can be harder in large apps.
- ✗ No built-in persistence (like localStorage).

8 Best Practices

Ans→
- ✓ Use Context for global/shared data only.
- ✓ Keep the context value minimal and focused.
- ✓ Split contexts for different concerns (e.g., ThemeContext, AuthContext, etc.).
- ✓ Use useMemo to prevent unnecessary re-renders.
- ✓ Combine with useReducer for complex state logic.
- ✓ Provide meaningful default values.
- ✓ Keep the provider as high as needed, but not higher than necessary.

9 Context vs Prop Drilling (Comparison)

Ans→

Feature	Prop Drilling	Context API
Data passing	Manually pass props	Automatically available
Code readability	Becomes messy	Cleaner and simpler
Best for	Small apps	Medium sized apps
Reusability	Harder	Easier
Performance	No extra re-renders	May re-render all consumers
Setup	No setup	Requires Provider and useContext

10 Key Takeaways

Ans→
- Context API solves the prop drilling problem.
- It's simple and powerful for global state.
- Use it wisely, only when needed.
- Combine with useReducer for complex logic.
- Avoid overusing it to prevent performance issues.
- It helps in building cleaner and more maintainable React apps.

Use Context Wisely, Not Everywhere

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Context + useReducer

Build a global state with Context and useReducer

Manage Global State Like a Pro ! Simple & Scalable

1 Why Context + useReducer ?

Ans →

- Context is used to share state across components.
- useReducer is used to manage complex state logic with a reducer function.
- Combining them gives a clean, scalable and maintainable global state solution.
- Useful for theme, authentication, cart, user data, settings, etc.
- Avoids prop drilling and keeps logic in one place.

2 Folder Structure

Ans →

```
src/
|-- context/
|     |-- AppContext.jsx
|     |-- AppProvider.jsx
|     |-- reducer.js
|-- components/
|     |-- Header.jsx
|     |-- UserProfile.jsx
|-- App.jsx
|-- main.jsx
```

Keep your context logic organized in a separate folder for better scalability.

3 Create Context

Ans →

```jsx
// AppContext.jsx
import { createContext } from "react";

export const AppContext = createContext();
```

createContext() creates a context that can be used to share state across components.

4 Reducer Function

Ans →

```jsx
// reducer.js
export const initialState = {
  user: null,
  theme: "light",
  cart: [],
};

export function reducer(state, action) {
  switch (action.type) {
    case "SET_USER":
      return { ...state, user: action.payload };
    case "TOGGLE_THEME":
      return {
        ...state,
        theme: state.theme === "light" ? "dark" : "light",
      };
    case "ADD_TO_CART":
      return { ...state, cart: [...state.cart, action.payload] };
    case "REMOVE_FROM_CART":
      return { ...state, cart: state.cart.filter(item => item.id !== action.payload)};
    default:
      return state;
  }
}
```

Reducer function decides how state changes based on the action type.

5 Create Provider Component

Ans →

```jsx
// AppProvider.jsx
import { createContext } from "react";
import { AppContext } from "../AppContext";
import { reducer, initialState } from "../reducer";

export function AppProvider({ children }) {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <AppContext.Provider value={{ state, dispatch }}>
      {children}
    </AppContext.Provider>
  );
}
```

We use useReducer here instead of useState to manage complex state in a single place.

6 Use Context in Components

Ans →

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

function Header() {
  const { state, dispatch } = useContext(AppContext);
  return (
    <div>
      <p>Theme: {state.theme}</p>
      <button onClick={() => dispatch({ type: "TOGGLE_THEME" })}>
        Toggle Theme
      </button>
    </div>
  );
}
```

useContext() gives access to the state and dispatch from the nearest Provider.

7 Example: Theme Toggle + User Info

Ans →

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

function UserProfile() {
  const { state, dispatch } = useContext(AppContext);
  return (
    <div>
      {state.user ? (
        <p>Welcome, {state.user.name}</p>
      ) : (
        <p>Please login</p>
      )}
      <button onClick={() => dispatch({ type: "SET_USER",
      payload: { name: "Sovit", email: "sovit@example.com" } })}>
        Login
      </button>
    </div>
  );
};
```

Output :-

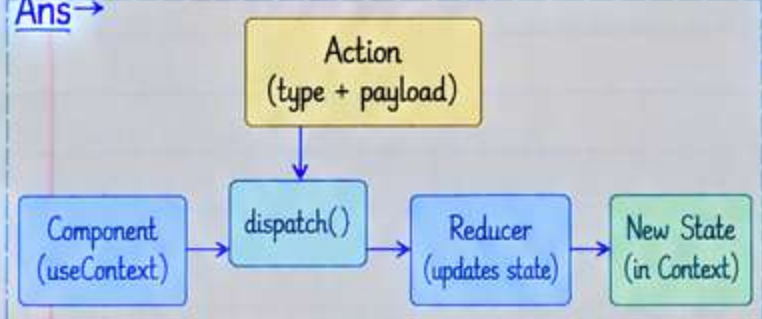
```
Theme: dark
Welcome, Sovit
[ Toggle Theme ]
```

Note :-

- Same context can manage multiple states like theme, user, cart, etc.
- Add more actions as per your app requirements.

8 Context + useReducer Flow

Ans →



Components dispatch actions → Reducer updates state → New state is available to all components using useContext().

9 Best Practices

Ans →

- ✓ Use Context + useReducer for complex global state.
- ✓ Keep reducer function pure (no side effects).
- ✓ Split contexts if the state is large.
- ✓ Use meaningful action names (e.g., SET_USER).
- ✓ Provide a default value for context.
- ✓ Keep state and dispatch in a single object.
- ✓ Organize files in a separate folder.
- ✓ Use this pattern for theme, auth, cart, etc.

10 Common Mistakes

Ans →

- ✗ Using Context for simple local state.
- ✗ Mutating state directly inside reducer.
- ✗ Not using a default value in createContext().
- ✗ Keeping too much data in a single context.
- ✗ Not organizing context files.
- ✗ Forgetting to wrap app with Provider.

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Router

Handle navigation and routing in React applications

Single Page Applications with Multiple Pages !

1 What is React Router ?

- React Router is a routing library for React to handle navigation between different components (pages).
- It allows us to build Single Page Applications (SPA) without reloading the entire page.
- It keeps the UI in sync with the URL.
- Most commonly used library: react-router-dom

React Router lets you build multi-page like experiences in a single page app (SPA).

2 Installation

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

Note :-

- We use react-router-dom for web apps (instead of react-router).
- Make sure to install the latest version.
- It works with both Vite and Create React App.
- No extra setup needed in Vite.

Useful Imports :-

- BrowserRouter
- Routes
- Route
- Link
- NavLink
- useParams
- useNavigate
- useLocation
- Outlet

3 Basic Setup

```jsx
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Home from "./pages/Home";
import About from "./pages/About";

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

Key Points :-

- BrowserRouter wraps the entire app.
- Routes contains multiple Route.
- Each Route maps a URL path to a component.
- element is the component to render.
- / is the homepage route.

4 Link and NavLink

- Link is used to navigate between pages without reloading.
- NavLink is like Link but can apply active styles to the current route.

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

function Navbar() {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
      <NavLink to="/contact">Contact</NavLink>
    </nav>
  );
}
```

Note :-

- NavLink automatically adds an active class when the route is active.

Example (CSS) :-

```css
.active {
  color: blue;
  font-weight: bold;
}
```

5 Route Parameters (Dynamic Routes)

- We can pass dynamic values in the URL using route parameters.
- Useful for pages like /users/:id, /products/:id, etc.

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

function User() {
  const { id } = useParams();

  return <h2>User ID: {id}</h2>;
}
```

Example URL :-

/users/101
/users/abhishek

Here, :id is a dynamic parameter.

6 Programmatic Navigation

- We can navigate to a different route using useNavigate().
- Useful after form submission, login, etc.

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

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

  const handleLogin = () => {
    // after successful login
    navigate("/dashboard");
  };
  return <button onClick={handleLogin}>Login</button>;
}
```

Other Useful Hooks :-

- ✓ useNavigate() → navigate programmatically
- ✓ useParams() → get route parameters
- ✓ useLocation() → get current URL info

useLocation Example :-

```jsx
import { useLocation } from "react-router-dom";

const location = useLocation();
console.log(location.pathname);
```

7 Nested Routes (Basic Idea)

- We can define routes inside another route for nested pages.
- The parent component should use <Outlet /> to render child routes.

```jsx
import { Routes, Route, Outlet } from "react-router-dom";
function Dashboard() {
  return (
    <div>
      <h2>Dashboard</h2>
      <Outlet /> {/* child routes will render here */}
    </div>
  );
}
```

Nested routes help in creating layouts with common UI (Header, Sidebar, etc.)

8 Project Structure Example

```
src/
  components/
    Navbar.jsx
  pages/
    Home.jsx
    About.jsx
    User.jsx
    Dashboard.jsx
  App.jsx
  main.jsx
```

Keep routes, pages and components organized for better maintainability.

9 Best Practices

- ✓ Use meaningful and clean URLs.
- ✓ Keep routes and components in separate folders.
- ✓ Use NavLink for active navigation.
- ✓ Use route parameters for dynamic pages.
- ✓ Use nested routes for layouts.
- ✓ Handle 404 page (Not Found).
- ✓ Keep routing logic simple and organized.
- ✓ Use protected routes for authenticated pages (we'll cover this in next page).

10 Common Mistakes

- ✗ Forgetting to wrap with BrowserRouter.
- ✗ Using href instead of to in Link.
- ✗ Not using unique keys in route lists.
- ✗ Incorrect path names (case sensitive).
- ✗ Not handling 404 routes.
- ✗ Putting all routes inside one file (messy code).

"Good routing makes your app feel like a real product !"

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Advanced Routing

Nested routes, layouts, protected routes and more

Better Navigation Better UX Bigger Apps

1 Why Advanced Routing ?

- Single page routing is enough for small apps.
- Advanced routing helps in real-world apps with nested pages, layouts, authentication, redirects, etc.
- Provides better user experience and clean URLs.
- Helps in building scalable applications.

Routing = Right Component at the Right Time

2 Nested Routes

- Nested routes allow you to render child routes inside a parent route using <Outlet />.
- Useful for layouts like Dashboard, Settings, etc.
- Parent component remains the same, only
- child content changes.

3 Example: Nested Routes

```
import { BrowserRouter, Routes, Route, Outlet, Link }
from "react-router-dom";
import Layout from "../Layout";
import Dashboard from "./pages/Dashboard";
import Settings from "./pages/Settings";

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Layout />}>
          <Route index element={<Dashboard />} />
          <Route path="settings" element={<Settings />} />
        </Routes>
      </Routes>
    </BrowserRouter>
  );
}
export default App;
```

Folder Structure :-

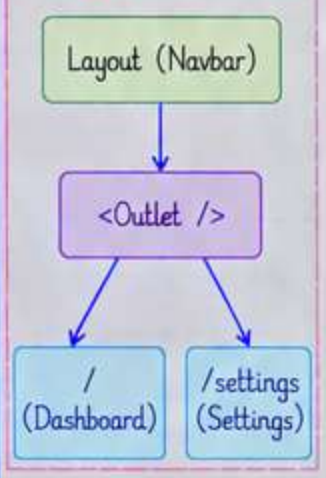
```
src/
 |-- components/
 |     |-- Layout.jsx
 |-- pages/
 |     |-- Dashboard.jsx
 |     |-- Settings.jsx
 |-- App.jsx
 |-- main.jsx
```

Note :-
<Outlet /> is used inside Layout.jsx where child routes will be rendered.

4 Layout Component (with Outlet)

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

function Layout() {
  return (
    <div>
      <nav>
        <Link to="/">Dashboard</Link>
        <Link to="/settings">Settings</Link>
      </nav>

      <hr />
      <div className="content">
        Outlet /> {/* Child route rendered here */}
      </div>
    </div>
  );
}
export default Layout;
```

Output (UI Flow) :-



5 Protected Routes

- Used to restrict access to certain routes (like dashboard) if user is not authenticated.
- We can check authentication and redirect to login page.

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

function ProtectedRoute({ isAathenticated }) {
  return isAuthenticated ? <Outlet /> : <Navigate to="/login" />;
}

export default ProtectedRoute;
```

Usage :-

```
<Route
  path="/dashboard"
  element={
    <ProtectedRoute
      isAuthenticated={user} >
    />
  }>
  <Route index element={<Dashboard/>}/>
</Route>
```

6 Redirects

- Redirect users to another page automatically.
- Useful after login, logout or for invalid routes.

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

// Redirect example
function Home() {
  return <Navigate to="/dashboard" />;
}
// or in route
<Route path="/" element={<Navigate to="/dashboard" />}/>
```

Common Redirects :-

- After login → dashboard
- After logout → login
- Invalid route → 404
- Old route → new route (for URL changes)

7 404 Not Found Page

- A custom 404 page is shown when no route matches.
- Improvos user experience.

```
function NotFound() {
  return (
    <div className="not-found">
      <h2>404 - Page Not Found</h2>
      <p>The page you are looking for deesn't exist</p>
      <Link to="/">Go to Home</Link>
    </div>
} );
// In App.jsx
<Route path="*" element={<NotFound />} />
```

Use * path at the end to catch all undefined routes.

8 Query Parameters

- Used to read values from URL (like search, filter, etc).

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

function ProductList() {
  const [searchParams] = useSearchParams();
  const category = searchParams.get("category");

  return <p>Category: {category}</p>;
}
// URL: /products?category=react
```

9 useLocation Hook

- Gives current URL information (pathname, search, hash, state).

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

function ShowLocation() {
  const location = useLocation();

  return (
    <div>
      <p>Path: {location.pathname}</p>
      <p>Search: {location.search}</p>
      <p>Hash: {location.hash}</p>
    </div>
  );
}
```

10 Best Practices

- ✔ Use meaningful and clean URLs.
- ✔ Organize routes and components in separate folders.
- ✔ Use layout routes for common UI (header, sidebar).
- ✔ Protect sensitive routes.
- ✔ Use 404 page for invalid routes.
- ✔ Keep routing logic simple and readable.
- ✔ Use lazy loading for route components (with React.lazy).
- ✔ Handle redirects properly.

Common Mistakes

- ✘ Forgetting to wrap app with <BrowserRouter>.
- ✘ Incorrect route paths or typos.
- ✘ Not using <Outlet /> for nested routes.
- ✘ No 404 page.
- ✘ Not handling authentication properly.
- ✘ Putting all routes inside one file.
- ✘ Overcomplicating routing unnecessarily.
- ✘ Not using NavLink for active styles.

"Good routing makes your app feel complete !"

Build Better Navigation
Build Bigger Things !

Karyvio
LEARN | BUILD | GROW | GET HIRED

React API & Async Data

Fetch data from APIs and handle async operations

Real Data
Real Projects
Real Skills
That's the Goal !

1 Why Work with APIs ?

Ans →

- APIs allow us to fetch and send data to a server.
- Used to get real data (users, products, posts, etc.).
- Make our app dynamic and useful.
- Most real-world applications (social media, e-commerce, dashboards, etc.) use APIs.
- Helps in building full-stack applications.

2 Common HTTP Methods

Ans →

Method	Purpose
GET	Fetch data (read)
POST	Send data (create)
PUT	Update entire resource
PATCH	Update partial data
DELETE	Delete data

Base URL Example :-

https://jsonplaceholder.typicode.com

(Free API for practice)

3 Basic Fetch Example (GET)

Ans →

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

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

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(res => {
        if (!res.ok) throw new Error("Failed to fetch");
        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} - {user.email}</li>
      ))}
    </ul>
  );
}
export default Users;
```

Key Points :-

- fetch() returns a Promise.
- We use .then() or async/await.
- Always handle loading and error states.
- Convert response using res.json().
- Use useEffect to fetch data on component mount.

4 Using Async / Await (Cleaner Way)

Ans →

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

async function fetchPosts() {
  const res = await fetch("https://jsonplaceholder.typicode.com/posts");
  if (!res.ok) throw new Error("Failed to fetch posts");
  const data = await res.json();
  return data;
}

function Posts() {
  const [posts, setPosts] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState("");

  useEffect(() => {
    const loadPosts = async () => {
      try {
        const data = await fetchPosts();
        setPosts(data);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    };
    loadPosts();
  }, []);
  // render UI (similar to above)
}
```

Async/Await Benefits :-

- Cleaner and easier to read.
- Looks like synchronous code.
- Use try...catch for error handling.
- Recommended for modern React apps.

5 POST Request (Create Data)

Ans →

```
async function createPost() {
  const res = await fetch("https://jsonplaceholder.typicode.com/posts", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({
      title: "Learn React",
      body: "This is a test post",
      userId: 1
    })
  });
  const data = await res.json();
  console.log("Created Post:", data);
} )
```

Important :-

- Use correct HTTP method.
- Set headers (Content-Type).
- Send data using JSON.stringify().
- Handle response and errors.

6 Update (PUT/PATCH) and Delete

Ans →

```
// Update (PUT)
async function updatePost(id) {
  const res = await fetch(`https://jsonplaceholder.typicode.com/posts/${id}`, {
    method: "PUT",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({
      title: "Updated Title",
      body: "Updated content"
    })
  });
  const data = await res.json();
  console.log("Updated:", data);
}

// Delete
async function deletePost(id) {
  const res = await fetch(`https://jsonplaceholder.typicode.com/posts/${id}`, {
    method: "DELETE"
  });
  console.log("Deleted:", res.status);
}
```

Note :-

- PUT → replace entire resource.
- PATCH → update only specific fields.
- DELETE → remove the resource.

7 Loading, Error and Empty States

Ans →

```
if (loading) return <p>Loading data...</p>;
if (error) return <p>Something went wrong: {error}</p>;
if (data.length === 0) return <p>No data found.</p>;

return (
  <ul>
    {data.map(item => (
      <li key={item.id}>{item.title}</li>
    ))}
  </ul>
);
```

Good UX :-

- Show loading spinner or message.
- Handle errors gracefully.
- Show empty state when no data.
- Keeps users informed and improves experience.

8 Best Practices

Ans →

- ✓ Use async/await for cleaner code.
- ✓ Always handle loading, error and empty states.
- ✓ Use environment variables for API base URL.
- ✓ Create a seperate API service file.
- ✓ Use try...catch for error handling.
- ✓ Cancel requests if component unmounts (AbortController).
- ✓ Keep API logic separate from UI.
- ✓ Follow proper folder structure.
- ✓ Use meaningful variable and function names.

9 Common Mistakes

Ans →

- ✗ Not handling errors.
- ✗ Not showing loading state.
- ✗ Using wrong HTTP method.
- ✗ Forgetting to convert response to JSON.
- ✗ Hardcoding API URLs.
- ✗ Updating state after component unmounts.
- ✗ Not using environment variables.
- ✗ Mixing API logic inside components.
- ✗ Ignoring edge cases (empty data, network failure, etc.).

"APIs turn your app from static to real. Use them wisely !"

Learn Today, Build Tomorrow !

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Architecture

Build scalable and maintainable React applications

Good Architecture Today Bigger Applications Tomorrow !

1 Why Project Architecture Matters ?

Ans →

- As applications grow, code becomes complex.
- Good architecture keeps code organized, scalable and maintainable.
- Makes collaboration easier in teams.
- Helps in reusability, testing and faster development.
- Prevents "spaghetti code".
- Essential for real-world and production applications.

"Scalable Code = Less Stress + More Growth"

2 Feature-Based Folder Structure

Ans →

```
src/
├── app/                    # app level config (routes, providers)
├── features/               # each feature in separate folder
│   ├── auth/
│   │   ├── components/
│   │   ├── pages/
│   │   ├── hooks/
│   │   ├── services/
│   │   └── index.js
│   └── tasks/
│       └── ... (same structure)
├── shared/                 # reusable components, hooks, utils
│   ├── components/
│   ├── hooks/
│   ├── utils/
│   └── constants/
├── assets/                 # images, icons, styles
├── App.jsx
└── main.jsx
```

Keep features independent and self-contained for better scalability.

3 Component Structure (Best Practice)

Ans →

Example: TaskList component

```
- features/
    tasks/
      ├── components/
      └── TaskList/
            ├── TaskList.jsx
            ├── TaskList.css
            └── index.js   # for cleaner imports
```

Why this way?

- Keeps related files together.
- Easy to find and maintain.
- Scalable for large projects.
- Makes code more modular.

4 Separation of Concerns

Ans →

Layer	Responsibility
Components	UI rendering (presentational)
Pages	Page level components (route based)
Hooks	Reusable logic (custom hooks)
Services	API calls and data handling
Utils	Helper functions
Constants	Static values (API URLs, roles, etc.)
Context	Global state management
Routes	Application routing
Assets	Images, icons, styles

"Each file should have a single responsibility."

5 State Management Strategy

Ans →

- Local State: useState → for simple, local data.
- Context API: for global data (theme, user, etc.).
- useReducer: for complex state logic.
- Combine Context + useReducer for scalable global state.
- Keep state as close as possible to where it is used.
- Avoid unnecessary global state.

Right tool for the Right problem !

6 Example: Feature Module (Tasks)

Ans →

```
// features/tasks/hooks/useTasks.js
import { useState, useEffect } from "react";
import { getTasks } from "../services/taskService";

export function useTasks() {
  const [tasks, setTasks] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    getTasks()
      .then((data) => { setTasks(data); setLoading(false); })
      .catch(() => { setLoading(false); });
  }, []);

  return { tasks, loading };
}
```

Custom hooks help in keeping components clean and reusable.

7 Reusability (Shared Folder)

Ans →

```
├── shared/
    ├── components/
    │   ├── Button.jsx
    │   ├── Input.jsx
    │   ├── Modal.jsx
    │   └── index.js
    ├── hooks/
    │   ├── useLocalStorage.js
    │   └── useDebounce.js
    ├── utils/
    │   ├── helpers.js
    │   └── validators.js
    └── constants/
        ├── api.js
        └── index.js
```

Benefits ::

- Reusable components across features.
- Saves development time.
- Maintains consistency in UI and logic.
- Easy to update and scale.

8 Environment Variables

Ans →

- Store sensitive data like API base URL in .env file.
- Use Vite: variables must start with VITE_

.env

```
VITE_API_BASE_URL=https://api.example.com
VITE_APP_NAME=Karyvio
VITE_ENV=development
```

Never hardcode API keys or sensitive data in your code.

9 Best Practices

Ans →

- ✓ Use feature-based folder structure.
- ✓ Keep components small and reusable.
- ✓ Use meaningful and consistent naming.
- ✓ Separate UI, logic and API calls.
- ✓ Use custom hooks for reusable logic.
- ✓ Keep global state minimal and structured.
- ✓ Use environment variables.
- ✓ Follow code linting and formatting (ESLint, Prettier).
- ✓ Write clean, self-explanatory code.
- ✓ Plan for scalability from the beginning.

10 Common Mistakes

Ans →

- ✗ Putting all code in one folder.
- ✗ Creating very large components.
- ✗ Not following a consistent folder structure.
- ✗ Overusing global state.
- ✗ Mixing UI and business logic.
- ✗ Hardcoding sensitive data.
- ✗ Not using reusable components.
- ✗ Ignoring code organization until the project becomes large.
- ✗ Not using meaningful names.

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Advanced Topics + Production

Take your React skills from development to real-world production

Write
Better Code
Build Real Products
Think Long Term
That's the Goal !

1 Code Splitting with React.lazy()

Ans→
- Load components only when needed.
- Improves initial load time (smaller bundle).
- Used with Suspense for fallback UI.

```
import { lazy, Suspense } from "react";
const Dashboard = lazy(() => import("./pages/Dashboard"));

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <Dashboard />
    </Suspense>
  );
}
```

When to use ?
- Routes
- Large components
- Admin / rarely used features

2 Suspense in Detail

- Suspense lets you show a fallback UI while a component is loading (lazy loading, data fetching, etc).
- Can be used with React.lazy or data fetching libraries (such as React Query).

```
<Suspense fallback={<div>Loading Page...</div>}>
  <Routes>
    <Route path="/dashboard" element={<Dashboard />} />
  </Routes>
</Suspense>
```

You can also use nested Suspense for different sections.

3 React Transitions (React 18)

Ans→
- useTransition helps keep the UI responsive during state updates.
- Useful for expensive operations like searching, filtering, or large lists.

```
import { useState, useTransition } from "react";
function Search() {
  const [query, setQuery] = useState("");
  const [isPending, startTransition] = useTransition();
  function handleSearch(e) {
    const value = e.target.value;
    startTransition(() => {
      setQuery(value);
    });
  }
  return (
    <div>
      <input onChange={handleSearch} />
      {isPending && <p>Searching...</p>}
    </div>
  );
}
```

Benefits :-
- Keeps UI smooth and responsive.
- Marks non-urgent updates.
- Great for search, filters, and heavy renders.

4 Rendering Strategies

Ans→ Different rendering strategies based on the use case.

Strategy	Meaning	Use Case	Examples
CSR (Client Side)	Render in browser using JS	Interactive apps	Vite + React
SSR (Server Side)	Render on server per request	SEO, dynamic data	Next.js (getServerSideProps)
SSG (Static Generation)	Pre-render at build time	Static content	Next.js (getStaticProps)
ISR (Incremental)	Update static pages after deployment	Blogs, docs	Next.js (revalidate)

5 React in Production (Build & Deploy)

Ans→
- Build command (Vite): `npm run build`
- Output folder: dist/
- Deploy to: Vercel, Netlify, Cloudflare Pages, AWS S3, etc.
- Use environment variables for API URLs.
- Enable compression (gzip / brotli).
- Set up custom domain.
- Use HTTPS and proper headers.

Example .env :-

```
VITE_API_BASE_URL=https://api.example.com
VITE_APP_NAME=Karyvio
VITE_ENV=production
```

"Build for Production,
Not Just for Your Local Machine !"

6 Security Best Practices

Ans→
- Keep dependencies up to date (npm audit).
- Use environment variables (don't expose secrets).
- Validate and sanitize user input.
- Use HTTPS and secure headers.
- Avoid storing sensitive data in localStorage.
- Implement authentication (JWT, OAuth, etc).
- Prevent XSS and CSRF attacks.
- Use Content Security Policy (CSP).
- Protect API keys and sensitive endpoints.

Security is a feature, not an afterthought !

7 Accessibility (a11y)

- Use semantic HTML elements (header, nav, main, etc).
- Add alt text for images.
- Use proper heading structure (h1, h2, h3...).
- Ensure keyboard navigation (tab, enter).
- Maintain good color contrast.
- Use ARIA labels when needed.
- Test with screen readers.
- Make your app usable for everyone !

Inclusive products create a better internet !

8 Testing Overview

Ans→
- Testing helps catch bugs early and improves code quality.
- Common testing tools in React:
  - Jest (test runner)
  - React Testing Library (component testing).
  - Vitest (for Vite projects)
  - Cypress / Playwright (end-to-end testing).

```
// Example: simple test
import { render, screen } from "@testing-library/react";
render(<h1>Hello React</h1>);
expect(screen.getByText("Hello React")).toBeInTheDocument();
```

9 Performance Checklist

Ans→
- ✓ Use code splitting (React.lazy).
- ✓ Avoid unnecessary re-renders.
- ✓ Use React.memo, useMemo, useCallback.
- ✓ Optimize large lists (virtualization).
- ✓ Keep bundle size small.
- ✓ Use proper keys in lists.
- ✓ Lazy load images and assets.
- ✓ Use production build (minified).
- ✓ Analyze bundle (e.g. vite-bundle-visualizer).
- ✓ Monitor performance (Lighthouse, DevTools).

10 Key Takeaways & Next Steps

- ✓ Use advanced features where needed, not everywhere.
- ✓ Choose the right rendering strategy for your project.
- ✓ Focus on performance, security, accessibility and testing.
- ✓ Follow best practices and keep learning.
- ✓ Build real projects and deploy them.
- ✓ Stay updated with the latest React features (React 19+).
- ✓ You're now ready to build production-ready React applications !

Next Steps :-
1. Build and deploy your own project.
2. Explore Next.js for full-stack React.
3. Learn advanced state management (Zustand, Redux).
4. Dive deeper into testing.
5. Contribute to open source.
6. Keep improving and never stop learning !

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Final Summary & Next Steps

Recap everything and plan your journey ahead

It's Not the End
It's the Beginning
Keep Learning
Keep Building

1 Key Takeaways

- Component-based UI development.
- Understanding of JSX, components, props, and state.
- Working with hooks (useState, useEffect, useContext, etc.).
- Handling events, forms and conditional rendering.
- Building reusable components.
- Managing global state with Context + useReducer.
- Routing with React Router (basic to advanced).
- Fetching and handling API data (async/await).
- Performance optimization techniques.
- Writing clean, maintainable and scalable code.
- Best practices for real-world projects.

You now have a strong foundation in React
Keep practicing to become job ready !

2 Learning Path (What's Next ?)

1. Practice more small projects.
2. Build a full-stack project (React + Node.js).
3. Learn a state management library (Redux or Zustand).
4. Explore Next.js (SSR, SSG, API routes).
5. Learn TypeScript for better development experience.
6. Work with databases (MongoDB, PostgreSQL, etc.).
7. Learn testing in depth (Jest, React Testing Library).
8. Explore deployment and DevOps (Vercel, Docker, CI/CD).
9. Contribute to open source projects.
10. Keep building your portfolio and apply for jobs.

Consistency beats talent
Practice a little every day !

3 Mini Project Ideas

- Todo App (with CRUD and localStorage)
- Weather App (using real API)
- Blog App (React + API)
- E-commerce Product Listing
- User Authentication (login/signup)
- Movie Search App (using external API)
- Notes App (with Context + useReducer)
- Chat App UI (real-time with Firebase)
- Portfolio Website
- Full-Stack Project (React + Node.js + MongoDB)

Build Projects Not Just Notes !

4 Useful Resources

Resource	Link
React Official Docs	https://react.dev
React Router Docs	https://reactrouter.com
Vite Docs	https://vitejs.dev
Next.js Docs	https://nextjs.org
MDN Web Docs	https://developer.mozilla.org
React Testing Library	https://testing-library.com
Tailwind CSS	https://tailwindcss.com
TypeScript	https://www.typescriptlang.org
Vercel (Deployment)	https://vercel.com
Awesome React (GitHub)	https://github.com/enaqx/awesome-react

Good Developers Know Where to Learn !

5 Career Opportunities

- Frontend Developer (React.js)
- Full Stack Developer (React + Node.js)
- UI/UX Developer
- Software Engineer (Frontend)
- Freelancer (React Projects)
- Open Source Contributor
- Tech Blogger / Content Creator
- Startup Founder (Build your own product).

React Skills Can Take You To Amazing Opportunities !

6 Common Interview Questions

- What is React and why do we use it?
- Difference between functional and class components?
- What are hooks? (useState, useEffect, etc.)
- What is the virtual DOM?
- How does React handle re-renders?
- What is the difference between useMemo and useCallback?
- How does Context API work?
- How do you optimize a React application?
- What is the difference between CSR, SSR and SSG?
- Explain React Router and protected routes.
- How do you handle API errors and loading states?
- What are best practices for scalable React apps?

Prepare Well Not Just to Get a Job But to Grow as a Developer !

7 Real-World Tips

- Write clean and readable code.
- Use meaningful component and variable names.
- Keep components small and focused.
- Follow a consistent folder structure.
- Use environment variables for sensitive data.
- Handle errors and edge cases.
- Optimize performance (lazy loading, memoization).
- Write tests for important features.
- Use Git and maintain a good commit history.
- Always keep learning and stay updated.

Clean Code
Happy Developers
Better Products !

8 React Developer Checklist

- Understand core concepts (JSX, components, props, state).
- Learn and use hooks effectively.
- Build and deploy multiple projects.
- Use React Router for navigation.
- Fetch and handle API data.
- Implement global state management.
- Follow best practices and clean architecture.
- Ensure responsive and accessible UI.
- Optimize performance.
- Write tests.
- Deploy to production.
- Keep learning new tools and technologies.

Tick All The Boxes And Become A Confident React Developer !

9 Final Words

"React is just a tool.
What really matters is what you build with it.
Keep learning, keep building, keep improving.
The best is yet to come !"
- Karyvio

10 Remember This ...

- ☆ It's okay to make mistakes.
- ☆ Practice is more important than perfection.
- ☆ Build real projects, not just tutorials.
- ☆ Learn, apply, and teach others.
- ☆ Stay curious and never stop learning.
- ☆ You are capable of building amazing things !

"Small Steps Every Day Lead to Big Results !