useReducer: State With Rules

State with a rule book: reducer, actions and dispatch, a cart that only changes by the rules, and the bridge from useState thinking to Redux thinking.

Introduction

`useState` is perfect while a value changes in simple ways. But when state has **rules** — an item can only be removed if it exists, a total must never drop below zero — scattered setter calls start breaking them. **`useReducer`** puts every change through one gatekeeper function: components **dispatch actions** ("what happened"), and the **reducer** decides the new state by the rules. *(Roman Urdu: useReducer mein state seedha nahi badalta — pehle action jata hai, reducer qanoon se faisla karta hai.)*

Key points

- **Three parts:** the **reducer** — a pure function `(state, action) => newState`; the **actions** — small objects like `{ type: "add", item }` describing what happened; and **dispatch** — the function components call to send an action in.
- **Setup:** `const [state, dispatch] = useReducer(reducer, initialState);` — it returns the current state and dispatch, like useState's pair but with a gatekeeper instead of a raw setter.
- **The reducer never mutates:** it returns a **new** object/array built from the old one (spread again). Same rule as state updates, enforced harder because every change passes through here.

  (Roman Urdu: Reducer purani state ko chhedta nahi — hamesha nayee copy bana kar wapas bhejta hai.)
- **Unknown actions return the state unchanged:** the `default:` case is not decoration — dispatching an action the reducer does not know must be a safe no-op.
- **When it earns its keep:** several values that change together (a cart's items + total), changes with validation rules, or many event types touching the same state. For one simple value, `useState` stays the honest tool.
- **This is Redux thinking in miniature:** store, actions with `type`, pure reducers — lesson 5.11 scales exactly this pattern to the whole app, so learning it here makes Redux feel familiar, not new.

Code examples

Example 1 — A counter with rules

```
import { useReducer } from "react";

function counterReducer(state, action) {
  switch (action.type) {
    case "add":
      return state + 1;
    case "minus":
      return state > 0 ? state - 1 : state;
    case "reset":
      return 0;
    default:
      return state;
  }
}

function RuleCounter() {
  const [count, dispatch] = useReducer(counterReducer, 0);
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => dispatch({ type: "add" })}>Add</button>
      <button onClick={() => dispatch({ type: "minus" })}>Minus</button>
      <button onClick={() => dispatch({ type: "reset" })}>Reset</button>
    </div>
  );
}
```

What it does: The buttons never touch the state — they only announce events. The reducer enforces the lesson's rule (the count may never go below zero) in exactly one place, so no button anywhere can break it.

What you will see: Add climbs freely; Minus stops dead at 0 no matter how often it is clicked — the rule lives in the reducer, not in the button.

Example 2 — A cart that changes only by the rules

```
import { useReducer } from "react";

function cartReducer(items, action) {
  switch (action.type) {
    case "add":
      return [...items, action.item];
    case "remove":
      return items.filter((item) => item.id !== action.id);
    case "clear":
      return [];
    default:
      return items;
  }
}

function Cart() {
  const [items, dispatch] = useReducer(cartReducer, []);
  const total = items.reduce((sum, item) => sum + item.price, 0);

  return (
    <div>
      <button onClick={() => dispatch({ type: "add", item: { id: Date.now(), name: "Chai",
price: 250 } })}>
        Add Chai (Rs 250)
      </button>
      <p>Items: {items.length} · Total: Rs {total}</p>
      <ul>
        {items.map((item) => (
          <li key={item.id}>
            {item.name} — Rs {item.price}
            <button onClick={() => dispatch({ type: "remove", id: item.id })}>Remove</button>
          </li>
        ))}
      </ul>
      <button onClick={() => dispatch({ type: "clear" })}>Clear cart</button>
    </div>
  );
}
```

What it does: Three legal moves exist in the whole app — add, remove, clear — and each is one reducer case returning a fresh array. The total is **derived during render** from the items, never stored, so it can never disagree with the list.

What you will see: every "Add Chai" grows the list and the total together; Remove deletes exactly that row; Clear empties the cart and zeroes the total.

Example 3 — From useState thinking to reducer thinking

```
// useState style: the form logic is scattered across handlers
function FormOld() {
  const [name, setName] = useState("");
  const [error, setError] = useState("");
  function submit() {
    if (name.length < 3) setError("Name is too short");
    else setError("");
  }
  return null;
}

// useReducer style: one state, one rule book
function formReducer(state, action) {
  switch (action.type) {
    case "type":
      return { ...state, name: action.value };
    case "submit":
      return state.name.length < 3
        ? { ...state, error: "Name is too short" }
        : { ...state, error: "", sent: true };
    default:
      return state;
  }
}
```

What it does: The old style keeps related values in separate states updated from separate places. The reducer style keeps the form in **one** state object and routes every event — typing, submitting — through one function where the rules sit side by side and stay consistent.

What you will see: same form behaviour, but in the second version every rule can be read (and tested) in a single small function.

Quick recap

- `useReducer` routes every state change through a **reducer**: components `dispatch({ type: ... })`, the reducer returns the new state by the rules.
- Reducers are pure and never mutate — new copies out, unchanged state for unknown actions.
- It is the right tool when state has rules or several values move together — and it is **Redux in miniature**.

Common mistakes

1. **Mutating inside the reducer** — `state.items.push(x); return state;` hands back the same object, and React may skip the render. The fix: build a new value every time — `return [...state, x]`, `{ ...state, done: true }`.
2. **Forgetting the default case** — an unmatched action falls through and returns `undefined`, and the state vanishes. The fix: `default: return state;` — always, even in tiny reducers.
3. **Reaching for useReducer for a single simple value** — a lone boolean or text does not need a rule book; the ceremony slows you down. The fix: `useState` stays the default; promote to a reducer when rules and related values actually appear.

Practice

1. Build a `TodoMini` on useReducer: state is an array of `{ id, text, done }`; actions are `add` (with text), `toggle` (with id, flips done by mapping to a new object), and `clearDone`. Render the list with a checkbox per row dispatching `toggle`.
2. Take the Example 1 counter and add two rules of your own: an action `addFive` that adds 5 but never lets the count pass 20, and a `half` action that halves the count (rounding down). Prove both rules hold by clicking wildly.

Practice quiz

1. What are the three parts of useReducer?

- State, setter, and effect
- The reducer, a pure function (state, action) => newState; actions, small objects like { type: "add", item }; and dispatch, the function components call to send an action in
- Store, slice, and selector
- Props, state, and context

Answer: The reducer, a pure function (state, action) => newState; actions, small objects like { type: "add", item }; and dispatch, the function components call to send an action in

2. What must a reducer do with an action it does not know?

- Return undefined
- Return the state unchanged, the default case is a safe no-op
- Throw an error
- Delete the state

Answer: Return the state unchanged, the default case is a safe no-op

3. When does useReducer earn its keep over useState?

- Always, useState is obsolete

- When several values change together, changes have validation rules, or many event types touch the same state; for one simple value useState stays the honest tool
- Never, useReducer is only for Redux
- Only for counters

Answer: When several values change together, changes have validation rules, or many event types touch the same state; for one simple value useState stays the honest tool

---

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/react-usereducer · More free tools and notes at https://scopetreasure.com