ScopeTreasure Courses

REACT COURSES · LESSON 5.11 · CODE SCOPE SERIES

Redux: State Management

One store for the whole app: why big apps outgrow useContext, the store–action–reducer triangle, dispatch and subscribe in plain words, and when Redux earns its place.

Introduction

Your app now has state living in many components — a user here, a cart there, a theme in Context. In a big app that scattering becomes the problem itself: who owns what, and how do distant cousins share it? **Redux** is the classic answer: **one store** for the app's shared state, changed only through **actions**, by pure **reducers** — the useReducer pattern you just learned, scaled to the whole app, with a strict one-way flow. *(Roman Urdu: Redux poori app ka aik store hai — saari shared state aik jagah, tabdeeli sirf actions se.)*

Key points

- **The triangle: store** (the single state object for the app) → components **dispatch actions** (`{ type: "cart/add", ... }`) → **reducers** compute the new store → subscribed components re-render with fresh values. The flow never runs backwards.
- **One source of truth:** shared state lives in exactly one place, so two screens can never show two different carts again. Debugging becomes "read the store, follow the actions."
- **Why not just useContext?** Context is a delivery pipe, not a state manager: it has no action discipline, no tooling, and every value change re-renders all readers. Redux adds rules, a history of every change (Redux DevTools lets you replay actions), and middleware for async work.
  (Roman Urdu: Context sirf pipe hai; Redux pura nizam hai — qanoon, record, aur tools ke saath.)
- **Actions are plain objects with a `type`:** the same vocabulary as useReducer actions. Action *creators* are small functions that build them, so components never hand-write the objects.
- **Reducers stay pure:** given the same state + action, the same new state — never mutating, never fetching inside. Pure reducers are what make replay and testing possible.
- **Components read slices, not the whole store:** "give me the cart" — and a component re-renders only when the slice it reads actually changes.
- **When Redux earns its place:** big apps, many screens sharing and changing the same data, teams needing discipline. For a small app it is honest overkill — Context or the lighter libraries (5.21) serve better. Modern Redux is written with **Redux Toolkit** (lesson 5.20), which removes the old boilerplate this lesson's concepts used to carry.

Code examples

Example 1 — A tiny store by hand: the pattern without the library

```
// A reducer — exactly like useReducer's
function counterReducer(state = { count: 0 }, action) {
  switch (action.type) {
    case "counter/add":
      return { count: state.count + 1 };
    case "counter/minus":
      return { count: state.count - 1 };
    default:
      return state;
  }
}

// A minimal store: holds state, takes actions, notifies listeners
function createStore(reducer) {
  let state = reducer(undefined, {});
  const listeners = [];
  return {
    getState: () => state,
    dispatch(action) {
      state = reducer(state, action);
      listeners.forEach((fn) => fn());
    },
    subscribe(fn) {
      listeners.push(fn);
    },
  };
}

const store = createStore(counterReducer);
store.subscribe(() => console.log("Store now:", store.getState()));
store.dispatch({ type: "counter/add" });
store.dispatch({ type: "counter/add" });
store.dispatch({ type: "counter/minus" });
```

What it does: This is all Redux really is, stripped of the library: one state, `dispatch` sending actions through a pure reducer, and `subscribe` telling listeners "something changed." React-Redux's job is only to connect this store to components efficiently.

What you will see: the console logs "Store now: { count: 1 }", then "{ count: 2 }", then "{ count: 1 }" — one change per action, in order, fully traceable.

Example 2 — The shape of a real Redux state tree

```
const state = {
  user: { name: "Ayesha", loggedIn: true },
  cart: {
    items: [
      { id: 1, name: "Chai", price: 250, qty: 2 },
      { id: 2, name: "Biscuit", price: 100, qty: 1 },
    ],
  },
  ui: { theme: "dark", sidebarOpen: false },
};

// Each part is owned by its own reducer (a "slice" of the tree):
// userReducer handles user/* actions, cartReducer cart/*, uiReducer ui/*
```

What it does: The whole app's shared state is one tree, split into named slices — user, cart, ui. Each slice has its own reducer that only touches its branch, which is why Redux scales: fifty features means fifty small reducers, not one giant one.

What you will see: nothing visual — read it as a map: every screen in the app reads from this one tree, and every change arrives as an action to exactly one slice.

Example 3 — Dispatch flow: adding to the cart from a product page

```
// The action creator — a function that builds the action
function addToCart(item) {
  return { type: "cart/add", payload: item };
}

// The cart slice's reducer
function cartReducer(state = { items: [] }, action) {
  switch (action.type) {
    case "cart/add":
      return { items: [...state.items, action.payload] };
    default:
      return state;
  }
}

// On the product page, a click is just:
store.dispatch(addToCart({ id: 1, name: "Chai", price: 250, qty: 1 }));
// The navbar's cart badge, subscribed to the cart slice,
// re-renders with the new count — the two components never met.
```

What it does: The product page announces an event; the store routes it to the cart reducer; every component reading the cart (badge, cart page, checkout) updates from the new tree. Distant components cooperate without a single prop passed between them.

What you will see: click "Add" on a product and the navbar badge climbs instantly — the announcement travelled through the store, not through component layers.

Quick recap

- **Redux = one store + actions + pure reducers + subscriptions**, the useReducer pattern at app scale, with a strict one-way flow.
- It beats raw Context for big shared state: discipline, slice-wise updates, and time-travel debugging via action history.
- Honest scope: small apps do not need it; and modern Redux is written with **Redux Toolkit** (5.20), not hand-written boilerplate.

Common mistakes

1. **Putting EVERYTHING in the store** — a single input's text or one card's hover does not belong to the whole app. The fix: the store holds *shared* state; local details stay in component state, or the store becomes a junk drawer.
2. **Mutating state in a reducer** — Redux's change detection relies on new objects; a mutated state means missed renders and broken time-travel. The fix: pure reducers returning fresh copies (Redux Toolkit's Immer makes this feel automatic in 5.20).
3. **Fetching inside reducers** — a reducer that calls an API is no longer pure or replayable. The fix: fetch in components/thunks and dispatch the *result* as an action ("cart/loaded").

Practice

1. Extend Example 1's hand-made store: add a `cartReducer`-style branch is too much — instead add actions `counter/addBy` (with a `payload` number) and `counter/reset`. Dispatch a small sequence and predict each logged state before you run it.
2. Draw your last project's state tree (paper or comments): name three slices, list two actions per slice with their types, and mark which values you would honestly keep OUT of the store as local state — and why.

Practice quiz

1. What is the Redux triangle, in order?

- Components, then store, then actions
- Store (the single state object), components dispatch actions, reducers compute the new store, subscribed components re-render; the flow never runs backwards
- Actions, then store, then components

- Reducers, then actions, then store

Answer: Store (the single state object), components dispatch actions, reducers compute the new store, subscribed components re-render; the flow never runs backwards

2. Why is Redux more than just useContext for big shared state?

- It is not, they are identical
- Context is a delivery pipe with no action discipline, no tooling, and every value change re-renders all readers; Redux adds rules, an action history for time-travel debugging, and middleware for async work
- Context cannot hold objects
- Redux is faster to type

Answer: Context is a delivery pipe with no action discipline, no tooling, and every value change re-renders all readers; Redux adds rules, an action history for time-travel debugging, and middleware for async work

3. May a reducer fetch data from an API?

- Yes, reducers do everything
- No, reducers stay pure: same state plus action gives the same new state, never mutating, never fetching inside, or replay and testing break
- Yes, but only with fetch
- Only in Redux Toolkit

Answer: No, reducers stay pure: same state plus action gives the same new state, never mutating, never fetching inside, or replay and testing break

---

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-redux · More free tools and notes at https://scopetreasure.com