**`useReducer`**

\* For complex state logic involving multiple related state variables or when the next state depends on the previous state, `useReducer` is preferred over multiple `useState` calls.

\* It uses a \*\*reducer function\*\* that takes current state and an action, then returns a new state based on the action.

\* This pattern resembles Redux but is built into React.

Benefits:

\* Centralizes state updates.

\* Makes state transitions predictable.

\* Simplifies debugging and testing.

Example:

```jsx

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;

}

}

function Counter() {

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

return (

<>

<button onClick={() => dispatch({ type: 'decrement' })}>-</button>

<span>{state.count}</span>

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

</>

);

}

```

---

**Custom Hooks**

\* Custom hooks are reusable functions prefixed with `use` that encapsulate common stateful logic.

\* They allow you to abstract and share logic between components without duplicating code.

\* Helps keep components clean and focused on rendering UI.

Example: Managing form inputs with a custom hook

```jsx

function useInput(initialValue) {

const [value, setValue] = React.useState(initialValue);

const onChange = e => setValue(e.target.value);

return { value, onChange };

}

function Form() {

const name = useInput('');

const email = useInput('');

return (

<form>

<input type="text" {...name} placeholder="Name" />

<input type="email" {...email} placeholder="Email" />

</form>

);

}

```

This pattern dramatically reduces boilerplate in forms.

**🏷️ Props & Data Passing**

**Props**

\* Props are the primary method of passing data \*\*down the component tree\*\*.

\* They make components \*\*configurable and reusable\*\* by allowing parent components to send different data or behavior to children.

\* Props are \*\*immutable\*\* inside the child component—children should never try to modify props directly.

\* If the child needs to manage dynamic data, it should use state internally or receive callback props to notify the parent of changes.

```jsx

function Greeting({ name }) {

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

}

<Greeting name="Alice" />

```

**Props Drilling**

\* When many layers of components pass props just to get data from top-level to deeply nested child, it’s called “props drilling.”

\* This can make code harder to maintain and read.

\* It also leads to unnecessary re-rendering of intermediate components.

**Solutions**:

\* Use \*\*React Context API\*\* to provide data globally or scoped within a tree.

\* Use external state management libraries like \*\*Redux\*\*, \*\*MobX\*\*, or others.

---

. **Functional Components and Props**

\* Functional components receive props as their argument (typically destructured).

\* They are simpler and easier to test than class components.

\* This style encourages pure components that just render UI based on input props.

```jsx

function Button({ label, onClick }) {

return <button onClick={onClick}>{label}</button>;

}