React useReducer – Senior-Level Notes

1️⃣ Why useReducer Exists (The Real Problem)

`useReducer` does not exist because `useState` is bad; it exists because `useState` does not scale well for complex state logic.

Problems with useState in Real Applications

- State becomes object-heavy
- Multiple fields change together
- Updates depend on previous state
- Business logic is scattered across handlers
- No clear definition of why state changed

Example: Real-World Issue with useState

```
const [user, setUser] = useState({
  name: "",
  email: "",
  isLoading: false,
  error: null
});

setUser({ ...user, isLoading: true });
setUser({ ...user, name: "Rohit" }); // ❌ stale snapshot risk
```

Why Seniors Dislike This

- Logic is implicit
- Easy to introduce bugs
- No traceable state transitions
- Hard to reason during debugging

2️⃣ Batching – What It Is and What It Is NOT

What Batching Actually Means

Batching is a React optimization where multiple state updates are grouped into **one render**:

```
setA(1);
setB(2);
// only one render occurs
```

Important Facts

- Applies to: `useState`, `useReducer`, class `setState`
- React 18 batches updates in: Events, Promises, setTimeout, Async callbacks

❌Incorrect Conclusion: “useReducer is better because it batches updates”

✅Fact: Batching is **not a differentiator**

3️⃣The Real Cause of Stale State

Stale state is caused by **JavaScript closures**, NOT batching.

```
const [count, setCount] = useState(0);
setCount(count + 1);
setCount(count + 1);
// Final result = 1, not 2
```

- Each render captures a fixed snapshot of state.

4️⃣Functional useState Solves Stale State

```
setCount(prev => prev + 1);
setCount(prev => prev + 1);
// prev = 0 → 1
// prev = 1 → 2
```

- ✔️No stale closures - ✔️Always latest state

⚠️Note: Stale state is **NOT** the reason `useReducer` exists.

5️⃣So Why Do We Still Need useReducer?

Functional `useState` fixes stale state but **does not fix architectural complexity**.

```
setUser(prev => ({ ...prev, isLoading: true }));
setUser(prev => ({ ...prev, name: "Rohit" }));
setUser(prev => ({ ...prev, error: null }));
```

Issues

- Logic is spread everywhere

- No single source of truth
- No concept of “event”
- Hard to understand intent

6️⃣ What useReducer Actually Is (Core Idea)

`useReducer` is a **predictable state machine for UI**. - Instead of “set this value”, you say: **“This event happened → calculate next state”**

7️⃣ Core Mental Model

```
(previousState, action) → nextState
```

Concepts

Term	Meaning
state	Current snapshot
action	What happened
reducer	Pure function deciding next state
dispatch	Triggers transition

Reducer Rules

- Must be pure
- No side effects
- No mutation
- Same input → same output

8️⃣ Why Reducers Never Get Stale State

Critical Internal Difference

- `useState` → Reads from closure
- `useReducer` → React injects latest internal state

```
dispatch({ type: "INC" });
dispatch({ type: "INC" });
// React internally: state1 = reducer(0, INC), state2 = reducer(1, INC)
```

- ✔️Sequential - ✔️Predictable - ✔️No stale data

9️⃣ Real-World Example (Authentication State Machine)

```
const initialState = {
  user: null,
  isLoading: false,
  error: null
};

function authReducer(state, action) {
  switch (action.type) {
    case "LOGIN_START":
      return { ...state, isLoading: true, error: null };
    case "LOGIN_SUCCESS":
      return { user: action.payload, isLoading: false, error: null };
    case "LOGIN_ERROR":
      return { ...state, isLoading: false, error: action.payload };
    case "LOGOUT":
      return initialState;
    default:
      return state;
  }
}

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

⬜ This is **Redux's core idea**

🔟 useState vs useReducer (Senior Comparison)

Aspect	useState (functional)	useReducer
Stale state	❌ fixed	❌ fixed
Batching	✅	✅
Centralized logic	❌	✅
Explicit events	❌	✅
State machine model	❌	✅
Scalability	⚠️	✅

1️⃣1️⃣ useReducer + useContext (Redux Without Redux)

```
const AppContext = createContext();
```

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

1️⃣2️⃣ Side Effects – Where They Belong

- ❌ Never inside reducer
- ✔️ Always in useEffect

```
useEffect(() => {
  if (state.isLoading) {
    login()
      .then(user => dispatch({ type: "LOGIN_SUCCESS", payload: user }))
      .catch(err => dispatch({ type: "LOGIN_ERROR", payload: err }));
  }
}, [state.isLoading]);
```

1️⃣3️⃣ Performance Notes (Senior Level)

- Reducers are cheap
- `dispatch` reference is stable
- No need for `useCallback(dispatch)`
- Performance issues rarely come from reducers

1️⃣4️⃣ Common Mistakes (Interview Traps)

- ❌ Mutating state
- ❌ Side effects in reducer
- ❌ Too many actions for trivial state
- ❌ Using reducer where useState is enough

1️⃣5️⃣ When to Use useReducer (Rule of Thumb)

- Multiple fields change together
- Updates depend on previous state
- UI follows a workflow (loading → success → error)
- State represents business logic

1️⃣6️⃣ One-Line Senior Explanation (Memorize)

“useReducer is a predictable state container that models UI behavior as explicit state transitions rather than implicit mutations.”

1️⃣7️⃣ Interview Answer (Final)

“Functional useState fixes stale closures, but useReducer exists to manage complex, interdependent state with centralized, predictable transitions. Batching is a React optimization, not a reason to choose reducers.”