

React Notes

React Fundamentals (Basics)

1. What is React?

React is an **open-source JavaScript library** developed by **Facebook (Meta)** for **building fast, interactive, and reusable user interfaces (UIs)** — especially for **single-page applications (SPAs)**.

It allows developers to create large web applications that can update and render efficiently without reloading the entire page.

☐ Key Characteristics of React

Feature	Description
Library (not a framework)	React focuses only on the view layer (UI). You can combine it with other libraries for routing or state management.
Declarative	You describe <i>what the UI should look like</i> , and React efficiently updates and renders components when data changes.
Component-Based Architecture	UI is divided into small, reusable components — each managing its own state and logic.
Virtual DOM	React uses a lightweight copy of the actual DOM to improve performance by updating only the parts that change.
Unidirectional Data Flow	Data flows from parent to child components via props, making debugging and understanding easier.
JSX (JavaScript XML)	A syntax extension that lets you write HTML-like code inside JavaScript — easier to visualize UI structure.

☐ Basic Example:

```
// App.js
import React from 'react';

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

export default App;
```

Explanation:

- `import React` → allows using JSX.
- `function App()` → defines a functional component.
- `return <h1>...</h1>` → returns a UI element.
- `export default App` → exports the component so it can be used elsewhere (like `index.js`).

❑ Why Use React?

Reason	Explanation
Fast Rendering	Virtual DOM minimizes actual DOM updates → better performance.
Reusable Components	Components can be reused across pages → saves time and reduces code duplication.
Easy to Learn	Simple concepts — components, props, state, and hooks — make it beginner-friendly.
Strong Community Support	Large ecosystem with libraries, tutorials, and job opportunities.
SEO-Friendly	Can be rendered on the server (using Next.js) for better SEO performance.
Backed by Meta (Facebook)	Actively maintained and production-tested by top tech companies.

❑ How React Works (Simplified Flow)

1. Developer writes components using **JSX**.
2. React converts JSX → JavaScript → **Virtual DOM**.
3. Virtual DOM compares (diffs) with previous version.
4. Only changed elements are updated in the **Real DOM**.

This process is called **Reconciliation**.

Setting Up React Environment (Vite / CRA)

❑ Prerequisites

- Install **Node.js** & **npm** → <https://nodejs.org>
 - Check versions:
 - `node -v`
 - `npm -v`
-

1. Create React App (CRA)

Steps:

```
npx create-react-app my-app
cd my-app
npm start
```

- Runs on <http://localhost:3000>
 - Preconfigured with Webpack & Babel
 - Folder: src/, public/, package.json
-

❑ 2. Using Vite (Faster Method)

Steps:

```
npm create vite@latest my-vite-app
cd my-vite-app
npm install
npm run dev
```

- Runs on <http://localhost:5173>
 - Uses **ESBuild + Rollup** (super fast)
 - Folder: src/, index.html, vite.config.js
-

❑ CRA vs Vite

Feature	CRA	Vite
Speed	Slow	❑ Fast
Bundler	Webpack	ESBuild
Hot Reload	Slower	Instant
Config	Hidden	Customizable

Folder Structure Overview

❑ Typical React App Structure (Vite Example)

```
my-app/
├── node_modules/    # Installed dependencies
├── public/          # Static assets (images, icons)
└── src/             # Source code (main work area)
```

— App.jsx	# Root component
— main.jsx	# Entry point
— App.css	# Component styling
— index.css	# Global styles
— package.json	# Project info + dependencies
— vite.config.js	# Build configuration
— index.html	# Main HTML template

□ Key Files

- **index.html** → Main container (<div id="root">)
- **main.jsx** → Connects React to DOM
- **App.jsx** → Root component (main UI)
- **package.json** → Scripts & dependencies

□ 4. Understanding JSX (JavaScript XML)

□ What is JSX?

JSX = **JavaScript + XML-like syntax** used in React to describe UI.
It allows writing **HTML inside JavaScript**.

Example:

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

□ Why JSX?

- Makes code **clean & readable**
- Combines **markup + logic**
- Converts to `React.createElement()` internally

□ JSX Rules

- Must have **one parent element**
- Use **className** instead of `class`
- JavaScript expressions → `{ }`
- Self-close tags → ``, `<input />`

Example:

```
function Welcome() {  
  const name = "Prathamesh";  
  return <h2>Hello, {name} □</h2>;  
}
```

```
}
```

□ 5. Functional Components

□ What are Functional Components?

A **functional component** is a **JavaScript function** that returns JSX (UI). It's the simplest and most commonly used component type in modern React.

Example:

```
function Greeting() {  
  return <h1>Welcome to React!</h1>;  
}  
export default Greeting;
```

□ Key Points

- Must **return JSX**
- Component names → **start with Capital Letter**
- Can receive **props (inputs)**

Example with Props:

```
function Welcome(props) {  
  return <h2>Hello, {props.name}</h2>;  
}
```

Use:

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

□ Advantages

- Simple & reusable
- Easy to test
- Supports **Hooks** (like `useState`, `useEffect`)

Class Components (Legacy but Useful)

□ Definition

Class Components are **ES6 classes** that extend `React.Component`. They were used before Hooks for state and lifecycle management.

□ Syntax

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

class Welcome extends Component {
  render() {
    return <h2>Hello, {this.props.name}</h2>;
  }
}

export default Welcome;
```

□ Key Points

- Must extend React.Component
- Use render() method to return JSX
- Access props → this.props
- Manage state → this.state

□ Example with State

```
class Counter extends Component {
  state = { count: 0 };
  render() {
    return <p>Count: {this.state.count}</p>;
  }
}
```

□ Modern React prefers **Functional Components + Hooks**, but class components are still found in legacy projects.

7. Rendering Elements

□ Definition

Rendering means **displaying React elements** (JSX) inside the DOM.

□ Basic Example

```
const element = <h1>Hello React!</h1>;
ReactDOM.createRoot(document.getElementById('root')).render(element);
```

□ Updating Elements

React automatically re-renders only the **changed parts** using the **Virtual DOM**.

Example:

```
function tick() {  
  const time = <h2>{new Date().toLocaleTimeString()}</h2>;  
  ReactDOM.createRoot(document.getElementById('root')).render(time);  
}  
setInterval(tick, 1000);
```

8. Props (Passing Data Between Components)

□ Definition

Props = short for **Properties** → used to **pass data** from a **parent** component to a **child** component.

They are **read-only** and **immutable** inside the child.

□ Example

```
function Welcome(props) {  
  return <h2>Hello, {props.name}</h2>;  
}
```

```
function App() {  
  return (  
    <div>  
      <Welcome name="Prathamesh" />  
      <Welcome name="Jadhav" />  
    </div>  
  );  
}
```

□ Key Points

- Passed like HTML attributes
- Accessible via props object
- Props are **immutable** (cannot be changed)

□ Using Destructuring

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

9. Component Composition

□ Definition

Composition means combining multiple small components to build a **complex UI**. It promotes **reusability** and **clean structure**.

□ Example

```
function Header() {  
  return <h1>React App</h1>;  
}  
  
function Footer() {  
  return <p>© 2025 Jadhav</p>;  
}  
  
function App() {  
  return (  
    <div>  
      <Header />  
      <p>Welcome to my app!</p>  
      <Footer />  
    </div>  
  );  
}
```

□ Benefits

- Easier to maintain
- Reusable, modular code
- Improves readability and scalability

Using key in Lists

□ Definition

Keys help **React identify which items changed, added, or removed** when rendering lists. They make list rendering **efficient**.

□ Example

```
const items = ['A', 'B', 'C'];  
const list = items.map((item, index) => <li key={index}>{item}</li>);
```

□ Rules

- Keys must be **unique** among siblings.
- Prefer **stable IDs** over array indexes (for performance).
- Keys help React track elements when updating the Virtual DOM.

□ **Good Practice:**

```
<li key={user.id}>{user.name}</li>
```

11. Conditional Rendering (if, &&, ?:)

□ **Definition**

React allows rendering UI conditionally — based on state or props.

□ **Using if**

```
if (isLoggedIn) {  
  return <h2>Welcome Back!</h2>;  
} else {  
  return <h2>Please Log In</h2>;  
}
```

□ **Using && (Short-Circuit)**

```
{isLoggedIn && <p>You are logged in □</p>}
```

□ **Using Ternary ?:**

```
<p>{isLoggedIn ? 'Welcome!' : 'Please Log In'}</p>
```

□ **Use ternary operators inside JSX** for concise code.

12. Inline and External CSS Styling

□ **1. Inline Styling**

- Define styles as a **JavaScript object**
- Property names in **camelCase**

```
function App() {  
  const style = { color: 'blue', backgroundColor: 'lightgray' };  
  return <h2 style={style}>Hello Inline Style</h2>;  
}
```

or directly:

```
<h2 style={{ color: 'red' }}>Hello</h2>
```

2. External CSS File

- Create App.css

```
.title {  
  color: purple;  
  text-align: center;  
}
```

- Import & apply:

```
import './App.css';
```

```
function App() {  
  return <h2 className="title">Hello CSS</h2>;  
}
```

❑ **Tip:** Prefer **external CSS** or **CSS modules** for scalability.

13. Handling Events (onClick, onChange, etc.)

❑ Definition

React events are handled using **camelCase event names** (e.g., onClick, onChange).

❑ Example: onClick

```
function Button() {  
  function handleClick() {  
    alert('Button Clicked!');  
  }  
  return <button onClick={handleClick}>Click Me</button>;  
}
```

❑ Example: onChange

```
function InputBox() {  
  const handleChange = (e) => console.log(e.target.value);  
  return <input type="text" onChange={handleChange} />;  
}
```

}

□ Common React Events

Event	Trigger
onClick	Button or element click
onChange	Input change
onSubmit	Form submission
onMouseOver	Hover
onKeyDown	Keyboard press

- **Best Practice:** Use event handlers as **functions**, not inline strings.

React State Management (Core Concepts)

14. What is State

□ Definition

- **State** is a **JavaScript object** that holds **data or UI information** specific to a component.
- Changes in state trigger **re-rendering** of the component.

□ Key Points

- Local to the component
- Mutable (can change over time)
- Triggers UI updates automatically

□ Example

```
function Counter() {  
  let count = 0; // Not state  
  return <p>Count: {count}</p>;  
}
```

- Here count does not cause re-render when updated.
- Need **state** to manage reactive updates (use useState hook).

15. useState() Hook

□ Definition

useState() is a **React Hook** that lets functional components **have state**.

□ Syntax

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

- state → current value
- setState → function to update value
- initialValue → starting value

□ 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>
  );
}
```

16. Updating and Managing State

□ Rules

1. **Never modify state directly** (e.g., count = count + 1)
2. Use **setter function** (setState)
3. Updates can be **asynchronous** → always prefer **functional update** for dependent values

□ Functional Update Example

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

□ Complex State

- Can use objects or arrays as state

```
const [user, setUser] = useState({ name: '', age: 0 });
```

```
setUser(prev => ({ ...prev, name: 'Prathamesh' }));
```

17. Lifting State Up

□ Definition

- Moving **state to a common parent** so multiple child components can **share and update data**.

□ Example

```
function Parent() {  
  const [name, setName] = useState("");  
  
  return (  
    <>  
      <Child1 setName={setName} />  
      <Child2 name={name} />  
    </>  
  );  
}  
  
function Child1({ setName }) {  
  return <input onChange={e => setName(e.target.value)} />;  
}  
  
function Child2({ name }) {  
  return <p>Hello, {name}</p>;  
}
```

- **Benefit:** Child components can share data without duplicating state.
-

18. Props Drilling Problem

□ Definition

- Passing props through **multiple intermediate components** to reach a deep child.
- Leads to **tedious and hard-to-maintain code**.

□ Example

```
<GrandParent>  
  <Parent>
```

```
<Child data={someData} />
</Parent>
</GrandParent>
```

- someData may need to pass through Parent unnecessarily.

❑ **Solution:** Use **Context API** or **state management libraries** (Redux, Zustand, etc.)

19. Controlled vs Uncontrolled Components

Feature	Controlled Component	Uncontrolled Component
Value	Managed by React state	Managed by DOM
Data Flow	One-way (state → UI)	Direct DOM access
Access	Use value + onChange	Use ref to access DOM value
Example	<input value={name} onChange={handleChange} />	<input defaultValue="John" />

❑ Controlled components are preferred for **form handling** in React.

20. Two-Way Data Binding

❑ **Definition**

- **Two-way binding** allows **UI to update state** and **state to update UI** automatically.
- React implements this using **state + onChange events**.

❑ **Example**

```
function TextInput() {
  const [text, setText] = useState("");

  return (
    <input
      type="text"
      value={text}      // State → UI
      onChange={e => setText(e.target.value)} // UI → State
    />
  )
}
```

```
);  
}
```

- Commonly used in **forms** and input fields.

React Hooks (Essential Hooks)

21. Introduction to Hooks

□ Definition

- **Hooks** are special **functions in React** that let you use **state and lifecycle features** inside **functional components**.
- Introduced in **React 16.8** to eliminate the need for class components.

□ Key Benefits

- Use **state** in functional components (useState)
- Handle **side effects** (useEffect)
- Reuse **logic** without HOCs or render props

□ Example

```
import React, { useState } from 'react';  
  
function Counter() {  
  const [count, setCount] = useState(0);  
  return <button onClick={() => setCount(count + 1)}>Count: {count}</button>;  
}
```

22. Rules of Hooks

1. **Call Hooks only at the top level** → Do not call inside loops, conditions, or nested functions.
2. **Call Hooks only from React functions** → Functional components or custom hooks.
3. **Custom hooks** must start with **use** prefix.

- Following these rules ensures predictable behavior.
-

23. useEffect() – Side Effects & Cleanup

□ Definition

- useEffect lets you **perform side effects** in functional components:
 - Fetch data
 - Set timers
 - Update DOM manually

□ Syntax

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

□ Example: Timer

```
import { useState, useEffect } from 'react';  
  
function Timer() {  
  const [count, setCount] = useState(0);  
  
  useEffect(() => {  
    const interval = setInterval(() => setCount(c => c + 1), 1000);  
    return () => clearInterval(interval); // cleanup  
  }, []);  
  
  return <p>Seconds: {count}</p>;  
}
```

- Cleanup prevents **memory leaks**.

24. Dependency Array in useEffect()

Dependency Array	Behavior
[]	Run once after initial render
[dep1, dep2]	Run whenever dependencies change
omitted	Run after every render

- ❑ Best Practice: Always include dependencies to avoid stale state bugs.
-

25. useRef() – Accessing DOM & Persisting Values

❑ Definition

- useRef stores a **mutable value** that **does not trigger re-renders**.
- Often used to **access DOM elements**.

❑ Syntax

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

❑ Example: DOM Access

```
function InputFocus() {  
  const inputRef = useRef(null);  
  
  const focusInput = () => inputRef.current.focus();  
  
  return <input ref={inputRef} type="text" />;  
}
```

Notes

- Can also store any **persistent value** across renders (like timers).
-

26. useMemo() – Memoization

❑ Definition

- useMemo **caches the result of a function** to avoid expensive recalculations on re-render.

❑ Syntax

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

❑ Example

```
const expensive = useMemo(() => factorial(count), [count]);
```

- ❑ Use when **computation is heavy** and dependent values rarely change.

27. useCallback() – Function Optimization

□ Definition

- useCallback **memoizes a function** so it is **not recreated** on every render.
- Useful when passing functions to child components.

□ Syntax

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

□ Example

```
const handleClick = useCallback(() => setCount(c => c + 1), []);
```

- Prevents **unnecessary re-renders** of children.
-

28. useContext() – Global State Sharing

□ Definition

- useContext allows a component to **consume context values** without prop drilling.

□ Example

```
import { createContext, useContext } from 'react';
```

```
const ThemeContext = createContext('light');
```

```
function Child() {  
  const theme = useContext(ThemeContext);  
  return <p>Theme: {theme}</p>;  
}
```

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

- Solves **props drilling** problem for global/shared state.
-

29. useReducer() – Alternative to useState

□ Definition

- useReducer is a **state management hook** for complex state logic.
- Similar to **Redux**, but built-in.

□ Syntax

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

□ Example

```
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, { count: 0 });
```

- Useful when state depends on **previous state or multiple actions**.
-

30. useEffect()

□ Definition

- Similar to useEffect, but **fires synchronously after DOM mutations**.
- Use for **reading layout and synchronously re-rendering**.

□ Note

- Can block paint → use only when necessary.

```
useLayoutEffect(() => {  
  // DOM measurements here  
}, []);
```

31. Custom Hooks (Creating and Reusing Logic)

□ Definition

- Custom hooks are **functions that use built-in hooks to reuse logic** across components.

□ Syntax

```
function useCounter(initial) {  
  const [count, setCount] = useState(initial);  
  const increment = () => setCount(c => c + 1);  
  return { count, increment };  
}
```

□ Example

```
function Counter() {  
  const { count, increment } = useCounter(0);  
  return <button onClick={increment}>Count: {count}</button>;  
}
```

- Best Practice: **Start custom hook name with use.**

React Routing and Navigation

32. React Router Introduction

□ Definition

- **React Router** is a standard library for **routing in React**.
- It allows building **single-page applications (SPA)** with **dynamic URL navigation** without full page reloads.

□ Key Features

- Declarative routing
 - Nested routes
 - Route parameters
 - Programmatic navigation
 - Protected routes (authentication-based)
-

33. Installing and Setting Up React Router

□ Installation

npm install react-router-dom

□ Basic Setup

```
import { BrowserRouter } from 'react-router-dom';  
import App from './App';
```

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

- Wrap your app in <BrowserRouter> for routing functionality.
-

34. <BrowserRouter>, <Routes>, <Route>

□ Usage

```
import { Routes, Route } from 'react-router-dom';  
import Home from './Home';  
import About from './About';
```

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

□ Key Points

- <BrowserRouter> → Wraps the app, enables routing
 - <Routes> → Container for all <Route>
 - <Route> → Defines a path & component (element)
-

35. Link and NavLink Components

□ Definition

- `<Link>` and `<NavLink>` replace `<a>` tags for **client-side navigation**.

□ Example

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

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

□ Difference:

- `<Link>` → Simple navigation
 - `<NavLink>` → Adds **active class** when route is active
-

36. Route Parameters (useParams)

□ Definition

- Route parameters allow passing **dynamic values in the URL**.

□ Example

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

```
function User() {  
  const { id } = useParams(); // Get URL param  
  return <h2>User ID: {id}</h2>;  
}
```

```
// Route
```

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

37. Navigate Programmatically (useNavigate)

□ Definition

- `useNavigate` allows **programmatic navigation** (redirects).

□ Example

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

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

  const handleLogin = () => {
    // authentication logic
    navigate('/dashboard'); // redirect
  };

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

38. Nested Routes

□ Definition

- Nest routes to create **hierarchical UI structure**.

□ Example

```
function Dashboard() {
  return (
    <div>
      <h1>Dashboard</h1>
      <Routes>
        <Route path="profile" element={ <Profile /> } />
        <Route path="settings" element={ <Settings /> } />
      </Routes>
    </div>
  );
}
```

// Access: /dashboard/profile, /dashboard/settings

- Use <Outlet /> in parent component to render child routes.
-

39. Protected Routes (Authentication-based Routing)

□ Definition

- Routes that require **user authentication**.
- Prevents unauthorized access.

□ Example

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

function ProtectedRoute({ isAuth, children }) {
  return isAuth ? children : <Navigate to="/login" />;
}

// Usage
<Route path="/dashboard" element={
  <ProtectedRoute isAuth={loggedIn}>
    <Dashboard />
  </ProtectedRoute>
} />
```

- Ensures only authenticated users can access sensitive pages.

React Forms and User Input

40. Handling Forms in React

□ Definition

- React **forms** handle **user input** and **capture data** in state.
- Forms can be **controlled** or **uncontrolled**.

□ Basic Form Example

```
function SimpleForm() {
  const handleSubmit = (e) => {
    e.preventDefault(); // Prevent page reload
    alert('Form Submitted!');
  };

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

- Use `e.preventDefault()` to **stop default browser behavior**.
-

41. Controlled Form Inputs

□ Definition

- Controlled inputs are **connected to React state**.
- Value is updated via **state + onChange** → enables **two-way binding**.

□ Example

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

```
function ControlledForm() {  
  const [name, setName] = useState("");  
  
  return (  
    <form>  
      <input  
        type="text"  
        value={name}           // state → UI  
        onChange={(e) => setName(e.target.value)} // UI → state  
      />  
      <p>Hello, {name}</p>  
    </form>  
  );  
}
```

- Best practice for **form validation** and **dynamic updates**.
-

42. Handling Multiple Inputs

□ Definition

- Use **single state object** to manage **multiple inputs** efficiently.

□ Example

```
function MultiInputForm() {  
  const [formData, setFormData] = useState({ name: "", email: "" });  
  
  const handleChange = (e) => {
```

```
const { name, value } = e.target;
setFormData(prev => ({ ...prev, [name]: value }));
};

return (
  <form>
    <input name="name" value={formData.name} onChange={handleChange} />
    <input name="email" value={formData.email} onChange={handleChange} />
    <p>Name: {formData.name}, Email: {formData.email}</p>
  </form>
);
}
```

- ❑ Use **computed property names** [name] to update dynamic fields.
-

43. Form Validation

❑ Definition

- Validates user input **before submission**.
- Can be **client-side** or **server-side**.

❑ Example: Simple Validation

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

  const handleSubmit = (e) => {
    e.preventDefault();
    if (!email.includes('@')) {
      setError('Invalid email!');
    } else {
      setError("");
      alert('Form Submitted!');
    }
  };

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

```
    />
    {error && <p style={{ color:'red'}}>{error}</p>}
    <button type="submit">Submit</button>
  </form>
);
}
```

□ Validation can include:

- Required fields
- Email format
- Password strength
- Length checks

44. Using Libraries like Formik or React Hook Form

□ Why Use Libraries

- Simplify **form handling**, **validation**, and **submission**.
- Reduce **boilerplate code**.

□ Formik Example

```
import { Formik, Form, Field } from 'formik';
```

```
function MyForm() {
  return (
    <Formik
      initialValues={{ name: '', email: '' }}
      onSubmit={values => alert(JSON.stringify(values))}
    >
      <Form>
        <Field name="name" placeholder="Name" />
        <Field name="email" placeholder="Email" type="email" />
        <button type="submit">Submit</button>
      </Form>
    </Formik>
  );
}
```

□ React Hook Form Example

```
import { useForm } from 'react-hook-form';
```

```
function MyForm() {
  const { register, handleSubmit } = useForm();
  const onSubmit = data => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("name")} placeholder="Name" />
      <input {...register("email")} placeholder="Email" />
      <button type="submit">Submit</button>
    </form>
  );
}
```

□ Both libraries support:

- Built-in validation
- Easy integration with Yup for **schema-based validation**
- Better **performance and scalability**

React Component Communication

45. Parent to Child Communication (Props)

□ **Definition**

- Parent components can **pass data to child components** using **props**.
- Props are **read-only** in the child.

□ **Example**

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

function Parent() {
  return <Child name="Prathamesh" />;
}
```

□ Props allow **data flow in one direction** (Parent → Child).

46. Child to Parent Communication (Callbacks)

□ Definition

- Child components **cannot directly modify parent state**.
- Use **callback functions passed as props** to send data to parent.

□ Example

```
function Child({ sendData }) {  
  return <button onClick={() => sendData('Hello from Child')}>Send</button>;  
}
```

```
function Parent() {  
  const handleData = (data) => alert(data);  
  return <Child sendData={handleData} />;  
}
```

- Common pattern for **event handling and state updates** from child to parent.
-

47. Sibling Component Communication

□ Definition

- Siblings cannot communicate directly.
- Use **common parent** to share data between siblings.

□ Example

```
function Sibling1({ setMessage }) {  
  return <button onClick={() => setMessage('Hello from Sibling1')}>Send</button>;  
}
```

```
function Sibling2({ message }) {  
  return <p>Message: {message}</p>;  
}
```

```
function Parent() {  
  const [message, setMessage] = useState("");  
  return (  
    <>  
      <Sibling1 setMessage={setMessage} />  
      <Sibling2 message={message} />  
    </>  
  );  
}
```

- Sibling communication flows via **Parent state**.
-

48. Context API for Global State

□ Definition

- Context API allows sharing **global data** across components **without props drilling**.

□ Example

```
import { createContext, useContext } from 'react';
```

```
const ThemeContext = createContext('light');
```

```
function Child() {  
  const theme = useContext(ThemeContext);  
  return <p>Current Theme: {theme}</p>;  
}
```

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

- Context is great for **theme, auth, or language** data shared across app.
-

49. Prop Drilling vs Context API

Concept	Description	Example
Prop Drilling	Passing props through multiple intermediate components	GrandParent → Parent → Child
Context API	Provides global state accessible by any nested component	Use createContext + useContext

- Context solves **prop drilling problem**, making code **cleaner and scalable**.

React Component Lifecycle

50. Mounting, Updating, Unmounting (Class Components)

□ Lifecycle Phases

1. **Mounting** – Component is **created and inserted into DOM**
 - Methods:
 - constructor() → initialize state
 - static getDerivedStateFromProps() → sync state with props
 - render() → returns JSX
 - componentDidMount() → called after component is mounted
2. **Updating** – Component **re-renders** due to state or props changes
 - Methods:
 - static getDerivedStateFromProps() → sync state with props
 - shouldComponentUpdate() → control re-render (return true/false)
 - render() → re-render JSX
 - getSnapshotBeforeUpdate() → capture snapshot before DOM update
 - componentDidUpdate() → called after DOM updates
3. **Unmounting** – Component **removed from DOM**
 - Method: componentWillUnmount() → cleanup tasks (timers, subscriptions)

□ Example

```
class Demo extends React.Component {  
  componentDidMount() {  
    console.log('Mounted');  
  }  
  componentDidUpdate() {  
    console.log('Updated');  
  }  
  componentWillUnmount() {  
    console.log('Unmounted');  
  }  
  render() {  
    return <h2>Lifecycle Demo</h2>;  
  }  
}
```

51. Mapping Lifecycle to Hooks (Functional Components)

Class Component Method	Functional Equivalent (Hooks)	Notes
componentDidMount	useEffect(() => {}, [])	Runs once after initial render
componentDidUpdate	useEffect(() => {}, [dependencies])	Runs when dependencies change
componentWillUnmount	useEffect(() => { return () => {} }, [])	Cleanup function inside useEffect
render	JSX return	Always re-render on state/props change

❑ Example

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

function Demo() {
  useEffect(() => {
    console.log('Mounted');

    return () => console.log('Unmounted'); // Cleanup
  }, []);

  return <h2>Lifecycle with Hooks</h2>;
}
```

❑ Hooks **combine multiple lifecycle behaviors** in one function.

52. Cleanup Functions in useEffect

❑ Definition

- **Cleanup function** runs before the component **unmounts** or **before next effect execution**.
- Prevents **memory leaks** (timers, subscriptions, event listeners).

❑ Example: Timer Cleanup

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

```
function Timer() {  
  const [count, setCount] = useState(0);  
  
  useEffect(() => {  
    const interval = setInterval(() => setCount(c => c + 1), 1000);  
  
    return () => clearInterval(interval); // Cleanup  
  }, []);  
  
  return <p>Seconds: {count}</p>;  
}
```

□ Key Notes

- Runs **on unmount**
- Runs **before next effect** if dependencies change
- Essential for subscriptions, intervals, API aborts

React API Calls and Data Fetching

53. Fetching Data using Fetch API / Axios

□ Fetch API

```
useEffect(() => {  
  fetch('https://jsonplaceholder.typicode.com/posts')  
    .then(res => res.json())  
    .then(data => console.log(data))  
    .catch(err => console.error(err));  
}, []);
```

□ Axios

- Axios is a **popular library** for HTTP requests.

```
npm install axios  
import axios from 'axios';  
import { useEffect } from 'react';
```

```
useEffect(() => {  
  axios.get('https://jsonplaceholder.typicode.com/posts')  
    .then(res => console.log(res.data))  
    .catch(err => console.error(err));  
}, []);
```

- ❑ **Axios Advantages:** automatic JSON parsing, better error handling, supports interceptors.
-

54. Loading and Error States

❑ Definition

- Use state to **track loading and errors** for better UX.

❑ Example

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

function DataFetcher() {
  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState("");

  useEffect(() => {
    axios.get('https://jsonplaceholder.typicode.com/posts')
      .then(res => setData(res.data))
      .catch(err => setError('Error fetching data'))
      .finally(() => setLoading(false));
  }, []);

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

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

55. useEffect for API Calls

❑ Key Points

- Place **API calls inside useEffect** to avoid calling on every render.
- **Dependency array** controls when the call runs.

❑ Example

```
useEffect(() => {
  async function fetchData() {
    const res = await axios.get('https://jsonplaceholder.typicode.com/posts');
```

```
    setData(res.data);
  }
  fetchData();
}, []); // empty array → run once
```

- ❑ Best practice: wrap async calls in a **function inside useEffect**.
-

56. Handling Async Functions

❑ Example

```
useEffect(() => {
  const fetchData = async () => {
    try {
      const response = await axios.get('https://jsonplaceholder.typicode.com/posts');
      setData(response.data);
    } catch (err) {
      setError('Failed to fetch');
    }
  };
  fetchData();
}, []);
```

❑ Notes

- Async functions **cannot be directly passed** to useEffect
 - Use **internal async function** or **IIFE**.
-

57. Pagination and Filtering

❑ Pagination Example

```
const [page, setPage] = useState(1);
useEffect(() => {
  axios.get(`https://api.example.com/posts?page=${page}`)
    .then(res => setData(res.data));
}, [page]);
```

```
<button onClick={() => setPage(prev => prev + 1)}>Next</button>
```

❑ Filtering Example

```
const [query, setQuery] = useState("");
const filteredData = data.filter(item => item.title.includes(query));

<input value={query} onChange={e => setQuery(e.target.value)} placeholder="Search..." />
```

- ❑ Combines **dynamic user input** and **API call updates**.
-

58. Custom Hook for Data Fetching

❑ Definition

- Encapsulate **data fetching logic** into a **reusable hook**.

❑ Example

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

function useFetch(url) {
  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState("");

  useEffect(() => {
    const fetchData = async () => {
      try {
        const res = await axios.get(url);
        setData(res.data);
      } catch {
        setError('Failed to fetch');
      } finally {
        setLoading(false);
      }
    };
    fetchData();
  }, [url]);

  return { data, loading, error };
}

// Usage
function App() {
  const { data, loading, error } = useFetch('https://jsonplaceholder.typicode.com/posts');
```

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

- ❑ **Benefits:** reusable, cleaner code, separation of concerns.

React Performance Optimization & Advanced Features

59. React Rendering Process

❑ Key Points

- React uses a **declarative approach** to update the UI.
- Components **re-render** when:
 - **State changes** (useState)
 - **Props change** (parent updates data)

❑ Render Cycle

1. Triggered by state or props changes
2. Reconciliation determines **what needs to be updated**
3. Virtual DOM compares with real DOM
4. Minimal updates are applied to **real DOM**

- ❑ Goal: Reduce **unnecessary re-renders** for performance.
-

60. Reconciliation & Virtual DOM

❑ Virtual DOM

- **Lightweight JS representation of DOM**
- React compares Virtual DOM with **previous version** → calculates **diff** → updates only changed nodes in real DOM.

❑ Key Concepts

- **Diffing Algorithm:** Efficiently determines minimal changes
- **Keys in Lists:** Help React identify elements for reconciliation

- ❑ Minimizes **DOM manipulation**, improving performance.

61. useMemo and useCallback Optimization

☐ useMemo

- **Memoizes expensive computations**

```
const memoizedValue = useMemo(() => computeHeavy(data), [data]);
```

☐ useCallback

- **Memoizes function instances** to avoid unnecessary child re-renders

```
const memoizedFn = useCallback(() => doSomething(a), [a]);
```

- ☐ Use when **computation is heavy** or **functions are passed as props**.
-

62. React.memo for Component Memoization

☐ Definition

- **Higher-order component** that prevents **re-render if props do not change**.

☐ Example

```
const Child = React.memo(({ value }) => {  
  console.log('Child rendered');  
  return <p>{value}</p>;  
});
```

- ☐ Useful for **pure functional components** with **unchanging props**.
-

63. Lazy Loading (React.lazy & Suspense)

☐ Definition

- Load components **only when needed** → reduces initial bundle size.

☐ Example

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

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

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

- ☐ Suspense shows **fallback UI** while the component is loading.
-

64. Code Splitting

☐ Definition

- Split app into **smaller bundles** → load **on-demand**.

☐ Techniques

1. **Dynamic imports** with React.lazy
2. **Route-based splitting**

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

- ☐ Reduces **initial load time** → improves **performance**.
-

65. Profiling Components

☐ Definition

- Measure **render performance** to detect slow components.

☐ Steps

1. Install React DevTools
2. Open **Profiler tab**
3. Record interactions → see **render duration** and re-render reasons

- ☐ Helps **optimize performance bottlenecks**.
-

Advanced React Concepts

66. Context API Deep Dive

□ Definition

- Context API allows **global state management** without prop drilling.
- Useful for **themes, auth, language settings**.

□ Usage

```
import { createContext, useContext } from 'react';
```

```
const ThemeContext = createContext();
```

```
function Child() {  
  const theme = useContext(ThemeContext);  
  return <p>Theme: {theme}</p>;  
}
```

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

- Can be combined with useReducer for **complex global state**.
-

67. Higher Order Components (HOC)

□ Definition

- HOC is a **function that takes a component and returns a new component**.
- Used for **reusing logic** (authentication, logging, etc.)

□ Example

```
function withLogger(WrappedComponent) {  
  return function(props) {  
    console.log('Rendering:', WrappedComponent.name);  
    return <WrappedComponent {...props} />;  
  };  
}
```

```
};  
}
```

```
const EnhancedComponent = withLogger(MyComponent);
```

- HOCs **wrap components** to add functionality without modifying original component.
-

68. Render Props Pattern

□ Definition

- A component **accepts a function as a prop** to render dynamic content.

□ Example

```
function MouseTracker({ render }) {  
  const [x, setX] = useState(0);  
  const [y, setY] = useState(0);  
  
  return (  
    <div onMouseMove={e => { setX(e.clientX); setY(e.clientY); }}>  
      {render(x, y)}  
    </div>  
  );  
}
```

```
<MouseTracker render={(x, y) => <p>Mouse at {x}, {y}</p>} />
```

- Provides **flexibility** in rendering without breaking component structure.
-

69. Portals (Rendering Outside Root DOM)

□ Definition

- Portals allow rendering a component **outside the root DOM hierarchy**.
- Useful for **modals, tooltips, overlays**.

□ Example

```
import { createPortal } from 'react-dom';
```

```
function Modal({ children }) {
```

```
return createPortal(  
  <div className="modal">{children}</div>,  
  document.getElementById('modal-root')  
)  
}
```

- ❑ Keeps modal **outside main DOM** but maintains React state and context.
-

70. Error Boundaries

❑ Definition

- Error boundaries **catch JavaScript errors** in child components.
- Prevents entire app from crashing.

❑ Example

```
class ErrorBoundary extends React.Component {  
  state = { hasError: false };  
  
  static getDerivedStateFromError() {  
    return { hasError: true };  
  }  
  
  componentDidCatch(error, info) {  
    console.log(error, info);  
  }  
  
  render() {  
    return this.state.hasError ? <h2>Something went wrong.</h2> : this.props.children;  
  }  
}  
  
<ErrorBoundary>  
  <MyComponent />  
</ErrorBoundary>
```

- ❑ Only works with **class components**.
-

71. Refs and Forwarding Refs

❑ Refs

- useRef allows **direct DOM access** or storing mutable values.

□ Forwarding Refs

```
const Input = React.forwardRef((props, ref) => <input ref={ref} {...props} />);
```

```
function App() {  
  const inputRef = useRef();  
  return <Input ref={inputRef} />;  
}
```

- Forward refs allow **parent components to control child DOM nodes**.
-

72. Compound Components Pattern

□ Definition

- Components work **together as a group**, sharing **implicit state**.
- Example: Tabs, Accordion

□ Example

```
function Tabs({ children }) {  
  const [activeIndex, setActiveIndex] = useState(0);  
  return React.Children.map(children, (child, index) =>  
    React.cloneElement(child, { active: index === activeIndex, setActiveIndex, index })  
  );  
}
```

- Enables **flexible and reusable UI patterns**.
-

73. Controlled vs Uncontrolled Components (Advanced Use)

Type	Definition	Use Case
Controlled	Value managed by React state	Form validation, dynamic forms
Uncontrolled	Value managed by DOM	Quick forms, integrating 3rd-party libraries

□ Example Uncontrolled

```
const inputRef = useRef();
<form onSubmit={() => console.log(inputRef.current.value)}>
  <input ref={inputRef} />
</form>
```

74. Debouncing & Throttling in React

□ Definition

- **Debounce** → Delays execution until **user stops typing**
- **Throttle** → Limits execution to **once per interval**

□ Example: Debounce Input

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

function Search({ onSearch }) {
  const [query, setQuery] = useState("");

  useEffect(() => {
    const handler = setTimeout(() => onSearch(query), 500);
    return () => clearTimeout(handler); // cleanup
  }, [query]);

  return <input value={query} onChange={e => setQuery(e.target.value)} />;
}
```

- Improves **performance in search inputs or scroll events**.

React State Management Libraries

75. Redux (Basics, Actions, Reducers, Store)

□ Definition

- **Redux** is a predictable **state container** for React apps.
- Manages **global state** outside components.

□ Core Concepts

1. **Store** – holds the state
2. **Actions** – describe **what happened** { type, payload }
3. **Reducers** – define **how state changes** based on action

4. **Dispatch** – sends action to reducer

□ **Example**

```
// Action
const increment = () => ({ type: 'INCREMENT' });

// Reducer
const counterReducer = (state = 0, action) => {
  switch(action.type) {
    case 'INCREMENT': return state + 1;
    default: return state;
  }
};

// Store
import { createStore } from 'redux';
const store = createStore(counterReducer);

// Dispatch
store.dispatch(increment());
console.log(store.getState()); // 1
```

- **Pros:** Predictable state, easy debugging
 - **Cons:** Boilerplate code
-

76. Redux Toolkit (Simplified Redux)

□ **Definition**

- Official **Redux wrapper** to **reduce boilerplate**.
- Includes createSlice, configureStore, createAsyncThunk.

□ **Example**

```
import { configureStore, createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter',
  initialState: 0,
  reducers: {
    increment: state => state + 1,
    decrement: state => state - 1
  }
});
```

```
});  
  
export const { increment, decrement } = counterSlice.actions;  
  
const store = configureStore({ reducer: counterSlice.reducer });  
store.dispatch(increment());  
console.log(store.getState()); // 1
```

- ❑ Reduces **setup time**, automatically integrates **DevTools**.
-

77. Zustand (Lightweight State Management)

❑ Definition

- **Minimalistic state management** library for React.
- Simple API, **no boilerplate**, great for small-medium apps.

❑ Example

```
import create from 'zustand';  
  
const useStore = create(set => ({  
  count: 0,  
  increment: () => set(state => ({ count: state.count + 1 })))  
});  
  
function Counter() {  
  const { count, increment } = useStore();  
  return <button onClick={increment}>{count}</button>;  
}
```

- ❑ Easy to **read, write, and integrate** without context or reducers.
-

78. Recoil / Jotai Overview

❑ Recoil

- **State management library by Facebook**
- Provides **atoms (state units)** and **selectors (derived state)**

❑ Example

```
import { atom, useRecoilState } from 'recoil';

const countState = atom({ key: 'count', default: 0 });

function Counter() {
  const [count, setCount] = useRecoilState(countState);
  return <button onClick={() => setCount(count + 1)}>{count}</button>;
}
```

□ Jotai

- Minimalistic, **atomic state library** for React
- Similar to Recoil, simpler API

□ Both are **modern alternatives** to Redux for smaller apps or component-level state.

79. React Query / TanStack Query for Server State

□ Definition

- **React Query** manages **server state**, caching, background fetching, and synchronization.

□ Example

```
import { useQuery } from '@tanstack/react-query';
import axios from 'axios';

function App() {
  const { data, isLoading, error } = useQuery(['posts'], () =>
    axios.get('https://jsonplaceholder.typicode.com/posts').then(res => res.data)
  );

  if (isLoading) return <p>Loading...</p>;
  if (error) return <p>Error fetching data</p>;

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

□ Features:

- Caching
- Automatic re-fetching
- Pagination support
- Background updates

React Testing

80. Introduction to Testing

□ Definition

- Testing ensures **components, logic, and apps work correctly**.
- Types of testing in React:
 1. **Unit Testing** – test individual functions/components
 2. **Integration Testing** – test interaction between components
 3. **End-to-End (E2E) Testing** – test entire application flow

□ Benefits

- Catch bugs early
 - Prevent regressions
 - Improve code quality
-

81. Jest Basics

□ Definition

- **Jest** is a popular testing framework for **JavaScript & React**.
- Provides **test runner, assertions, mocking, and snapshot testing**.

□ Example

```
npm install --save-dev jest
// sum.js
export const sum = (a, b) => a + b;
```

```
// sum.test.js
import { sum } from './sum';
```

```
test('adds 2 + 3 to equal 5', () => {
  expect(sum(2, 3)).toBe(5);
});
```

- `expect(value).toBe(expected)` → basic assertion
 - Run tests: `npx jest`
-

82. React Testing Library (RTL)

□ Definition

- Focuses on **testing components from user perspective**.
- Encourages **testing UI interactions** rather than implementation details.

□ Installation

```
npm install --save-dev @testing-library/react @testing-library/jest-dom
```

□ Example

```
import { render, screen, fireEvent } from '@testing-library/react';
import Button from './Button';
```

```
test('renders button and handles click', () => {
  const handleClick = jest.fn();
  render(<Button onClick={handleClick}>Click Me</Button>);
```

```
  const button = screen.getByText(/Click Me/i);
  fireEvent.click(button);
```

```
  expect(handleClick).toHaveBeenCalledTimes(1);
});
```

- RTL encourages **querying elements like users** (getByText, getByRole)
 - Works well with Jest for assertions.
-

83. Snapshot Testing

□ Definition

- Capture **rendered UI structure** and save a snapshot.
- Detects **unexpected UI changes** in future renders.

□ Example

```
import { render } from '@testing-library/react';
import Button from './Button';
```

```
test('matches snapshot', () => {
  const { asFragment } = render(<Button>Click Me</Button>);
  expect(asFragment()).toMatchSnapshot();
});
```

```
});
```

- ❑ Useful for **component regression testing**
 - ❑ Snapshots stored in `__snapshots__` folder
-

84. Testing Hooks & Components

❑ Testing Hooks

- Use `@testing-library/react-hooks` or render hooks inside test components.

❑ Example: `useCounter` Hook

```
import { renderHook, act } from '@testing-library/react-hooks';
import { useState } from 'react';
```

```
function useCounter(initial = 0) {
  const [count, setCount] = useState(initial);
  const increment = () => setCount(c => c + 1);
  return { count, increment };
}
```

```
test('useCounter increments count', () => {
  const { result } = renderHook(() => useCounter(0));
```

```
  act(() => result.current.increment());
  expect(result.current.count).toBe(1);
});
```

❑ Tips for Component Testing

- Test **UI rendering** and **user interactions**
- Use **mock functions** for callbacks (`jest.fn()`)
- Test **props changes** and **state updates**

Styling in React

85. Inline Styling

❑ Definition

- Styles written **directly inside JSX** using **JavaScript objects**.

- CSS properties are **camelCase**, values are usually **strings**.

☐ **Example**

```
function Button() {  
  const style = { backgroundColor: 'blue', color: 'white', padding: '10px' };  
  return <button style={style}>Click Me</button>;  
}
```

- ☐ Quick, dynamic styling
 - ☐ Not reusable for large apps.
-

86. CSS Modules

☐ **Definition**

- **Locally scoped CSS** to avoid class name collisions.
- File naming: `ComponentName.module.css`

☐ **Example**

```
/* Button.module.css */  
.button {  
  background-color: blue;  
  color: white;  
  padding: 10px;  
}  
import styles from './Button.module.css';  
  
function Button() {  
  return <button className={styles.button}>Click Me</button>;  
}
```

- ☐ Scoped, reusable, avoids global namespace issues.
-

87. Styled Components

☐ **Definition**

- CSS-in-JS library for **component-level styling**.
- Dynamic props support.

❑ Example

npm install styled-components

import styled from 'styled-components';

```
const Button = styled.button`
  background-color: ${props => props.primary ? 'blue' : 'gray'};
  color: white;
  padding: 10px;
`;
```

<Button primary>Click Me</Button>

❑ Dynamic, scoped, keeps style close to component.

88. Emotion / Tailwind CSS in React

❑ Emotion

- Similar to Styled Components
- Supports **css prop** and **styled API**

❑ Tailwind CSS

- Utility-first CSS framework
- Use **className** directly in JSX for styling

```
<button className="bg-blue-500 text-white px-4 py-2 rounded">
  Click Me
</button>
```

❑ Rapid, responsive, and consistent styling.

89. Material UI / Shadcn / Chakra UI

❑ Definition

- **Component libraries** for React with **prebuilt UI components**

❑ Example: Material UI

npm install @mui/material @emotion/react @emotion/styled

```
import Button from '@mui/material/Button';
```

```
<Button variant="contained" color="primary">Click Me</Button>
```

- ❑ Saves development time, accessible, responsive, theme support.
-

❑ Deployment & Build

90. Environment Variables

❑ Definition

- Store **sensitive data or configs** for different environments.
- Prefix with **REACT_APP_** for React apps.

❑ Example

```
REACT_APP_API_URL=https://api.example.com  
console.log(process.env.REACT_APP_API_URL);
```

- ❑ Useful for **API endpoints, keys, feature flags**.
-

91. Production Build (npm run build)

❑ Steps

1. Run build command:

```
npm run build
```

2. Creates **optimized production folder (build)**
 - Minified JS/CSS
 - Tree-shaking for unused code
3. Deploy to **Netlify, Vercel, Firebase, or any server**

❑ Tips

- Set **homepage** in package.json if deploying to subpath
- Use .env files for **environment-specific variables**

- Ready-to-deploy, optimized, and fast-loading React app.