REACT

DEVELOPER

NOTES

👨🏻‍💻 Name: Mr. Rupesh Lal

🌐 Website: www.itmodem.com

📞 WhatsApp: +91 7870886355

💼 Profession: Software Engineer

**📘 React Stack Notes – Full Outline (Basic to Advanced + All Hooks)**

**📘Chapter 1: Introduction to React**

1. What is React?
2. Why React? (Benefits)
3. React vs Vanilla JS vs Other Frameworks
4. Key Features (Virtual DOM, Components, One-Way Data Flow)
5. React Workflow Overview (Tree, Rendering, Updates)
6. What is JSX?
7. How to Write and Render JSX

**📘Chapter 2: Environment Setup**

1. Install Node.js & npm
2. Create React App (CRA)
3. Vite (optional)
4. React Folder Structure Explained
5. Running Your First React App

**📘Chapter 3: Components & Props**

1. Functional Components
2. Class Components (for understanding legacy)
3. Props and Children
4. Default Props & PropTypes
5. Component Composition

**📘Chapter 4: useState Hook (State Handling)**

1. What is State?
2. useState Syntax & Usage
3. Multiple States
4. Updating State Based on Previous Value
5. Exercise: Counter App

**📘Chapter 5: Event Handling**

1. Handling Events in JSX
2. Passing Parameters to Event Handlers
3. Handling Forms and Inputs
4. Exercise: Registration Form

**📘Chapter 6: Conditional Rendering & Lists**

1. if/else, Ternary Operator
2. Logical AND (&&)
3. Rendering Lists with .map()
4. Unique Keys in List Rendering
5. Exercise: Todo App

**📘Chapter 7: useEffect Hook (Side Effects)**

1. What is a Side Effect?
2. useEffect Syntax & Usage
3. useEffect with Dependency Array
4. Cleanup Functions
5. Exercise: Live Clock / Fetch API on Load

**📘Chapter 8: useRef Hook**

1. useRef vs DOM Access
2. useRef for Focus/Input Fields
3. useRef for Storing Previous Values
4. useRef for Timer/Interval
5. Exercise: Input Focus on Button Click

**📘Chapter 9: useContext Hook (Global State)**

1. Prop Drilling Problem
2. Creating a Context
3. Using Context Provider & useContext
4. When to Use Context API
5. Exercise: Theme Switcher App

**📘Chapter 10: useReducer Hook**

1. What is Reducer?
2. useReducer vs useState
3. Creating Reducer Functions
4. Exercise: Complex Counter with Actions
5. Mini Project: Todo App with useReducer

**📘Chapter 11: useMemo Hook**

1. What is Memoization?
2. Preventing Expensive Recalculations
3. useMemo Syntax & Examples
4. Exercise: Fibonacci Calculator with useMemo

**📘Chapter 12: useCallback Hook**

1. Preventing Re-Renders of Child Components
2. useCallback vs useMemo
3. Passing Stable Functions to Children
4. Exercise: Optimized Button Click Tracker

**📘Chapter 13: Custom Hooks**

1. What are Custom Hooks?
2. Creating Reusable Hook Functions
3. Naming Conventions (useXYZ)
4. Example: useLocalStorage, useTimer
5. Exercise: Build your own useCounter Hook

**📘Chapter 14: React Router (v6+)**

1. Setting up React Router
2. <BrowserRouter>, <Routes>, <Route>
3. useParams, useNavigate, useLocation
4. Route Nesting
5. Protected Routes (basic)
6. Mini Project: Multi-Page Blog App

**📘Chapter 15: Forms and Validation**

1. Controlled Inputs
2. Form Handling
3. Basic Validation with useState
4. Libraries (Formik, Yup – Optional)
5. Exercise: Signup Form with Error Handling

**📘Chapter 16: API Integration (fetch & Axios)**

1. Using fetch()
2. Using Axios (GET, POST, PUT, DELETE)
3. Displaying Loading/Error/Success
4. Exercise: Fetch Users from JSONPlaceholder
5. Mini Project: Weather App

**📘Chapter 17: Component Communication**

1. Parent to Child (Props)
2. Child to Parent (Callback Props)
3. Sibling Communication (Lifting State Up)
4. Exercise: Like/Dislike Button Sync

**📘Chapter 18: State Management with Context + useReducer**

1. Combining useContext + useReducer
2. Global State Management Example
3. Mini Project: Global Todo App

**📘Chapter 19: Performance Optimization**

1. React.memo
2. useMemo & useCallback Review
3. Lazy Loading with React.lazy() and Suspense
4. Avoiding Unnecessary Re-Renders

**📘Chapter 20: Deployment**

1. npm run build
2. Hosting on Netlify, Vercel, Firebase
3. Using Environment Variables
4. Deployment Tips

**🟨 BONUS SECTIONS**

**🔹 All Hooks Summary Cheat Sheet**

* useState
* useEffect
* useContext
* useRef
* useReducer
* useMemo
* useCallback
* useLayoutEffect (advanced)
* useImperativeHandle (advanced)
* useId, useDebugValue (latest additions)
* Custom Hooks

**🔹 Mini Projects**

* Profile Card UI
* Todo List App
* Weather Fetch App
* Expense Tracker
* Blog Viewer (Router + API)
* Timer with Pause/Resume
* Contact Form with Backend Integration

**🔹 Interview Questions Bank**

* **Beginner:** JSX, Props, State, useState
* **Intermediate:** useEffect, useContext, Routing
* **Advanced:** useMemo, useCallback, Custom Hooks, Optimization
* **Real-world:** Component structure, API errors, Debugging

**📘Chapter 1 : Introduction to React**

This section will help you understand what React is, why it’s popular, how it compares to other tools, and what makes it powerful.

🔸 1. 𝗪𝗵𝗮𝘁 𝗶𝘀 𝗥𝗲𝗮𝗰𝘁?

🧠 Concept: React is a popular open-source JavaScript library for building user interfaces, especially single-page applications (SPAs). It was developed by Facebook in 2013.

📌 React helps developers build fast and interactive UIs using reusable components.

🚀 React = JavaScript + Components + Declarative UI

Example:

function Welcome() {

return <h1>Hello, React!</h1>;

}

—

🔸 2. 𝗪𝗵𝘆 𝗥𝗲𝗮𝗰𝘁? (𝗕𝗲𝗻𝗲𝗳𝗶𝘁𝘀)

✅ Key Advantages:

* 🔁 Reusable Components – Write once, use multiple times
* ⚡ Fast Performance – Uses Virtual DOM to optimize updates
* 🧩 Component-Based – Divide UI into logical, testable pieces
* 🎯 Declarative – Just describe what UI should look like
* 📦 Rich Ecosystem – Tons of libraries, tools, and community support
* 🛠 Backed by Facebook – Actively maintained

—

🔸 3. 𝗥𝗲𝗮𝗰𝘁 𝘃𝘀 𝗩𝗮𝗻𝗶𝗹𝗹𝗮 𝗝𝗦 𝘃𝘀 𝗢𝘁𝗵𝗲𝗿 𝗙𝗿𝗮𝗺𝗲𝘄𝗼𝗿𝗸𝘀

| **Feature** | **React** | **Vanilla JS** | **Angular / Vue** |
| --- | --- | --- | --- |
| Approach | Component-based | Manual DOM updates | Framework with more features |
| Performance | High (via Virtual DOM) | Slower with large UIs | Also optimized |
| Learning Curve | Moderate | Easy (but verbose) | Angular: steep / Vue: easy |
| Scalability | Excellent | Poor | Excellent |
| Code Maintenance | Easy (modular components) | Hard (spaghetti code) | Modular |

📌 Important: React is a library (not a full framework), giving developers more flexibility and freedom.

—

🔸 4. 𝗞𝗲𝘆 𝗙𝗲𝗮𝘁𝘂𝗿𝗲𝘀 (𝗩𝗶𝗿𝘁𝘂𝗮𝗹 𝗗𝗢𝗠, 𝗖𝗼𝗺𝗽𝗼𝗻𝗲𝗻𝘁𝘀, 𝗢𝗻𝗲-𝗪𝗮𝘆 𝗗𝗮𝘁𝗮 𝗙𝗹𝗼𝘄)

🟢 𝗩𝗶𝗿𝘁𝘂𝗮𝗹 𝗗𝗢𝗠 React maintains a virtual (in-memory) representation of the real DOM. It calculates the difference (diffing) and updates only the changed parts of the real DOM.

🟢 𝗖𝗼𝗺𝗽𝗼𝗻𝗲𝗻𝘁𝘀 Everything in React is a component (functions or classes). Each component is responsible for rendering a piece of the UI.

Example:

function Header() {

return <header>Welcome</header>;

}

🟢 𝗢𝗻𝗲-𝗪𝗮𝘆 𝗗𝗮𝘁𝗮 𝗙𝗹𝗼𝘄 Data flows from parent to child via props. This makes the app predictable and easier to debug.

—

🔸 5. 𝗥𝗲𝗮𝗰𝘁 𝗪𝗼𝗿𝗸𝗳𝗹𝗼𝘄 𝗢𝘃𝗲𝗿𝘃𝗶𝗲𝘄 (𝗧𝗿𝗲𝗲, 𝗥𝗲𝗻𝗱𝗲𝗿𝗶𝗻𝗴, 𝗨𝗽𝗱𝗮𝘁𝗲𝘀)

🧭 React Workflow (Simplified):

1. Build a component tree
2. Render components to Virtual DOM
3. Compare Virtual DOM with the previous version
4. Update only the changed parts in the real DOM

📌 React uses a diffing algorithm to make UI updates fast and efficient.

—

🔸 6. 𝗪𝗵𝗮𝘁 𝗶𝘀 𝗝𝗦𝗫?

🧠 Concept: JSX (JavaScript XML) is a syntax extension for JavaScript. It looks like HTML but works inside JavaScript.

🧾 JSX allows writing UI components using HTML-like syntax, which improves readability and developer experience.

Example:

const heading = <h1>Hello, JSX!</h1>;

📌 Important:

* JSX is not required, but it’s highly recommended.
* JSX is transpiled into React.createElement(…) under the hood.

—

🔸 7. 𝗛𝗼𝘄 𝘁𝗼 𝗪𝗿𝗶𝘁𝗲 𝗮𝗻𝗱 𝗥𝗲𝗻𝗱𝗲𝗿 𝗝𝗦𝗫

✅ JSX Syntax Rules:

* Always return a single parent element
* Use className instead of class
* Use camelCase for event handlers (onClick, onChange)
* Self-close tags without content

Example JSX:

function App() {

return (

<div>

<h1>My First JSX App</h1>

<p>JSX makes UI intuitive!</p>

</div>

);

}

✅ Rendering JSX to the DOM:

ReactDOM is used to render JSX to the root DOM element.

index.js:

import React from 'react';

import ReactDOM from 'react-dom/client';

import App from './App';

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

root.render(<App />);

—

📝 Mini Exercise:

* Create a component named Greeting and return a

**with your name.**

* Try nesting multiple JSX elements in a fragment (<></>).
* Add a button and attach an onClick event handler.

**📘Chapter 2 : Environment Setup**

This section covers everything you need to set up a working React development environment — from installing Node.js to running your first app.

🔹 **1. Install Node.js & npm**

🧠 Concept: Node.js is a runtime environment that allows you to run JavaScript on the server and is required for building and running React applications. npm (Node Package Manager) comes bundled with Node.js and is used to install packages.

📥 Steps:

* Visit: [https://nodejs.org](https://nodejs.org/)
* Download and install the LTS version.
* Verify installation:

Terminal:

node -v

npm -v

Example Output:

node -v → v20.12.2

npm -v → 10.5.0

✅ Note: You only need to install Node.js once on your system.

—

🔹 **2. Create React App (CRA)**

🧠 Concept: CRA is an officially supported way to create a new single-page React app with zero configuration. It comes pre-configured with Webpack, Babel, ESLint, and more.

📥 Create a React app:

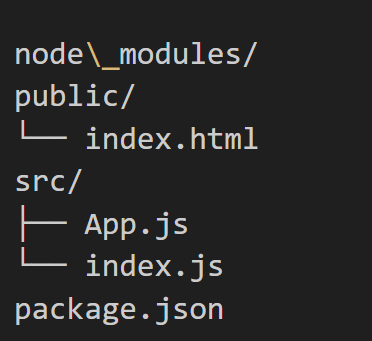
Terminal:

npx create-react-app my-app

cd my-app

npm start

📂  Folder structure (after CRA setup):

* node\_modules/
* public/ └── index.html
* src/ ├── App.js └── index.js
* package.json

🟡 Note: CRA is great for beginners, but may not be ideal for advanced performance tuning.

—

🔹 **3. Vite (Optional Alternative to CRA)**

🧠 Concept: Vite is a modern frontend build tool that offers faster startup and hot reload times. It’s a great alternative to CRA.

📥 Install Vite:

Terminal:

npm create vite@latest my-vite-app

cd my-vite-app

npm install

npm run dev

You’ll be prompted to select a framework → choose React.

📦 Vite is lightweight, fast, and supports ES Modules by default.

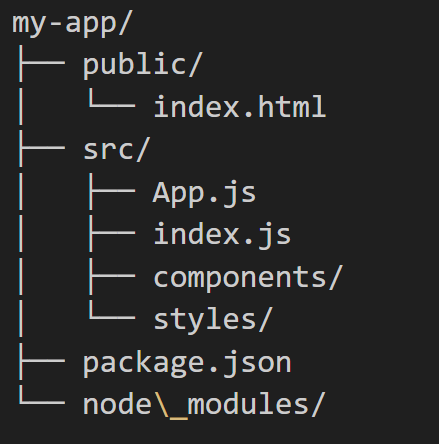
—

🔹 **4. React Folder Structure Explained**

After creating a React app (using CRA or Vite), you’ll see a basic folder structure.

📂 src/ folder:

* index.js or main.jsx — Entry point of the app
* App.js — Root component
* components/ — Custom reusable components
* assets/ — Images, icons, styles
* styles/ — CSS or SCSS files
* pages/ — Page-level components (optional in large apps)
* hooks/ — Custom React hooks (optional)
* utils/ — Utility/helper functions (optional)

📁 Example structure:

🧠 Best Practice: Keep components modular and organize code by feature or route in large apps.

—

🔹 **5. Running Your First React App**

To start your app:

CRA:

npm start

Vite:

npm run dev

🧾 Output: A browser window will open with:

* [http://localhost:3000](http://localhost:3000/) (CRA)
* [http://localhost:5173](http://localhost:5173/) (Vite)

You should see the default “Welcome to React” or “Vite + React” page.

✅ Tip: Any changes you make in App.js will auto-reload in the browser (hot reload).

—

📝 Mini Exercises:

* Create a new React project using CRA and open it in VS Code.
* Explore the src folder and modify the App.js to display your name.
* Try creating a simple component (e.g., Hello.jsx) and render it inside App.

—

🧠 Summary Table:

| **Tool** | **Purpose** | **Command** |
| --- | --- | --- |
| Node.js | JavaScript runtime environment | node -v |
| npm | Package manager for JavaScript | npm install package-name |
| CRA | Quick start React app | npx create-react-app my-app |
| Vite | Lightweight React bundler | npm create vite@latest my-app |
| Start App | Run your app in local server | CRA: npm start / Vite: npm run dev |

—

**📘Chapter 3: Components & Props**

This section dives into the core building blocks of any React app—components and props. You’ll learn how to create components, pass data, and compose complex UIs using smaller reusable pieces.

🔸 1. 𝗙𝘂𝗻𝗰𝘁𝗶𝗼𝗻𝗮𝗹 𝗖𝗼𝗺𝗽𝗼𝗻𝗲𝗻𝘁𝘀

🧠 Concept: Functional components are JavaScript functions that return JSX. They are the most common and recommended way to build components in React.

Example:

function Welcome(props) {

return <h1>Hello, {props.name}!</h1>;

}

Modern ES6+ syntax using arrow functions:

const Welcome = ({ name }) => <h1>Hello, {name}!</h1>;

✅ Best Practice: Use functional components with hooks for modern React development.

—

🔸 2. 𝗖𝗹𝗮𝘀𝘀 𝗖𝗼𝗺𝗽𝗼𝗻𝗲𝗻𝘁𝘀 (𝗳𝗼𝗿 𝘂𝗻𝗱𝗲𝗿𝘀𝘁𝗮𝗻𝗱𝗶𝗻𝗴 𝗹𝗲𝗴𝗮𝗰𝘆)

🧠 Concept: Class components were the primary way to handle state and lifecycle methods before hooks were introduced.

Example:

import React, { Component } from 'react';

class Welcome extends Component {

render() {

return <h1>Hello, {this.props.name}!</h1>;

}

}

🟡 Note: Understanding class components is useful for reading legacy codebases but not necessary for most modern development.

—

🔸 3. 𝗣𝗿𝗼𝗽𝘀 𝗮𝗻𝗱 𝗖𝗵𝗶𝗹𝗱𝗿𝗲𝗻

🧠 Concept: Props (short for properties) are used to pass data from one component to another, usually from parent to child.

Example:

function Greeting(props) {

return <p>Hello, {props.name}</p>;

}

<Greeting name="John" />

📦 Children Prop: You can pass JSX inside a component using props.children.

Example:

function Wrapper(props) {

return <div className="box">{props.children}</div>;

}

<Wrapper>

<p>This is wrapped content</p>

</Wrapper>

—

🔸 4. 𝗗𝗲𝗳𝗮𝘂𝗹𝘁 𝗣𝗿𝗼𝗽𝘀 & 𝗣𝗿𝗼𝗽𝗧𝘆𝗽𝗲𝘀

🧠 Concept:

* Default Props: Set default values for props.
* PropTypes: Help document and type-check props at runtime.

Example:

function Message({ text }) {

return <p>{text}</p>;

}

Message.defaultProps = {

text: "Hello World!",

};

Message.propTypes = {

text: PropTypes.string,

};

🧩 You need to import PropTypes from ‘prop-types’:

import PropTypes from 'prop-types';

—

🔸 5. 𝗖𝗼𝗺𝗽𝗼𝗻𝗲𝗻𝘁 𝗖𝗼𝗺𝗽𝗼𝘀𝗶𝘁𝗶𝗼𝗻

🧠 Concept: Component composition is the idea of building complex UIs by combining multiple components together like Lego blocks.

Example:

function Header() {

return <h1>My Website</h1>;

}

function Content() {

return <p>Welcome to my site.</p>;

}

function Page() {

return (

<div>

<Header />

<Content />

</div>

);

}

📌 This approach makes your UI modular, maintainable, and reusable.

—

📝 Mini Exercises:

* Create a functional component named UserCard with props for name and age.
* Create a Wrapper component that uses props.children to wrap JSX.
* Use PropTypes and defaultProps in one of your components.

**📘Chapter 4: useState Hook (State Handling)**

This section focuses on understanding state in React and using the useState Hook to manage dynamic data inside functional components.

🔸 1. 𝗪𝗵𝗮𝘁 𝗶𝘀 𝗦𝘁𝗮𝘁𝗲?

🧠 Concept: State represents dynamic data that changes over time within a component. When state changes, React re-renders the component to reflect the new data.

Example:

* State can store things like: counter values, input text, toggles, etc.
* State is local to the component where it is declared.

—

🔸 2. 𝘂𝘀𝗲𝗦𝘁𝗮𝘁𝗲 𝗦𝘆𝗻𝘁𝗮𝘅 & 𝗨𝘀𝗮𝗴𝗲

🧠 Concept: useState is a React Hook that lets you add state to functional components.

📌 Syntax:

const [state, setState] = useState(initialValue);

Example:

import React, { useState } from 'react';

function Counter() {

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

return (

<div>

<p>Count: {count}</p>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

}

✅ Rule: Hooks like useState must be called at the top level of a component (not inside loops, conditions, etc.).

—

🔸 3. 𝗠𝘂𝗹𝘁𝗶𝗽𝗹𝗲 𝗦𝘁𝗮𝘁𝗲𝘀

You can declare multiple useState hooks in a single component.

Example:

const [name, setName] = useState('');

const [age, setAge] = useState(25);

Each state hook manages a separate piece of data.

—

🔸 4. 𝗨𝗽𝗱𝗮𝘁𝗶𝗻𝗴 𝗦𝘁𝗮𝘁𝗲 𝗕𝗮𝘀𝗲𝗱 𝗼𝗻 𝗣𝗿𝗲𝘃𝗶𝗼𝘂𝘀 𝗩𝗮𝗹𝘂𝗲

If the new state depends on the old one, pass a callback to setState.

Example:

setCount((prevCount) => prevCount + 1);

📌 Why? Because state updates are asynchronous—using the previous value ensures accuracy.

—

🔸 5. 𝗘𝘅𝗲𝗿𝗰𝗶𝘀𝗲: 𝗖𝗼𝘂𝗻𝘁𝗲𝗿 𝗔𝗽𝗽

🎯 Objective: Build a simple counter with increment and reset buttons.

Example:

import React, { useState } from 'react';

function CounterApp() {

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

const handleReset = () => setCount(0);

return (

<div>

<h2>Count: {count}</h2>

<button onClick={() => setCount((prev) => prev + 1)}>Increment</button>

<button onClick={handleReset}>Reset</button>

</div>

);

}

💡 Tips:

* Try adding a decrement button.
* Display a message like “Negative count!” if count goes below 0.

—

**📘Chapter 5 : Event Handling**

This section explains how to handle user interactions such as clicks, typing, and form submissions in React using JSX.

🔸 1. 𝗛𝗮𝗻𝗱𝗹𝗶𝗻𝗴 𝗘𝘃𝗲𝗻𝘁𝘀 𝗶𝗻 𝗝𝗦𝗫

🧠 Concept: In React, event handling is similar to handling events in vanilla JavaScript, but with camelCase syntax and function references.

📌 Syntax Example:

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

Do not call the function directly (i.e., do not use onClick={handleClick()}).

Example:

function handleClick() {

alert('Button clicked!');

}

—

🔸 2. 𝗣𝗮𝘀𝘀𝗶𝗻𝗴 𝗣𝗮𝗿𝗮𝗺𝗲𝘁𝗲𝗿𝘀 𝘁𝗼 𝗘𝘃𝗲𝗻𝘁 𝗛𝗮𝗻𝗱𝗹𝗲𝗿𝘀

If you need to pass arguments to an event handler, use an arrow function.

Example:

function greet(name) {

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

}

<button onClick={() => greet('John')}>Greet</button>

—

🔸 3. 𝗛𝗮𝗻𝗱𝗹𝗶𝗻𝗴 𝗙𝗼𝗿𝗺𝘀 𝗮𝗻𝗱 𝗜𝗻𝗽𝘂𝘁𝘀

React uses controlled components, where input values are managed by state.

Example:

import React, { useState } from 'react';

function FormExample() {

const [name, setName] = useState('');

const handleChange = (e) => {

setName(e.target.value);

};

const handleSubmit = (e) => {

e.preventDefault();

alert(`Submitted name: ${name}`);

};

return (

<form onSubmit={handleSubmit}>

<input type="text" value={name} onChange={handleChange} />

<button type="submit">Submit</button>

</form>

);

}

🧾 Important:

* Use e.preventDefault() to stop the default form action.
* Use value and onChange to bind input state.

—

🔸 4. 𝗘𝘅𝗲𝗿𝗰𝗶𝘀𝗲: 𝗥𝗲𝗴𝗶𝘀𝘁𝗿𝗮𝘁𝗶𝗼𝗻 𝗙𝗼𝗿𝗺

🎯 Goal: Create a simple form with these inputs:

* Name
* Email
* Password
* Submit Button

📌 Add basic validation and display submitted data as an alert or console.log.

Starter Example:

import React, { useState } from 'react';

function RegistrationForm() {

const [formData, setFormData] = useState({

name: '',

email: '',

password: ''

});

const handleChange = (e) => {

const { name, value } = e.target;

setFormData((prev) => ({

...prev,

[name]: value

}));

};

const handleSubmit = (e) => {

e.preventDefault();

console.log('Form Submitted:', formData);

};

return (

<form onSubmit={handleSubmit}>

<input type="text" name="name" value={formData.name} onChange={handleChange} placeholder="Name" />

<input type="email" name="email" value={formData.email} onChange={handleChange} placeholder="Email" />

<input type="password" name="password" value={formData.password} onChange={handleChange} placeholder="Password" />

<button type="submit">Register</button>

</form>

);

}

✅ Try validating empty fields or displaying a confirmation message.

—

**📘 Chapter 6 : Conditional Rendering & Lists**

This section covers how to show different content based on conditions and how to render lists dynamically using map.

🔸 1. 𝗶𝗳/𝗲𝗹𝘀𝗲, 𝗧𝗲𝗿𝗻𝗮𝗿𝘆 𝗢𝗽𝗲𝗿𝗮𝘁𝗼𝗿

🧠 Concept: You can use standard JavaScript control flows like if/else or the ternary operator to render components conditionally.

Example using if/else:

if (isLoggedIn) {

return <Dashboard />;

} else {

return <Login />;

}

Example using ternary operator:

return (

<div>

{isLoggedIn ? <Dashboard /> : <Login />}

</div>

);

—

🔸 2. 𝗟𝗼𝗴𝗶𝗰𝗮𝗹 𝗔𝗡𝗗 (&&)

Use && when you want to conditionally show content only when a condition is true.

Example:

{isAdmin && <button>Delete User</button>}

📌 It’s a shorthand for: if (isAdmin) { show the button }

—

🔸 3. 𝗥𝗲𝗻𝗱𝗲𝗿𝗶𝗻𝗴 𝗟𝗶𝘀𝘁𝘀 𝘄𝗶𝘁𝗵 .𝗺𝗮𝗽()

Use .map() to dynamically render lists in React.

Example:

const fruits = ['Apple', 'Banana', 'Orange'];

return (

<ul>

{fruits.map((fruit, index) => (

<li key={index}>{fruit}</li>

))}

</ul>

);

—

🔸 4. 𝗨𝗻𝗶𝗾𝘂𝗲 𝗞𝗲𝘆𝘀 𝗶𝗻 𝗟𝗶𝘀𝘁 𝗥𝗲𝗻𝗱𝗲𝗿𝗶𝗻𝗴

📌 Keys help React identify which items have changed, added, or removed.

Best Practice:

* Use a unique id from the data (not array index if items can be reordered).
* Never use random values as keys.

Example:

const tasks = [

{ id: 1, title: 'Do laundry' },

{ id: 2, title: 'Buy groceries' }

];

return (

<ul>

{tasks.map(task => (

<li key={task.id}>{task.title}</li>

))}

</ul>

);

—

🔸 5. 𝗘𝘅𝗲𝗿𝗰𝗶𝘀𝗲: 𝗧𝗼𝗱𝗼 𝗔𝗽𝗽

🎯 Goal: Build a simple to-do app with these features:

* Input field to add new tasks
* Button to add task
* Display tasks in a list
* Ability to delete tasks

Starter Example:

import React, { useState } from 'react';

function TodoApp() {

const [task, setTask] = useState('');

const [todos, setTodos] = useState([]);

const handleAdd = () => {

if (task.trim()) {

setTodos([...todos, { id: Date.now(), title: task }]);

setTask('');

}

};

const handleDelete = (id) => {

setTodos(todos.filter(todo => todo.id !== id));

};

return (

<div>

<input type="text" value={task} onChange={(e) => setTask(e.target.value)} placeholder="New task" />

<button onClick={handleAdd}>Add</button>

<ul>

{todos.map(todo => (

<li key={todo.id}>

{todo.title}

<button onClick={() => handleDelete(todo.id)}>Delete</button>

</li>

))}

</ul>

</div>

);

}

✅ Bonus Challenge:

* Add checkbox to mark tasks as done
* Highlight or strike-through completed tasks

—

**📘 Chapter 7: useEffect Hook (Side Effects)**

“Not all code belongs in the UI. Side effects let React interact with the world outside of rendering.”

**🔹 1. ❓ What is a Side Effect?**

A side effect is any operation that affects something outside the scope of the current function, such as:

* Fetching data from an API
* Manipulating the DOM directly
* Setting up a timer or subscription
* Logging to the console

⚠️ In React, side effects are not handled inside the main component body — they go inside useEffect.

**🔹 2. ⚙️ useEffect Syntax & Usage**

React provides the useEffect hook to perform side effects in function components.

**Basic Syntax:**

import { useEffect } from 'react';

useEffect(() => {

*// Side effect logic here*

} [Dependencies] );

👀 By default, this runs after every render (including the first time).

**🔹 3. 🧩 useEffect with Dependency Array**

You can control when useEffect runs using the dependency array.

**Examples:**

✅ Run only on mount (like componentDidMount):

useEffect(() => {

console.log("Component mounted");

}, []);

✅ Run when a specific variable changes:

useEffect(() => {

console.log("User updated!");

}, [user]);

⚠️ If the dependency changes, useEffect will re-run.

**🔹 4. 🧹 Cleanup Functions**

Cleanup prevents memory leaks by tearing down timers, subscriptions, etc., before the component unmounts or re-runs the effect.

**Example:**

useEffect(() => {

const timer = setInterval(() => {

console.log("Tick");

}, 1000);

return () => {

clearInterval(timer); *// cleanup*

};

}, []);

🔄 This mimics componentWillUnmount.

**🔹 5. 📝 Exercise: Live Clock / Fetch API on Load**

Try implementing one of these projects:

**⏰ Live Clock:**

* Set up a timer using setInterval inside useEffect
* Update current time every second
* Use cleanup to clear the interval on unmount

**🌐 Fetch API on Component Load:**

* Use fetch or axios inside useEffect with empty dependency array
* Display data (e.g., user info or posts)
* Handle loading and error states

✅ You’ve now learned how to use useEffect to:

* Run side effects
* Respond to changes
* Clean up resources

Happy Coding! 🚀

**📘 Chapter 8: useRef Hook**

“Think of useRef as a tiny box that can store a mutable value without causing a re-render.”

**🔹 1. 🔍 useRef vs DOM Access**

React discourages direct DOM manipulation, but useRef provides a safe way to reference DOM elements.

🧠 useRef returns a mutable object:

const refContainer = useRef(initialValue);

The object persists across re-renders and can store either a DOM node or any mutable value.

**🔹 2. 🎯 useRef for Focus/Input Fields**

You can assign useRef to an input and programmatically set focus.

**Example:**

import { useRef } from "react";

function FocusInput() {

const inputRef = useRef(null);

const handleFocus = () => {

inputRef.current.focus();

};

return (

<>

<input ref={inputRef} type="text" placeholder="Click the button to focus" />

<button onClick={handleFocus}>Focus Input</button>

</>

);

}

✅ useRef gives direct access to the input element (inputRef.current).

**🔹 3. 🕘 useRef for Storing Previous Values**

You can use useRef to track the previous value of a prop or state.

**Example:**

const count = useState(0);

const prevCount = useRef();

useEffect(() => {

prevCount.current = count;

}, [count]);

<p>Previous Count: {prevCount.current}</p>

🧠 This helps in comparing current vs. previous values across renders.

**🔹 4. ⏱ useRef for Timer/Interval**

useRef is also useful for storing timer IDs to start/stop intervals.

**Example:**

const timerRef = useRef(null);

useEffect(() => {

timerRef.current = setInterval(() => {

console.log("Tick");

}, 1000);

return () => clearInterval(timerRef.current);

}, []);

⚙️ This pattern is common in games, clocks, and animations.

**🔹 5. 📝 Exercise: Input Focus on Button Click**

Build a small interactive task:

✅ Features:

* One input field
* One button
* On clicking the button, focus the input

💡 Bonus: Display a message saying “Focused!” after the click using a useState flag.

🧠 Summary

* useRef = Mutable container
* Can point to DOM or values across renders
* Doesn’t trigger re-render
* Ideal for focus, timers, and tracking previous values

Happy coding with useRef! 🚀

**📘 Chapter 9: useContext Hook (Global State)**

“Tired of passing props down the component tree? Welcome to useContext – your solution to global state sharing in React!”

**🔹 1. 😣 Prop Drilling Problem**

🧠 Prop Drilling = Passing data through multiple layers of components unnecessarily.

❌ Pain Points:

* Unmanageable in large applications
* Components receive props they don’t even use
* Hard to refactor and maintain

🎯 Solution: Context API – Share data without drilling through every level.

**🔹 2. 🏗 Creating a Context**

React provides a way to create a context using:

const MyContext = React.createContext(defaultValue);

✅ This creates a context object with two components:

* MyContext.Provider — used to wrap your component tree
* MyContext.Consumer or useContext() — used to read context

**🔹 3. 🔌 Using Context Provider & useContext**

✅ Context Provider makes data available to child components.

**Example:**

*// context.js*

const ThemeContext = React.createContext();

*// App.jsx*

<ThemeContext.Provider value={"dark"}>

<Toolbar />

</ThemeContext.Provider>

*// Toolbar.jsx*

const theme = useContext(ThemeContext);

console.log(theme); *// "dark"*

🧠 useContext(context) allows functional components to read the value.

📌 Note: The component using useContext must be a child of the corresponding Provider.

**🔹 4. 🧐 When to Use Context API**

Use Context API for:

* 🌍 Global state (theme, user, language, etc.)
* 🚦 App-level configuration
* 🔐 Authentication context
* 🧱 Avoiding prop drilling in deep component trees

❗ Don’t overuse! For complex state, use Redux or Zustand.

**🔹 5. 📝 Exercise: Theme Switcher App**

🛠 Build a Theme Switcher:

✅ Requirements:

* Create a ThemeContext with “light” / “dark”
* Add a toggle button to change the theme
* Pass theme value to styled components
* Display the current theme in the header

💡 Bonus: Store theme in localStorage for persistence.

🧠 Summary

* useContext helps you avoid prop drilling
* Combine with Provider for global value sharing
* Simple and effective for small-to-medium scale apps

🌀 Level up your React state management!

**📘 Chapter 10: useReducer Hook**

“When your state logic grows too complex for useState, it’s time to bring in the reducer!”

**🔹 1. 🧠 What is a Reducer?**

A reducer is a function that determines changes to an application’s state. It takes two arguments:

* The current state
* An action

It returns a new state based on the action type.

const reducer = (state, action) => {

switch (action.type) {

case "increment":

return { count: state.count + 1 };

default:

return state;

}

};

Think of it like a state “controller” — clean, centralized, and predictable.

**🔹 2. ⚖️ useReducer vs useState**

| **Feature** | **useState** | **useReducer** |
| --- | --- | --- |
| Simplicity | Great for simple states | Better for complex logic or multiple states |
| State Logic | Inline | Centralized in reducer function |
| Predictability | Less explicit | More controlled with switch/actions |
| Performance | Not optimized for batching | Better for advanced state transitions |

✅ Choose useReducer when:

* You have multiple related state variables
* State updates depend on previous state
* You need more structure

**🔹 3. 🏗 Creating Reducer Functions**

React Hook Syntax:

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

Example:

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 };

default:

return state;

}

}

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

Usage:

<button onClick={() => dispatch({ type: "increment" })}>+</button>

**🔹 4. 🧪 Exercise: Complex Counter with Actions**

🎯 Build a counter app with the following actions:

* ➕ Increment
* ➖ Decrement
* 🔁 Reset
* 🎲 Add custom number from input

Use a reducer to manage all state transitions!

**🔹 5. 🛠 Mini Project: Todo App with useReducer**

🧱 Features:

* Add todo
* Delete todo
* Toggle completion
* Filter completed/pending
* Store todos in an array inside state

🎯 Steps:

1. Define initialState: { todos: [] }
2. Create actions: ADD\_TODO, DELETE\_TODO, TOGGLE\_TODO
3. Use reducer to handle all updates
4. Render list using .map()
5. Add unique keys and conditionally render styles

💡 Bonus: Persist todos in localStorage

🚀 Summary

* useReducer = predictable state management
* Best for complex logic & multiple state transitions
* Pairs great with useContext for global state (like Redux-lite)

🧠 Master useReducer to level up your React state skills!

**📘 Chapter 11: useMemo Hook**

“Optimize performance, avoid unnecessary calculations — let useMemo do the work.”

**🔹 1. 🧠 What is Memoization?**

Memoization is a performance optimization technique that stores the result of expensive function calls and returns the cached result when the same inputs occur again.

In React, useMemo helps you avoid recalculating values unless dependencies change.

**🔹 2. 🚀 Preventing Expensive Recalculations**

Without useMemo, every render re-runs your expensive function — even when not needed.

🛑 Problem:

const expensiveValue = calculateHeavyTask(input); *// runs on every render*

✅ Solution:

const memoizedValue = useMemo(() => calculateHeavyTask(input), [input]);

React will recalculate only when the input changes — saving time and memory.

**🔹 3. 📘 useMemo Syntax & Examples**

Basic Syntax:

const memoizedValue = useMemo(() => {

return computeExpensiveValue(a, b);

}, [a, b]);

Example: Optimizing a slow Fibonacci calculator

const fib = (n) => {

if (n <= 1) return 1;

return fib(n - 1) + fib(n - 2);

};

const result = useMemo(() => fib(num), [num]);

🧠 Rule of Thumb: Only use useMemo when a computation is:

* Expensive (e.g. recursion, filtering, sorting)
* Causing performance issues
* Inside frequently re-rendered components

**🔹 4. 🧪 Exercise: Fibonacci Calculator with useMemo**

🎯 Build a simple calculator that takes a number input and displays the Fibonacci result.

Steps:

1. Create input for number
2. Display computed Fibonacci result
3. Use useMemo to optimize expensive recursive calculation
4. Add a counter to test unnecessary re-renders

Optional:

* Add a loading spinner before the result appears
* Show time taken to calculate (using performance.now())

🧠 Summary

* useMemo = avoid recomputation for performance
* Only re-runs when dependencies change
* Especially useful with expensive logic or slow renders

💡 Think of it as a React-level cache that improves efficiency.

Happy optimizing! ⚡

**📘 Chapter 12: useCallback Hook**

“Functions shouldn’t trigger re-renders unless necessary — that’s where useCallback comes in.”

**🔹 1. 🔄 Preventing Re-Renders of Child Components**

In React, functions are re-created on every render. This can cause unnecessary re-renders of child components, especially when functions are passed as props.

🛑 Problem:

<Child onClick={() => console.log('Clicked')} />

This function is recreated every time the parent renders, causing to re-render even if it didn’t need to.

✅ Solution:

const handleClick = useCallback(() => {

console.log('Clicked');

}, []);

<Child onClick={handleClick} />

Now the same function instance is reused until dependencies change.

**🔹 2. 🧠 useCallback vs useMemo**

| **Feature** | **useCallback** | **useMemo** |
| --- | --- | --- |
| Purpose | Memoize functions | Memoize computed values |
| Returns | Memoized callback function | Memoized result (any value) |
| Use case | Passing stable functions to children | Avoid recalculating expensive values |

**🔹 3. 📤 Passing Stable Functions to Children**

If a child component depends on reference equality to avoid re-rendering, use useCallback to memoize the function.

Example:

const increment = useCallback(() => {

setCount(c => c + 1);

}, []);

<ChildButton onClick={increment} />

If you don’t memoize increment, may re-render unnecessarily.

✅ useCallback ensures function identity is preserved across renders (until dependencies change).

**🔹 4. 🧪 Exercise: Optimized Button Click Tracker**

🎯 Objective: Build a parent component that tracks clicks and passes an increment function to a memoized child.

Steps:

1. Create a count state using useState
2. Create a memoized function using useCallback to increment count
3. Pass the function to a memoized child button component
4. Use React.memo on the child to prevent re-renders unless props change

Bonus:

* Add console logs to see if the child re-renders unnecessarily
* Try removing useCallback and observe the effect

🧠 Summary

* useCallback = memoize functions
* Avoids unnecessary re-renders of child components
* Essential when passing functions to deeply nested or memoized components
* Keeps your app performant, especially in large trees

💡 Remember: useCallback(fn, deps) === useMemo(() => fn, deps)

Keep it efficient, keep it fast. 🚀

**📘 Chapter 13: Custom Hooks**

“Write once. Reuse everywhere. Let your logic live its best life.”

**🔹 1. 💡 What are Custom Hooks?**

Custom Hooks are JavaScript functions that start with the word use and can call other Hooks inside them.  
They allow you to extract and reuse component logic.

✅ Why use them?

* Reuse logic across components
* Keep components clean and focused on UI
* Separate concerns: business logic in hooks, presentation in components

**🔹 2. 🔁 Creating Reusable Hook Functions**

A Custom Hook is just a function that:

* Starts with use
* Uses other React hooks (like useState, useEffect)
* Returns something useful (state, handlers, etc.)

Example: useCounter

import { useState } from 'react';

function useCounter(initialValue = 0) {

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

const increment = () => setCount(c => c + 1);

const decrement = () => setCount(c => c - 1);

const reset = () => setCount(initialValue);

return { count, increment, decrement, reset };

}

Usage:

const { count, increment, decrement, reset } = useCounter(10);

**🔹 3. 🏷 Naming Conventions (useXYZ)**

* Always prefix with use
* Keep it semantic (e.g., useTimer, useFetch, useForm, useLocalStorage)
* Follows the Hook Rules: only call from top-level and from inside functional components or other hooks

✅ Good: useFormData ❌ Bad: fetchDataHook

**🔹 4. 🛠 Example Custom Hooks**

**🗃 useLocalStorage**

function useLocalStorage(key, initialValue) {

const [value, setValue] = useState(() => {

const stored = localStorage.getItem(key);

return stored ? JSON.parse(stored) : initialValue;

});

useEffect(() => {

localStorage.setItem(key, JSON.stringify(value));

}, [key, value]);

return [value, setValue];

}

**⏱ useTimer**

function useTimer(start = 0) {

const [seconds, setSeconds] = useState(start);

useEffect(() => {

const interval = setInterval(() => setSeconds(s => s + 1), 1000);

return () => clearInterval(interval);

}, []);

return seconds;

}

**🔹 5. 📝 Exercise: Build Your Own useCounter Hook**

🎯 Goal: Create a reusable useCounter hook that provides:

* count state
* increment function
* decrement function
* reset function

Try:

* Adding a step value
* Setting an upper and lower limit
* Using localStorage to persist count

💡 Challenge: Build a stopwatch with useCounter + useEffect

🧠 Summary

* Custom Hooks = Powerful way to reuse logic
* Clean up your components
* Improve maintainability & readability
* Naming matters: always start with useXYZ

Hook it once. Use it everywhere. 🔁✨

**📘 Chapter 14: React Router (v6+)**

“Navigation made declarative. Let users explore your app—smoothly.”

**🔹 1. 🚦 Setting Up React Router**

Install React Router:

npm install react-router-dom

Wrap your app with in index.js or App.jsx:

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

import App from "./App";

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

root.render(

<BrowserRouter>

<App />

</BrowserRouter>

);

**🔹 2. 🛣️ , , and**

React Router v6 introduced a simpler syntax.

Example:

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

import Home from "./pages/Home";

import About from "./pages/About";

function App() {

return (

<Routes>

<Route path="/" element={<Home />} />

<Route path="/about" element={<About />} />

</Routes>

);

}

✅ Use element prop (not component).

**🔹 3. 🔍 useParams, useNavigate, useLocation**

* useParams(): Access route parameters

const { id } = useParams(); *// /post/:id*

* useNavigate(): Programmatic navigation

const navigate = useNavigate();

navigate("/login");

* useLocation(): Get info about current URL

const location = useLocation();

console.log(location.pathname);

**🔹 4. 🧩 Route Nesting**

You can create nested layouts easily.

Example:

<Route path="/dashboard" element={<DashboardLayout />}>

<Route path="profile" element={<Profile />} />

<Route path="settings" element={<Settings />} />

</Route>

Resulting paths:

* /dashboard/profile
* /dashboard/settings

✅ is used in DashboardLayout to render child routes.

**🔹 5. 🔐 Protected Routes (Basic Auth Guard)**

Create a wrapper for routes that require authentication:

function ProtectedRoute({ children }) {

const isLoggedIn = true; *// Replace with real logic*

return isLoggedIn ? children : <Navigate to="/login" />;

}

Usage:

<Route path="/dashboard" element={<ProtectedRoute><Dashboard /></ProtectedRoute>} />

**🔹 6. 📝 Mini Project: Multi-Page Blog App**

🧩 Pages:

* Home
* About
* Blog List
* Blog Detail (/blog/:id)

🎯 Features:

* Navigation menu
* Route-based content rendering
* useParams to fetch post by ID
* Protected route for admin page

💡 Bonus:

* Add 404 Page: <Route path=“\*” element={} />
* Use useNavigate for redirect after login

🚀 Summary

* React Router makes SPA navigation clean and declarative
* v6 uses + element prop (not render or component)
* Programmatic navigation = useNavigate
* Params = useParams, Location = useLocation
* Secure routes with ProtectedRoute component

📚 Keep routing your way through awesome UIs!

**📘 Chapter 15: Forms and Validation**

“Let users interact and submit data—cleanly and reliably.”

**🔹 1. ✍️ Controlled Inputs**

In React, form inputs are usually controlled components—meaning their value is managed by state.

✅ Example:

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

<input

type="text"

value={name}

onChange={(e) => setName(e.target.value)}

/>

📌 State drives the UI, and the UI updates the state.

**🔹 2. 📝 Form Handling**

Create a form and prevent the default browser behavior:

function handleSubmit(e) {

e.preventDefault(); *// stop page reload*

console.log("Form submitted!");

}

✅ Usage in JSX:

<form onSubmit={handleSubmit}>

<input type="text" value={name} onChange={...} />

<button type="submit">Submit</button>

</form>

**🔹 3. ⚠️ Basic Validation with useState**

You can validate form inputs using simple conditions and useState.

✅ Example:

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

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

function handleSubmit(e) {

e.preventDefault();

if (!email.includes("@")) {

setError("Invalid email");

} else {

setError("");

console.log("Form submitted!");

}

}

🎯 Show error message in UI:

{error && <p style={{ color: "red" }}>{error}</p>}

**🔹 4. 🛠 Libraries: Formik + Yup (Optional)**

For complex forms, you can use:

* 🧰 Formik – simplifies form state management
* ✅ Yup – for schema-based form validation

Install:

npm install formik yup

Basic Formik + Yup usage:

<Formik

initialValues={{ email: "" }}

validationSchema={Yup.object({

email: Yup.string().email().required("Email is required"),

})}

onSubmit={(values) => console.log(values)}

>

{({ handleChange, handleSubmit, values, errors }) => (

<form onSubmit={handleSubmit}>

<input name="email" onChange={handleChange} value={values.email} />

{errors.email && <p>{errors.email}</p>}

</form>

)}

</Formik>

**🔹 5. 📝 Exercise: Signup Form with Error Handling**

✅ Build a Signup Form with:

* Name, Email, Password fields
* useState for field values
* Validation:
  + Name must not be empty
  + Email must include “@”
  + Password must be 6+ characters
* Show error messages below each field
* Submit button triggers validation + prints values

Bonus:

* Clear form after successful submission
* Add Formik + Yup for cleaner validation (optional)

✅ Summary

* Controlled inputs bind form fields to state
* You can validate inputs manually or using libraries
* Formik + Yup = best friends for advanced forms
* Always provide user feedback (error or success)

🔐 Great forms = better user experience!

**📘 Chapter 16: API Integration (fetch & Axios)**

“React + APIs = Dynamic, real-time web apps.”

Learn how to fetch data from APIs, display results, and handle loading and error states.

**🔹 1. 🌐 Using fetch()**

JavaScript’s built-in fetch() is commonly used for making HTTP requests.

✅ Basic GET request:

useEffect(() => {

fetch("https://jsonplaceholder.typicode.com/users")

.then(response => response.json())

.then(data => setUsers(data))

.catch(error => console.error("Error:", error));

}, []);

📌 Always wrap fetch in useEffect to avoid infinite loops.

**🔹 2. 📦 Using Axios (GET, POST, PUT, DELETE)**

Axios is a promise-based HTTP client that simplifies requests.

🔸 Install:

npm install axios

🔸 Usage Examples:

* ✅ GET:

axios.get("/api/users")

.then(res => setUsers(res.data))

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

* ✅ POST:

axios.post("/api/users", { name: "John" })

.then(res => console.log("User Added"))

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

* ✅ PUT:

axios.put("/api/users/1", { name: "Updated" })

* ✅ DELETE:

axios.delete("/api/users/1")

**🔹 3. ⏳ Displaying Loading / Error / Success**

Good UI should reflect request status.

✅ Example:

const [loading, setLoading] = useState(true);

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

const [users, setUsers] = useState([]);

useEffect(() => {

axios.get("https://jsonplaceholder.typicode.com/users")

.then(res => {

setUsers(res.data);

setLoading(false);

})

.catch(err => {

setError("Something went wrong");

setLoading(false);

});

}, []);

🔹 Conditional Rendering:

if (loading) return <p>Loading...</p>;

if (error) return <p>{error}</p>;

return <UserList users={users} />;

**🔹 4. 📝 Exercise: Fetch Users from JSONPlaceholder**

Build a component that:

* Fetches users from <https://jsonplaceholder.typicode.com/users>
* Displays their name, email, and username
* Shows “Loading…” while fetching
* Shows an error message if the fetch fails

Bonus:

* Use Axios instead of fetch
* Add a search bar to filter users

**🔹 5. 🌦 Mini Project: Weather App**

Create a weather dashboard using a public weather API (e.g., OpenWeatherMap):

✅ Features:

* Input field for city name
* Button to fetch weather
* Show temperature, humidity, description
* Loading and error messages
* Use Axios or fetch
* Optional: Add icons and styling for weather condition

✅ Summary

* fetch() is built-in but Axios provides a cleaner syntax
* Always show loading and error states
* Practice with JSONPlaceholder before using real APIs
* Real-time data brings your React app to life! 💡

Happy fetching! 🚀

**📘 Chapter 17: Component Communication**

“In React, data flows down. But how do components talk to each other?”

Understanding how React components share data and actions is key to building interactive UIs.

**🔹 1. 🧭 Parent to Child (Props)**

Props are the primary way to pass data from a parent component to a child.

✅ Example:

function Welcome({ name }) {

return <h2>Hello, {name}!</h2>;

}

function App() {

return <Welcome name="John" />;

}

🧠 Key Idea: Props are read-only — children cannot modify them.

**🔹 2. 🔄 Child to Parent (Callback Props)**

To send data or events back to the parent, pass a function (callback) as a prop.

✅ Example:

function Child({ onButtonClick }) {

return <button onClick={() => onButtonClick("Hello from Child")}>Click Me</button>;

}

function Parent() {

const handleChildMessage = (msg) => {

console.log(msg);

};

return <Child onButtonClick={handleChildMessage} />;

}

📌 This enables child-to-parent communication through function calls.

**🔹 3. 🤝 Sibling Communication (Lifting State Up)**

Siblings can’t talk directly — so lift the shared state to their common parent.

✅ Example:

function SiblingA({ setValue }) {

return <button onClick={() => setValue("A clicked")}>A</button>;

}

function SiblingB({ value }) {

return <p>Received: {value}</p>;

}

function Parent() {

const [value, setValue] = useState("");

return (

<>

<SiblingA setValue={setValue} />

<SiblingB value={value} />

</>

);

}

🔼 State is lifted up to the parent and passed down to both children.

**🔹 4. 📝 Exercise: Like/Dislike Button Sync**

🎯 Build two components: LikeButton and DislikeButton.

Requirements:

* Shared state for total likes/dislikes
* Display count in parent
* Update count from children via callback props
* Add conditional rendering for button disable (e.g., user can’t like twice)

Hint: Use useState and lift state to the parent component.

✅ Summary

* 🔽 Parent → Child: via props
* 🔼 Child → Parent: via callback functions
* ↔️ Sibling → Sibling: lift shared state up
* Shared state is the key to synchronized communication.

React is powerful when your components talk clearly. 🎙️

**📘 Chapter 18: State Management with Context + useReducer**

“For complex apps, prop-drilling becomes painful. That’s where combining Context with useReducer shines.”

This chapter covers how to build global state logic using React’s built-in Context API and the useReducer hook — without needing external libraries like Redux.

**🔹 1. 🔗 Combining useContext + useReducer**

🧠 Why Combine?

* useReducer → handles complex state transitions
* useContext → shares the state across components

Together, they allow global state management in React with clean separation of logic.

✅ Setup Flow:

1. Create a context
2. Define a reducer function
3. Create a provider with useReducer
4. Use useContext to access state/actions anywhere

🧩 Example:

*// context/TodoContext.js*

import { createContext, useReducer } from 'react';

export const TodoContext = createContext();

const initialState = [];

function reducer(state, action) {

switch (action.type) {

case 'ADD\_TODO':

return [...state, { text: action.payload, id: Date.now() }];

case 'REMOVE\_TODO':

return state.filter(todo => todo.id !== action.payload);

default:

return state;

}

}

export const TodoProvider = ({ children }) => {

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

return (

<TodoContext.Provider value={{ state, dispatch }}>

{children}

</TodoContext.Provider>

);

};

**🔹 2. 🌐 Global State Management Example**

Anywhere in your app, you can use:

import { useContext } from 'react';

import { TodoContext } from './context/TodoContext';

function TodoList() {

const { state, dispatch } = useContext(TodoContext);

return (

<ul>

{state.map(todo => (

<li key={todo.id}>

{todo.text}

<button onClick={() => dispatch({ type: 'REMOVE\_TODO', payload: todo.id })}>

❌

</button>

</li>

))}

</ul>

);

}

📌 The Context + useReducer setup keeps logic centralized and avoids prop drilling.

**🔹 3. 🛠️ Mini Project: Global Todo App**

Build a fully functional global Todo App.

💡 Features:

* Add new tasks
* Delete tasks
* Centralized reducer logic
* Global access via Context API

📂 Suggested File Structure:

/src

├── context/

│ └── TodoContext.js

├── components/

│ ├── TodoInput.js

│ └── TodoList.js

└── App.js

💬 Bonus: Add filters (e.g., completed/pending) and persistent storage with localStorage.

✅ Summary

* useReducer gives predictable state transitions
* useContext shares that state globally
* Together they create a lightweight, scalable global state solution

This pattern is perfect for medium-scale React apps 🚀

**📘 Chapter 19: Performance Optimization**

“React is fast — but with the right patterns, we can make it even faster.”

This chapter explores how to optimize your React app for better speed, rendering performance, and responsiveness using built-in tools like memoization, lazy loading, and more.

**🔹 1. 🚀 React.memo**

🔍 What it does:

* React.memo is a higher-order component (HOC) that memoizes a functional component.
* It prevents re-renders if props haven’t changed.

✅ Usage:

import React from 'react';

const MyComponent = React.memo(({ name }) => {

console.log('Rendering...');

return <h1>Hello, {name}</h1>;

});

📌 React will skip re-rendering MyComponent if the name prop hasn’t changed.

**🔹 2. 🧠 useMemo & useCallback Review**

🧠 useMemo

* Memoizes the result of a computation.
* Use it to avoid expensive recalculations.

Example:

const expensiveValue = useMemo(() => {

return heavyCalculation(input);

}, [input]);

🧠 useCallback

* Memoizes a function definition.
* Useful when passing callbacks to child components.

Example:

const handleClick = useCallback(() => {

console.log('Clicked!');

}, []);

🛠 useCallback is often used with React.memo to prevent child re-renders.

**🔹 3. ⏳ Lazy Loading with React.lazy() and Suspense**

📦 React.lazy()

* Dynamically import components when needed.

const LazyComponent = React.lazy(() => import('./LazyComponent'));

🌀 Suspense

* Used to wrap lazy-loaded components and show a fallback UI.

<Suspense fallback={<p>Loading...</p>}>

<LazyComponent />

</Suspense>

✅ This technique improves initial load time for large apps.

**🔹 4. 🛑 Avoiding Unnecessary Re-Renders**

📌 Key Tips:

* Lift state only when needed — don’t hoist unnecessarily.
* Avoid anonymous functions in render if possible.
* Memoize functions or values passed to deeply nested components.
* Break large components into smaller ones to isolate renders.

🛠 Tools to inspect performance:

* React DevTools Profiler tab
* Console.logs for tracking render behavior
* useWhyDidYouUpdate (debug utility)

✅ Summary

* Use React.memo to avoid re-renders of pure components.
* useMemo and useCallback are essential for memoizing values/functions.
* Lazy load large components with React.lazy and Suspense.
* Write clean, isolated components and avoid lifting state unnecessarily.

Performance matters — especially as your app scales. Optimize smartly! 💡

**📘 Chapter 20: Deployment**

“Build once. Deploy anywhere.”  
This chapter will walk you through building and deploying your React application to the web using popular platforms.

**🚀 1. npm run build**

🛠️ What it does:

* Compiles your app into static files for production.
* Optimizes assets (minification, bundling, tree-shaking).

✅ Command:

npm run build

📁 Output: Creates a /build folder containing HTML, CSS, JS files optimized for deployment.

⚠️ Always test the build locally before deploying:

npx serve -s build

**🌐 2. Hosting on Netlify, Vercel, Firebase**

**🔹 Netlify**

* Connect GitHub repo or drag & drop the /build folder.
* Auto-deploys on git push.
* Set build command: npm run build Publish directory: build

➡️ Site URL: your-project.netlify.app

**🔹 Vercel**

* Ideal for React apps with serverless functions.
* Connect Git repo → auto deploy.
* Detects React & sets config automatically.

➡️ Site URL: your-project.vercel.app

**🔹 Firebase Hosting**

Install tools:

npm install -g firebase-tools

firebase login

firebase init

Set public directory to build Deploy with:

firebase deploy

➡️ Site URL: your-project.web.app

**🧪 3. Using Environment Variables**

💡 React uses variables that start with REACT\_APP\_

Create a .env file in your root:

REACT\_APP\_API\_URL=https://api.example.com

Access in code:

const apiUrl = process.env.REACT\_APP\_API\_URL;

⚠️ Never include secrets in .env files committed to Git.

**✅ 4. Deployment Tips**

🔐 Security:

* Avoid exposing sensitive data.
* Use HTTPS endpoints.

🧹 Clean Code:

* Remove console.logs and unused code before build.

📄 SEO & Meta Tags:

* Add proper titles and descriptions for your site.
* Use react-helmet for dynamic meta tags.

🧪 Test:

* Test your production build for bugs, responsiveness, and performance.

🛠 Tools:

* Lighthouse (Chrome DevTools) for performance audit.
* Netlify/Vercel dashboard for analytics and error logging.

🎉 Congratulations! Your app is now live and ready to impress the world. 🚀

**Cheat Sheet**

📘 React Complete Cheat Sheet (Chapters 1–20)

🟦 Chapter 1: Introduction to React

* React is a JavaScript library for building UIs.
* Uses reusable components, virtual DOM, and one-way data flow.

Example:

function Welcome() {

return <h1>Hello, React!</h1>;

}

🟦 Chapter 2: Setting Up React Project

* Use Create React App: npx create-react-app my-app
* File structure: /src, /public, etc.

🟦 Chapter 3: Components & Props

* Functional vs Class Components
* Props for data passing
* Composition with nested components

Example:

function Hello(props) {

return <h1>Hello, {props.name}</h1>;

}

🟦 Chapter 4: useState Hook

* useState for local component state

Example:

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

🟦 Chapter 5: Event Handling

* onClick, onChange etc.
* Pass parameters using arrow functions

🟦 Chapter 6: Conditional Rendering & Lists

* if/else, ternary, &&
* Render lists with .map()
* Use key prop

🟦 Chapter 7: useEffect Hook

* Perform side effects (API, timers, etc.)
* Dependency array for control

Example:

useEffect(() => {

fetchData();

}, []);

🟦 Chapter 8: useRef Hook

* Access DOM elements
* Store previous values, timers

🟦 Chapter 9: useContext Hook

* Avoid prop drilling
* Create + Provide + useContext

🟦 Chapter 10: useReducer Hook

* State logic via reducer functions
* Useful for complex state

Example:

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

🟦 Chapter 11: useMemo Hook

* Memoize expensive calculations

Example:

const value = useMemo(() => compute(num), [num]);

🟦 Chapter 12: useCallback Hook

* Memoize functions to prevent re-renders

Example:

const handleClick = useCallback(() => doSomething(), []);

🟦 Chapter 13: Custom Hooks

* Reusable hooks prefixed with use

Example:

function useCounter() {

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

return { count, increment: () => setCount(count + 1) };

}

🟦 Chapter 14: React Router (v6+)

* , ,
* useNavigate, useParams
* Nested & Protected Routes

🟦 Chapter 15: Forms and Validation

* Controlled components
* Validation using useState or libraries like Formik/Yup

🟦 Chapter 16: API Integration

* Use fetch() or axios
* Handle loading, error, success states

🟦 Chapter 17: Component Communication

* Parent → Child: props
* Child → Parent: callback props
* Sibling: Lift state up

🟦 Chapter 18: Context + useReducer

* Combine for global state management

🟦 Chapter 19: Performance Optimization

* React.memo, useMemo, useCallback
* Lazy loading with React.lazy and Suspense

🟦 Chapter 20: Deployment

* npm run build
* Host on Netlify, Vercel, Firebase
* Use .env files for config

🟨 Bonus: All Hooks Quick List

* useState – Local state
* useEffect – Side effects
* useContext – Global state
* useReducer – Complex state
* useRef – Access DOM, persistent values
* useMemo – Memoize values
* useCallback – Memoize functions
* Custom Hooks – Reusable logic

Happy Coding! 🚀

📘 Top 50 React Interview Questions

🟢 Beginner (Basics – 1 to 15)

1. What is React and what problem does it solve?
2. What is the difference between React and other JavaScript frameworks?
3. What is JSX?
4. How does JSX get compiled?
5. What are components in React?
6. Difference between Functional and Class components?
7. What are props in React?
8. What is state in React?
9. How does setState work?
10. What is a controlled vs uncontrolled component?
11. What are default props?
12. What is prop drilling?
13. How can you prevent prop drilling?
14. What is the role of keys in lists?
15. Why shouldn’t you use array index as a key?

🟡 Intermediate (16 to 35)

1. What are React Hooks?
2. How does useState work?
3. What is useEffect used for?
4. How do you handle side effects in React?
5. Explain dependency array in useEffect.
6. What are cleanup functions in useEffect?
7. What is useRef and when would you use it?
8. How can useRef be used to store previous values?
9. How does useContext work?
10. How do you create and provide a context?
11. When should you use Context API vs props?
12. How does useReducer differ from useState?
13. What is lazy initialization in useState/useReducer?
14. What is useParams and useNavigate?
15. What are nested routes?
16. How would you implement protected routes?
17. How do you handle forms in React?
18. What are controlled inputs?
19. How would you validate a form?

🔴 Advanced (36 to 50)

1. What is useMemo and why would you use it?
2. What is React.memo?
3. How do you prevent unnecessary re-renders?
4. What are custom hooks?
5. How do you build a custom hook?
6. What naming convention should you follow for hooks?
7. When would you use useLayoutEffect over useEffect?
8. What is React Suspense?
9. How does React.lazy work?
10. How do you fetch data from an API in React?
11. What is the difference between fetch and axios?

📘 Top 50 React Exercise Questions

🟢 Beginner Level (1–15)

1. Create a Hello World component.
2. Build a component that accepts props and displays them.
3. Create a reusable Button component.
4. Toggle visibility of text using a button (show/hide).
5. Build a simple Counter with + and - buttons.
6. Create a Welcome component with dynamic name (via props).
7. Make a simple profile card using JSX and props.
8. Build a component that displays current date/time.
9. Create a component that changes background color on click.
10. Display a list of items using .map().
11. Build a simple Todo list (add/delete tasks).
12. Render different components based on login status.
13. Show a message only if a checkbox is checked.
14. Create a dropdown and display selected value.
15. Build a component that accepts children props.

🟡 Intermediate Level (16–35)

1. Build a form with inputs (name, email) and show submitted data.
2. Create a live updating clock using useEffect.
3. Create a counter that resets after reaching 10.
4. Fetch and display data from an API using fetch().
5. Build a component with multiple useState hooks.
6. Create an input field and show real-time text preview.
7. Create a dark mode toggle using useState and className.
8. Show loading message while fetching data.
9. Create a timer that counts seconds.
10. Use useRef to focus an input on button click.
11. Use useRef to store previous value of a state.
12. Create a theme switcher with useContext.
13. Make a login form and show “Welcome, [name]” on submit.
14. Create a counter with useReducer and actions: INCREMENT, DECREMENT, RESET.
15. Build a Tabs component (switch content by tab).
16. Build a custom hook: useCounter().
17. Create a modal popup using conditional rendering.
18. Fetch and show random user data on button click.
19. Show/hide a component with a fade animation.
20. Create a simple React Router setup with Home, About, Contact pages.

🔴 Advanced Level (36–50)

1. Build a Todo App with useReducer.
2. Create a product list with Add to Cart functionality.
3. Build a search input with live filter results.
4. Create a form with validation (name, email, password).
5. Create a protected route that redirects if not logged in.
6. Create a Pagination component.
7. Use useMemo to memoize a heavy calculation (e.g., Fibonacci).
8. Use useCallback to prevent child re-renders on parent update.
9. Create a custom hook useLocalStorage() for saving data.
10. Create a multi-step form wizard.
11. Integrate Axios and fetch posts from JSONPlaceholder.
12. Lazy-load components using React.lazy() and Suspense.
13. Add loading spinner while a page loads.
14. Host a React app on Netlify (include .env file).
15. Build a weather app using OpenWeatherMap API and display weather by city name.

Thank

You

🙏🏻