ScopeTreasure Courses

REACT COURSES · LESSON 5.21 · CODE SCOPE SERIES

Zustand

Global state in a few lines: create() builds the store, components pick their slice, two stores side by side, persist across refresh — and when Zustand beats Redux.

Introduction

Redux Toolkit removed Redux's ceremony — **Zustand** asks why there was a ceremony at all. It is a tiny library (the video explains the German name: *Zustand* means "state") where one `create()` call builds a complete global store: state and the functions that change it, living together, with **no Provider, no actions, no reducers** required. *(Roman Urdu: Zustand aik chhoti si library hai — aik create() aur poora global store tayyar, baghair Provider ke.)*

Key points

- **The store is a hook:** `const useCartStore = create((set) => ({ ... }))` — what comes out is a custom hook, and components call it like any hook. The store file *is* the whole setup.
- **State and actions live together:** the object you give `create` holds data (`items: []`) and its own changers (`addItem: (item) => set(...)`) side by side — the capsule every component shares.
- **`set` is the only update verb:** `set((state) => ({ items: [...state.items, item] }))` — it merges your partial update into the store (shallow merge: top-level keys you name are replaced, others untouched).

  (Roman Urdu: set sirf woh hissa badalta hai jo tum batao — baaqi store waisa ka waisa rehta hai.)
- **Selectors keep renders honest:** `useCartStore((s) => s.items.length)` subscribes the component to *that piece only* — an unrelated field changing does not re-render it, the same discipline as `useSelector`, built into the call.
- **No Provider needed:** the store is not Context — it lives outside the component tree, so components anywhere (even plain functions, via `useCartStore.getState()`) can reach it.
- **`persist` middleware survives refresh:** wrap the store and Zustand saves it to localStorage and restores it on load — a cart that remembers, in about two lines.
- **When Zustand beats Redux:** small-to-medium apps, teams that want global state without architecture meetings. Redux Toolkit still wins where strict action history, huge teams and heavy middleware are the point — the video's honest decision table.

Code examples

Example 1 — A complete store in eight lines

```
import { create } from "zustand";

const useCartStore = create((set) => ({
  items: [],
  addItem: (item) => set((state) => ({ items: [...state.items, item] })),
  removeItem: (id) =>
    set((state) => ({ items: state.items.filter((i) => i.id !== id) })),
  clear: () => set({ items: [] }),
}));

export default useCartStore;
```

What it does: Data and its three operations in a single object. `addItem` receives the current state and returns the piece to merge — `set` does the rest. Compare this with the Redux versions: no action types, no separate creators, no switch — the store explains itself.

What you will see: nothing visual — this file is the entire state layer for the shop below, in fewer lines than a useReducer version of the same cart.

Example 2 — Components pick their slice

```
import useCartStore from "./cartStore";

function CartBadge() {
  // Re-renders ONLY when the count changes
  const count = useCartStore((s) => s.items.length);
  return <span className="badge">{count}</span>;
}

function AddButton({ product }) {
  // Subscribing to a function: its identity never changes,
  // so this component effectively never re-renders for the store
  const addItem = useCartStore((s) => s.addItem);
  return <button onClick={() => addItem(product)}>Add {product.name}</button>;
}

function CartTotal() {
  const items = useCartStore((s) => s.items);
  const total = items.reduce((sum, i) => sum + i.price, 0);
  return <p>Total: Rs {total}</p>;
}
```

What it does: Three components, three precise subscriptions — a number, a function, the list. No Provider wraps them, no props pass between them; each simply calls the store hook with a selector naming its piece. The badge re-renders on count changes only; the total derives from the list during render as always.

What you will see: add an item anywhere in the app and the badge and total update together, while components subscribed to unrelated pieces stay perfectly still.

Example 3 — persist: the cart survives a refresh

```
import { create } from "zustand";
import { persist } from "zustand/middleware";

const useCartStore = create(
  persist(
    (set) => ({
      items: [],
      addItem: (item) => set((state) => ({ items: [...state.items, item] })),
      clear: () => set({ items: [] }),
    }),
    { name: "cart-storage" }
  )
);

export default useCartStore;
```

What it does: The `persist` middleware wraps the same store definition and mirrors the data slice into `localStorage` under the name "cart-storage" on every change, restoring it when the app loads. Functions are not stored (they cannot be) — on reload Zustand rebuilds them from your definition and rehydrates the data.

What you will see: add two items, refresh the page — the cart still holds both, badge and total intact, exactly the behaviour lesson 5.9's useLocalStorage gave one component, now app-wide.

Quick recap

- **`create()` builds a store-hook:** state + changer functions in one object, updated through **`set`** (shallow merge), reached without any Provider.
- **Selectors** — `useStore((s) => s.piece)` — keep each component subscribed to exactly its slice.
- **`persist`** saves the store across refreshes in two lines; Zustand is the default answer until an app genuinely needs Redux's discipline and tooling.

Common mistakes

1. **Selecting a fresh object every time** — `useStore((s) => ({ a: s.a, b: s.b }))` builds a new object on each call, so the component re-renders on *every* store change. The fix: select primitives or single

fields, one call each (or use the `shallow` comparison helper when an object is unavoidable).

2. **Mutating inside set** — `set((s) => { s.items.push(x); return s; })` returns the same references and breaks the merge contract. The fix: return a partial with new arrays/objects — `{ items: [...s.items, x] }`.
3. **Making a store per component file** — five mini-stores that secretly need each other's data recreate the scattering Zustand exists to cure. The fix: one store per *domain* (cart, user, ui), imported where needed.

Practice

1. Build a `useThemeStore` with `theme` and a `toggle` function, then a second `useCounterStore` — prove in one small app that the two stores live side by side without a Provider and that changing one never re-renders components subscribed only to the other (show it with console logs).
2. Take the Example 1 cart and add `persist`. Then open your browser's devtools → Application → Local Storage and find the saved entry: note exactly what was stored (data only) and what was rebuilt (the functions) after a refresh.

Practice quiz

1. What does const useCartStore = create((set) => ({ ... })) give you?

- A context Provider
- A store-hook: the store file is the whole setup, components call it like any hook, and no Provider is needed
- A component
- A reducer

Answer: A store-hook: the store file is the whole setup, components call it like any hook, and no Provider is needed

2. What does set do in Zustand, and how do selectors keep renders honest?

- set replaces the whole store, selectors do nothing
- set shallow-merges your partial update, and a selector like useCartStore((s) => s.items.length) subscribes the component to that piece only
- set deletes the store, selectors rebuild it
- set only works once

Answer: set shallow-merges your partial update, and a selector like useCartStore((s) => s.items.length) subscribes the component to that piece only

3. How does a Zustand store survive a page refresh?

- It does not, stores are always lost
- With the persist middleware, which saves it to localStorage and restores it on load, in about two lines
- With a cookie, automatically
- By copying it into the URL

Answer: With the persist middleware, which saves it to localStorage and restores it on load, in about two lines

---

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