# React.JS CheatSheet

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# 1. INTRODUCTION TO REACT

### 1.1 What is React?

- React is a JavaScript library used to build user interfaces (UIs). That means it helps you create the parts of a website that users can see and interact with.
- React is especially good for building single-page applications (SPAs) — websites that update the content without reloading the entire page. For example, when you scroll Instagram, the page doesn't reload, but new content appears — that's similar to how React works.

**In simple words:**

- React helps you build websites that are fast, easy to manage, and don't reload every time something changes.

### 1.2 History and Features

- **Created by:** Jordan Walke, a software engineer at Facebook
- **Released in:** 2013 (open-sourced)
- **Used by:** Facebook, Instagram, Netflix, WhatsApp Web, etc.

**Key Features:**

- **Component-based:** You build UI in small pieces called components.
- **Reusable:** One component can be used in many places.
- **Virtual DOM:** It updates only the changed part of the page instead of the whole page.
- **Fast and Efficient:** Because of virtual DOM and smart updates.

### 1.3 Virtual DOM Explained

- DOM stands for Document Object Model — it's like a tree structure of your HTML.
- React creates a copy of the real DOM in memory. This copy is called the Virtual DOM.
- When something changes, React compares the virtual DOM with the real DOM and only updates the parts that actually changed.
- This makes the website faster, because it doesn't reload everything.

# 1. INTRODUCTION TO REACT

## 1.4 React vs Vanilla JS vs Other Frameworks

| Feature | React | Vanilla JS | Other Frameworks (e.g., Angular, Vue) |
|---|---|---|---|
| **Structure** | Component-based | No structure | Component-based |
| **Speed** | Very fast | Medium | Fast |
| **Learning Curve** | Easy to Medium | Easy | Medium to Hard |
| **Reusability** | High | Low | High |
| **DOM Updates** | Virtual DOM | Full page reload | Virtual DOM or optimized updates |

- **Conclusion:** React offers a balanced approach — faster than plain JavaScript and easier to learn than some heavy frameworks like Angular.

# 2. ENVIRONMENT SETUP

## 2.1 Installing Node.js and npm

- React apps need Node.js and npm (Node Package Manager) to work.

**Steps:**

1. Go to https://nodejs.org
2. Download the LTS version (Recommended for most users)
3. Install it

- After installing, open your terminal and check versions:

```nginx
node -v
npm -v
```

- If you see the versions, Node.js and npm are installed correctly.

## 2.2 Create React App (CRA)

- React provides a tool called Create React App (CRA) to start building apps quickly.
- To create a project:

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

- “npx” runs a command without installing it globally.
- “create-react-app” sets up everything for you.
- “npm” start runs your app in the browser.

## 2.3 Project Structure

- When you create a React app, you’ll see a folder like this:

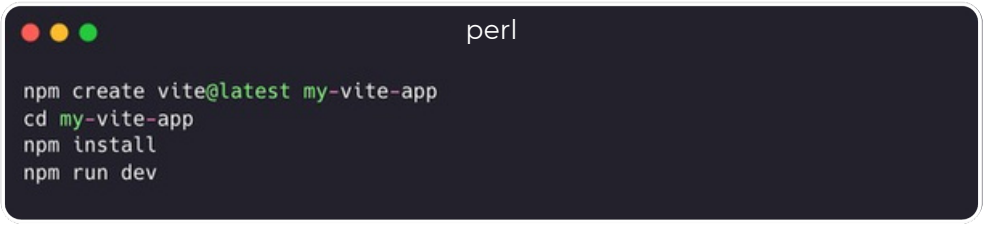
```pgsql
my-app/
  node_modules/
  public/
    index.html
  src/
    App.js
    index.js
  package.json
```

- **public/index.html:** Main HTML file
- **src/index.js:** Starting point of your React app
- **src/App.js:** Main component

# 2. ENVIRONMENT SETUP

### 2.4 Using Vite for Faster Setup

- Vite is a modern build tool that's faster than CRA.
- To create a Vite project:




- Vite starts faster and is better for modern projects.

# 3. JSX BASICS

### 3.1 What is JSX?

- JSX stands for JavaScript XML. It lets you write HTML-like code inside JavaScript.

Example:

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

- This looks like HTML, but it's actually JavaScript code.

### 3.2 JSX Syntax Rules

- Return only one parent element.

```jsx
return (
  <div>
    <h1>Title</h1>
    <p>Paragraph</p>
  </div>
);
```

- Use camelCase for attributes like className, onClick:

```jsx
<button className="btn" onClick={handleClick}>Click Me</button>
```

### 3.3 Embedding Expressions

- You can write JavaScript inside JSX using { }.

Example:

```jsx
const name = "Rudra";
return <h1>Hello, {name}</h1>;
```

### 3.4 JSX Best Practices

- Use meaningful names for variables and components.
- Keep JSX clean and readable.
- Avoid too much logic inside JSX.
- Use fragments (<> </>) if you don’t want to wrap everything in a .

# 4. COMPONENTS

### 4.1 Functional Components

- A simple way to create a component using a function.

```jsx
function Welcome() {
  return <h1>Hello from Functional Component</h1>;
}
```

### 4.2 Class Components

- Another way using ES6 class (older way).

```jsx
class Welcome extends React.Component {
  render() {
    return <h1>Hello from Class Component</h1>;
  }
}
```

### 4.3 Component Naming and Structure

- Always start component names with Capital letters.
- Each component should be in its own file.

```jsx
// File: Header.js
function Header() {
  return <h1>This is a Header</h1>;
}
export default Header;
```

### 4.4 Props in Components

- Props (short for properties) are used to pass data from one component to another.

```jsx
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Used like:

```jsx
<Welcome name="Rudra" />
```

### 4.5 Props Destructuring

- Instead of props.name, you can do:

```jsx
function Welcome({ name }) {
  return <h1>Hello, {name}</h1>;
}
```

# 4. COMPONENTS

### 4.6 Children Props

- Props can also include inner HTML using props.children.

# 5. STATE MANAGEMENT

### 5.1 useState Hook

- For functional components, we use the useState hook to manage data.

jsx

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

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

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

### 5.2 setState in Class Components

- In class components, we use this.state and this.setState().

jsx

```jsx
class Counter extends React.Component {
  constructor() {
    super();
    this.state = { count: 0 };
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={() => this.setState({ count: this.state.count + 1
})}>
          Add
        </button>
      </div>
    );
  }
}
```

### 5.3 Lifting State Up

- When two components need to share the same state, move the state to their common parent.

jsx

```jsx
function Parent() {
  const [value, setValue] = useState("");

  return (
    <>
      <Input value={value} setValue={setValue} />
      <Display value={value} />
    </>
  );
}
```

# 5. STATE MANAGEMENT

### 5.4 Passing State as Props

- You can pass the current state value to another component using props.

# 6. EVENT HANDLING

### 6.1 Handling Events in React

In React, we can handle user actions like clicks, typing, or hovering using event handlers.

Example: Button click

```jsx
function ClickButton() {
  function handleClick() {
    alert("Button was clicked!");
  }

  return <button onClick={handleClick}>Click Me</button>;
}
```

**Note:** We use camelCase like onClick instead of onclick.

### 6.2 Event Binding

- In class components, we often need to bind the event handler to this.

```jsx
class Welcome extends React.Component {
  constructor() {
    super();
    this.state = { name: "Rudra" };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    alert("Hello " + this.state.name);
  }

  render() {
    return <button onClick={this.handleClick}>Greet</button>;
  }
}
```

- In functional components, you don't need binding.

### 6.3 Synthetic Events

- React uses a system called Synthetic Events. It wraps browser events to work the same way across all browsers.
- It behaves like regular JavaScript events but works consistently everywhere.

Example:

```jsx
function handleInput(event) {
  console.log(event.target.value);
}
```

# 6. EVENT HANDLING

### 6.4 Prevent Default and Event Objects

- If you want to stop the default action (like refreshing the page on form submit), use event.preventDefault().

Example:

# 7. CONDITIONAL RENDERING

## 7.1 if/else Statements

- You can use normal if/else logic before the return.

```jsx
function Greeting(props) {
  if (props.isLoggedIn) {
    return <h1>Welcome back!</h1>;
  } else {
    return <h1>Please log in.</h1>;
  }
}
```

## 7.2 Ternary Operators

- Short way to write if/else inside JSX:

```jsx
return (
  <div>
    {isLoggedIn ? <h1>Hi User</h1> : <h1>Please Sign In</h1>}
  </div>
);
```

## 7.3 Logical && Operator

- Use && when you want to render something only if a condition is true.

```jsx
{cart.length > 0 && <p>You have {cart.length} items in cart</p>}
```

- If cart.length is 0, it won't show anything.

## 7.4 Switch Statements

- For multiple conditions, you can use switch.

```jsx
function StatusMessage({ status }) {
  switch (status) {
    case "loading":
      return <p>Loading...</p>;
    case "success":
      return <p>Data loaded!</p>;
    case "error":
      return <p>Error occurred.</p>;
    default:
      return <p>Unknown status</p>;
  }
}
```

# 8. LISTS AND KEYS

## 8.1 Rendering Lists with .map()

- When you want to show many items, use .map() to loop through them.

```jsx
const fruits = ["Apple", "Banana", "Mango"];

function FruitList() {
  return (
    <ul>
      {fruits.map((fruit) => <li>{fruit}</li>)}
    </ul>
  );
}
```

## 8.2 Using key Props

- Each item in a list needs a unique key so React can track changes.

```jsx
{fruits.map((fruit, index) => (
  <li key={index}>{fruit}</li>
))}
```

- Use a unique ID if available (don't always rely on index).

## 8.3 Handling Empty Lists

- You can check if the list is empty before showing it.

```jsx
{fruits.length === 0 ? (
  <p>No fruits available</p>
) : (
  <ul>{fruits.map(fruit => <li>{fruit}</li>)}</ul>
)}
```

# 9. FORMS IN REACT

## 9.1 Controlled vs Uncontrolled Components

- Controlled Component: React controls the input value using useState.
- Uncontrolled Component: DOM handles the input using a ref.

Controlled example:

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

<input value={name} onChange={(e) => setName(e.target.value)} />
```

## 9.2 Handling Inputs (text, checkbox, radio, select)

- **Text:**

```jsx
<input type="text" value={name} onChange={(e) => setName(e.target.value)} />
```

- **Checkbox:**

```jsx
<input type="checkbox" checked={isChecked} onChange={(e) =>
setIsChecked(e.target.checked)} />
```

- **Radio:**

```jsx
<input type="radio" value="Male" checked={gender === "Male"} onChange={(e) =>
setGender(e.target.value)} />
```

- **Select dropdown:**

```jsx
<select value={city} onChange={(e) => setCity(e.target.value)}>
  <option value="Delhi">Delhi</option>
  <option value="Mumbai">Mumbai</option>
</select>
```

## 9.3 Form Submission

```jsx
function MyForm() {
  const [name, setName] = useState("");

  function handleSubmit(e) {
    e.preventDefault();
    alert("Submitted name: " + name);
  }

  return (
    <form onSubmit={handleSubmit}>
      <input value={name} onChange={(e) => setName(e.target.value)} />
      <button>Submit</button>
    </form>
  );
}
```

# 9. FORMS IN REACT

## 9.4 Form Validation Basics

- Before submitting, check if the inputs are correct.

jsx

```jsx
function handleSubmit(e) {
  e.preventDefault();
  if (name === "") {
    alert("Name is required");
    return;
  }
  // Continue with form submission
}
```

# 10. USEEFFECT HOOK

### 10.1 Basic Usage of useEffect

- “useEffect” lets you run code after your component renders.

Example:

```jsx
useEffect(() => {
  console.log("Component mounted!");
});
```

### 10.2 Dependency Array

- You can control when useEffect runs using dependencies.

```jsx
useEffect(() => {
  console.log("Name changed!");
}, [name]);
```

- Empty array []: run only once (like on mount)
- With [name]: run when name changes

### 10.3 Cleanup Function

- If you want to clean up something (like removing a timer), return a function inside useEffect.

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

  return () => {
    clearInterval(timer); // cleanup
  };
}, []);
```

### 10.4 Fetching Data on Mount

```jsx
useEffect(() => {
  fetch("https://api.example.com/data")
    .then(res => res.json())
    .then(data => setData(data));
}, []);
```

- It runs once after the component loads and gets the data.

# 11. OTHER REACT HOOKS

### 11.1 useRef

- useRef is used to store a value that doesn't need to re-render the component.
- It's also used to directly access a DOM element (like an input box).

jsx

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

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

  function focusInput() {
    inputRef.current.focus(); // Focus the input box
  }

  return (
    <>
      <input ref={inputRef} />
      <button onClick={focusInput}>Focus Input</button>
    </>
  );
}
```

### 11.2 useContext

- It helps to share values like user info or theme between components without props.

jsx

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

const ThemeContext = createContext();

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

function Child() {
  const theme = useContext(ThemeContext);
  return <div>The theme is {theme}</div>;
}
```

# 11. OTHER REACT HOOKS

### 11.3 useReducer

- Like useState, but better when state is complex (like a counter with actions).

jsx

```jsx
import { useReducer } from 'react';

function reducer(state, action) {
  if (action.type === 'increment') return { count: state.count + 1 };
  return state;
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <>
      <p>{state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>+1</button>
    </>
  );
}
```

### 11.4 useMemo

- Stops a function from running again if the inputs (dependencies) didn't change.

jsx

```jsx
const expensiveValue = useMemo(() => {
  return slowCalculation(num);
}, [num]);
```

### 11.5 useCallback

- Stops a function from re-creating again and again.

jsx

```jsx
const handleClick = useCallback(() => {
  console.log('clicked');
}, []);
```

### 11.6 Custom Hooks

- Your own reusable hook logic using other hooks.

jsx

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

# 12. REACT ROUTER (V6+)

## 12.1 Installation

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

## 12.2 BrowserRouter, Routes, Route

```jsx
import { BrowserRouter, Routes, Route } from 'react-router-dom';

<BrowserRouter>
  <Routes>
    <Route path=\"/\" element={<Home />} />
    <Route path=\"/about\" element={<About />} />
  </Routes>
</BrowserRouter>
```

## 12.3 Link and NavLink

```jsx
<Link to=\"/about\">Go to About</Link>

<NavLink to=\"/about\">About</NavLink>
```

## 12.4 Route Parameters

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

- Use useParams() in User component to get the id.

## 12.5 Nested Routes

```jsx
<Route path=\"/dashboard\" element={<Dashboard />}>
  <Route path=\"stats\" element={<Stats />} />
</Route>
```

## 12.6 Redirects and Navigate

```jsx
<Navigate to=\"/login\" />
```

## 12.7 404 Page

```jsx
<Route path=\"*\" element={<NotFound />} />
```

# 13. CONTEXT API

## 13.1 Creating and Providing Context

```jsx
const UserContext = createContext();

<UserContext.Provider value={user}>
  <App />
</UserContext.Provider>
```

## 13.2 Consuming Context

```jsx
const user = useContext(UserContext);
```

## 13.3 Updating Context

- Use useState and pass both value and function via value.

## 13.4 Context vs Props

- Use props if data is used only by child.
- Use context when many components need the data.

# 14. STATE MANAGEMENT LIBRARIES

### 14.1 Redux (Basics)

- A central store to keep all app data.
- Use actions to update data.

### 14.2 Redux Toolkit

- Easier way to write Redux

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

### 14.3 useSelector and useDispatch

```jsx
const count = useSelector(state => state.counter);
const dispatch = useDispatch();
```

### 14.4 Zustand

- Lightweight library, less setup.

```jsx
const useStore = create(set => ({
  count: 0,
  increase: () => set(state => ({ count: state.count + 1 }))
}));
```

### 14.5 Recoil

- Works with atoms (like small pieces of state).

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

### 14.6 Context vs Redux

| Use Case | Tool |
| --- | --- |
| Small data like theme/user | Context |
| Big apps with many updates | Redux |

# 15. STYLING IN REACT

## 15.1 CSS Modules

jsx

```jsx
import styles from './App.module.css';
<div className={styles.box}>Text</div>
```

## 15.2 Inline Styling

jsx

```jsx
<div style={{ color: 'blue', fontSize: '20px' }}>Hello</div>
```

## 15.3 Styled Components

jsx

```jsx
const Button = styled.button`
  background: red;
  color: white;
`;
```

## 15.4 Tailwind CSS

jsx

```jsx
npm install -D tailwindcss
```

jsx

```jsx
<div className=\"bg-blue-500 text-white p-4\">Text</div>
```

## 15.5 SASS / SCSS

jsx

```jsx
npm install sass
```

jsx

```jsx
$color: blue;
.title { color: $color; }
```

# 16. REACT LIFECYCLE (CLASS COMPONENTS)

- In classcomponents, lifecycle methodsletyouruncodeatspecific times in a component's life (mount, update, unmount).

### 16.1 constructor()

- Runs first, when the component is created.
- Used to initialize state and bind methods.

```jsx
class MyComponent extends React.Component {
  constructor() {
    super();
    this.state = { count: 0 };
  }
}
```

### 16.2 render()

- Required method.
- Returns JSX to display on the screen.

```jsx
render() {
  return <h1>Hello</h1>;
}
```

### 16.3 componentDidMount()

- Called after component is added to the DOM.
- Good for API calls or timers.

```jsx
componentDidMount() {
  console.log('Component mounted');
}
```

### 16.4 componentDidUpdate()

- Runs when props or state change.
- Good for updating based on changes.

```jsx
componentDidUpdate(prevProps, prevState) {
  if (this.state.count !== prevState.count) {
    console.log('Count changed');
  }
}
```

# 16. REACT LIFECYCLE (CLASS COMPONENTS)

### 16.5 componentWillUnmount()

- Called before component is removed.
- Use it to clear timers or remove listeners

# 17. ERROR HANDLING

## 17.1 Error Boundaries

- A special class component that catches JavaScript errors in children.

jsx

```jsx
class ErrorBoundary extends React.Component {
  constructor() {
    super();
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    console.log(error, info);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}
```

## 17.2 Try-Catch in Event Handlers

jsx

```jsx
function handleClick() {
  try {
    // risky code
  } catch (error) {
    console.log('Error:', error);
  }
}
```

## 17.3 Handling Async Errors

jsx

```jsx
async function fetchData() {
  try {
    const res = await fetch('api/data');
    const data = await res.json();
  } catch (err) {
    console.log('Failed to fetch:', err);
  }
}
```

# 18. CODE SPLITTING AND LAZY LOADING

## 18.1 React.lazy() and Suspense

- React.lazy() loads components only when needed.

jsx

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

<Suspense fallback={<div>Loading...</div>}>
  <LazyComponent />
</Suspense>
```

## 18.2 Dynamic Imports

- Load parts of the code only when needed (saves load time).

jsx

```jsx
import('./HeavyComponent').then(module => {
  const Comp = module.default;
});
```

## 18.3 Loading Spinners

- Use during lazy loading or data fetching.

jsx

```jsx
<Suspense fallback={<Spinner />}>
  <LazyComponent />
</Suspense>
```

# 19. REFS AND DOM MANIPULATION

## 19.1 useRef andAccessingDOM Elements

- useRef() returns a reference to a DOM node.

```jsx
const inputRef = useRef();

<input ref={inputRef} />
<button onClick={() => inputRef.current.focus()}>Focus</button>
```

## 19.2 Forwarding Refs

- Pass a ref to child from parent.

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

## 19.3 DOM Interaction Best Practices

- Prefer React state.
- Use refs only when you must access DOM (e.g., focus, scroll).
- Avoid modifying DOM directly (don’t use document.querySelector).

# 20. PORTALS IN REACT

## 20.1 What is a Portal?

- Normally, in React, all components render inside a single root element — like this:

jsx

```jsx
<div id="root"></div>
```

- But what if you want to show something outside this root?
- For example, a popup, a modal, or a tooltip that appears on top of everything.
- That's where Portals help.
- A Portal lets you render a React component into a different part of the HTML, outside the main app root.

## 20.2 How to Create a Portal

**Step 1:** Add a new HTML element in index.html

- Open your public/index.html file and add a new  for the portal:

jsx

```jsx
<body>
  <div id="root"></div>
  <div id="portal-root"></div> <!-- This is where portal content will go -->
</body>
```

**Step 2:** Use createPortal() in your component

jsx

```jsx
import React from "react";
import ReactDOM from "react-dom/client";

// PortalComponent.jsx
function PortalComponent() {
  return ReactDOM.createPortal(
    <div style={{ backgroundColor: "white", padding: "20px", border: "2px
solid black" }}>
      <h2>This is inside a portal!</h2>
    </div>,
    document.getElementById("portal-root") // Render this inside #portal-root
  );
}

export default PortalComponent;
```

- You can now use this PortalComponent in your app.

# 20. PORTALS IN REACT

## 20.3 Full Example: Modalwith Portal

- Let's create a simple modal that opens and closes using a portal.

**Modal. jsx**

jsx

```jsx
import React from "react";
import ReactDOM from "react-dom";

function Modal({ onClose, children }) {
  return ReactDOM.createPortal(
    <div style={styles.overlay}>
      <div style={styles.modal}>
        {children}
        <button onClick={onClose}>Close</button>
      </div>
    </div>,
    document.getElementById("portal-root")
  );
}

const styles = {
  overlay: {
    position: "fixed",
    top: 0, left: 0,
    width: "100%", height: "100%",
    backgroundColor: "rgba(0,0,0,0.5)",
    display: "flex",
    justifyContent: "center",
    alignItems: "center",
    zIndex: 1000,
  },
  modal: {
    background: "white",
    padding: "20px",
    borderRadius: "10px",
    boxShadow: "0 0 10px black"
  }
};

export default Modal;
```

**App. jsx**

jsx

```jsx
import React, { useState } from "react";
import Modal from "./Modal";

function App() {
  const [showModal, setShowModal] = useState(false);

  return (
    <div>
      <h1>Welcome to My App</h1>
      <button onClick={() => setShowModal(true)}>Open Modal</button>

      {showModal && (
        <Modal onClose={() => setShowModal(false)}>
          <h2>This is a Modal</h2>
          <p>You can put anything here!</p>
        </Modal>
      )}
    </div>
  );
}

export default App;
```

# 20. PORTALS IN REACT

## 20.4 Use Cases of Portals

### Modals/Popups

- When you want a window to appear on top of everything else.

### Tooltips

- Small hint boxes that appear on hover.

### Dropdown Menus

- If a dropdown needs to overflow a parent with overflow: hidden.

### Sidebars or Toast Messages

- For alerts or temporary notifications.

## 20.5 Why Use Portals?

- It solves CSS issues when the parent component has overflow: hidden, z-index, or position: relative.
- Keeps UI elements like modals and tooltips outside the normal hierarchy.
- Cleaner and more flexible design.

# 21. FRAGMENTS AND STRICTMODE

## 21.1 <React.Fragment> and <>...</>

- In React, a component must return a single parent element.
- But what if you need to return multiple elements without wrapping them in a ?
- Use Fragments!

Example:

jsx

```jsx
// Using <div> (adds extra HTML)
return (
  <div>
    <h1>Hello</h1>
    <p>World</p>
  </div>
);

// Using <React.Fragment> (no extra HTML)
return (
  <React.Fragment>
    <h1>Hello</h1>
    <p>World</p>
  </React.Fragment>
);

// Shorter syntax: <>
return (
  <>
    <h1>Hello</h1>
    <p>World</p>
  </>
);
```

**Why Use It?**

No extra  tags in the DOM.

Keeps code clean.

## 21.2 <React.StrictMode>

- React provides a special wrapper that helps you find bugs early.

jsx

```jsx
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

**What it does:**

- Warns about unsafe code.
- Helps with future-proofing your app.
- Does not show anything to the user.
- Works only in development (not in production).

# 22. HIGHER ORDER COMPONENTS (HOC)

### 22.1 What is an HOC?

- An HOC is a function that takes a component and returns a new component.

```jsx
const EnhancedComponent = withFeature(OriginalComponent);
```

- Think of it like adding extra powers to a component.

### 22.2 Example: withLogger HOC

```jsx
function withLogger(WrappedComponent) {
  return function Enhanced(props) {
    console.log("Props:", props);
    return <WrappedComponent {...props} />;
  };
}
```

- Now use it:

```jsx
const MyComponentWithLogger = withLogger(MyComponent);
```

### 22.3 Common HOC Use Cases

- Logging props
- Adding loading spinners
- Access control (admin vs normal user)
- Reusing logic across components

### 22.4 Pros and Cons

**Pros**

- Reuse logic easily
- Cleaner components

**Cons**

- Can become complex to debug
- Not as popular now (Hooks are preferred)

# 23. CUSTOM HOOKS

## 23.1 When to Create Custom Hooks

- When you find yourself copying the same useEffect, useState, or logic in multiple places — create a custom hook.

## 23.2 Example: useLocalStorage

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

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];
}
```

Usage:

```jsx
const [name, setName] = useLocalStorage("username", "Guest");
```

## 23.3 Another Example: useWindowSize

```jsx
function useWindowSize() {
  const [size, setSize] = useState([window.innerWidth, window.innerHeight]);

  useEffect(() => {
    const handleResize = () => setSize([window.innerWidth,
window.innerHeight]);
    window.addEventListener("resize", handleResize);
    return () => window.removeEventListener("resize", handleResize);
  }, []);

  return size;
}
```

Usage:

```jsx
const [width, height] = useWindowSize();
```

## 23.4 Best Practices

- Always start with use
- Return only what's needed
- Keep it simple

# 24. TESTING REACT APPS

## 24.1 Unit Testing withJest

- Jest is a testing framework for JavaScript.

✅ Example test:

jsx

```jsx
// sum.js
export function sum(a, b) {
  return a + b;
}

// sum.test.js
import { sum } from "./sum";

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

## 24.2 Component Testing with React Testing Library

React Testing Library helps you test how a user would interact.

jsx

```jsx
// Greeting.js
function Greeting() {
  return <h1>Hello User!</h1>;
}

// Greeting.test.js
import { render, screen } from "@testing-library/react";
import Greeting from "./Greeting";

test("shows greeting text", () => {
  render(<Greeting />);
  expect(screen.getByText("Hello User!")).toBeInTheDocument();
});
```

## 24.3 Testing Hooks

- For hooks, use @testing-library/react-hooks or test them inside a component.

jsx

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

- Test using fireEvent from the testing library.

# 25. DEPLOYMENT

### 25.1 Build for Production

- Run this in the terminal:

```bash
npm run build
```

- It creates a build/ folder with all optimized files.

### 25.2 Hosting on Vercel

1. Go to https://vercel.com
2. Connect your GitHub repo
3. Click "Deploy"

Done ✅

### 25.3 Hosting on Netlify

1. Go to https://netlify.com
2. Drag your build folder
3. Or connect GitHub and deploy automatically

### 25.4 Hosting on Firebase

```bash
npm install -g firebase-tools
firebase login
firebase init
firebase deploy
```

- Choose "Hosting" and follow the steps.

### 25.5 GitHub Pages

- Install the package:

```bash
npm install gh-pages --save-dev
```

- Add to package.json:

```json
"homepage": "https://your-username.github.io/your-repo"
```

- Then run:

```bash
npm run build
npm run deploy
```

# 25. DEPLOYMENT

### 25.6 Environment Variables

- You can use environment variables for API keys or settings.
- Create a .env file:

env

```
REACT_APP_API_URL=https://example.com/api
```

- Use in code:

jsx

```
const url = process.env.REACT_APP_API_URL;
```

# 26. REACT WITH APIS

## 26.1 Fetching with fetch and axios

➤ Using fetch:

```jsx
useEffect(() => {
  fetch("https://api.example.com/data")
    .then(res => res.json())
    .then(data => setData(data))
    .catch(err => console.error(err));
}, []);
```

➤ Using axios:

```jsx
import axios from "axios";

useEffect(() => {
  axios.get("https://api.example.com/data")
    .then(res => setData(res.data))
    .catch(err => console.error(err));
}, []);
```

## 26.2 REST vs GraphQL

| Feature | REST | GraphQL |
|---|---|---|
| **Structure** | Multiple endpoints | Single endpoint |
| **Data Fetching** | Fixed response | Flexible (you choose fields) |
| **Overfetching** | Yes (extra data) | No |

## 26.3 Error Handling in API Calls

```jsx
try {
  const res = await axios.get("api/url");
  setData(res.data);
} catch (error) {
  console.error("Something went wrong", error);
}
```

# 26. REACT WITH APIS

## 26.4 Displaying Loading States

jsx

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

useEffect(() => {
  fetch("api")
    .then(res => res.json())
    .then(data => {
      setData(data);
      setLoading(false);
    });
}, []);

return loading ? <p>Loading...</p> : <p>Data Loaded</p>;
```

# 27. BEST PRACTICES & OPTIMIZATION

## 27.1 Component Reusability

- Create reusable components like Button, Input, Card instead of repeating code.

## 27.2 Code Splitting

- Use React.lazy() to load components only when needed.

jsx

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

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

## 27.3 Avoiding Unnecessary Renders

- Use React.memo to prevent re-render when props haven't changed.

jsx

```jsx
const MyComponent = React.memo(({ name }) => {
  return <p>{name}</p>;
});
```

## 27.4 Memoization

- Use useMemo or useCallback for expensive calculations or stable function references.

jsx

```jsx
const memoizedValue = useMemo(() => expensiveFunction(value), [value]);
```

## 27.5 Folder Structure Recommendation

css

```
src/
│
├── components/
│   └── Button.jsx
├── pages/
│   └── Home.jsx
├── hooks/
│   └── useAuth.js
├── utils/
│   └── helpers.js
├── assets/
│   └── logo.png
└── App.jsx
```

# 28. REACT WITH TYPESCRIPT (BONUS)

## 28.1 Why Use TypeScript?

- Catch errors before runtime
- Better developer experience with IntelliSense
- Safer code with types

## 28.2 Typing Props and State

tsx

```tsx
type Props = {
  name: string;
};

function Welcome({ name }: Props) {
  return <h1>Hello, {name}</h1>;
}
```

tsx

```tsx
const [count, setCount] = useState<number>(0);
```

## 28.3 Typing useEffect and useRef

tsx

```tsx
useEffect(() => {
  // Your logic
}, []);

const inputRef = useRef<HTMLInputElement>(null);
```

## 28.4 Interfaces and Types

tsx

```tsx
interface User {
  name: string;
  age: number;
}

const user: User = {
  name: "Ravi",
  age: 20,
};
```

# 29. POPULAR UI LIBRARIES FOR REACT

### Material UI (MUI)

- Google's design system
- Prebuilt styled components

bash

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

### Ant Design

- Enterprise UI with lots of components

bash

```
npm install antd
```

### Chakra UI

- Easy-to-use, accessible components

bash

```
npm install @chakra-ui/react @emotion/react @emotion/styled framer-motion
```

### Tailwind CSS

- Utility-first CSS framework

bash

```
npm install -D tailwindcss
npx tailwindcss init
```

### ShadCN/UI

- Beautiful components built with Tailwind, Radix UI

bash

```
npm install @shadcn/ui
```

# 30. COMMON MISTAKES & PRACTICE TASKS (BONUS SECTION)

## 31.1 Common Beginner Mistakes in React

| Mistake | Why It's a Problem | Correct Approach |
|---|---|---|
| **Mutating state directly** | React won't detect changes and re-render | Always use setState() or setSomething() |
| **Missing key in lists** | React has trouble tracking items | Use unique key props in .map() |
| **Incorrect useEffect usage** | Missing dependency array can cause infinite loops | Include the correct dependencies or use [] for run-once |
| **Calling hooks conditionally** | Breaks the rules of hooks | Always call hooks at the top level of the component |
| **Overusing props instead of state** | Can lead to prop drilling | Use context or state where appropriate |

## 31.2 Practice Tasks for Beginners

| # | Task | Concepts Practiced |
|---|---|---|
| 1 | Counter with +/− buttons | useState, events |
| 2 | Form with submit handler | Forms, controlled inputs |
| 3 | Render a favorite movie list | .map(), JSX, keys |
| 4 | Toggle dark/light mode | useState, conditional rendering |
| 5 | Joke fetcher app | useEffect, fetch API |
| 6 | Basic login page | Input handling, form validation |
| 7 | Like button toggle | useState, events |
| 8 | Dropdown list | JSX, select elements |
| 9 | Timer/Stopwatch | useEffect, setInterval, clearInterval |
| 10 | Weather app | API integration, conditional UI |

### Tip for Learners

- Try one small project every 2–3 days. Don't aim for perfection. Aim for progress.