# Zustand

Version 5.0 · verified 2026-08-11

A store that is just a hook — selectors and re-renders, set semantics, middleware, the slices pattern, and per-request stores for SSR.

## Contents

## Mental model

A Zustand store is a hook you create, not a provider you mount. `create` returns that hook: call it with a selector and the component subscribes to exactly that value, or call `getState` on the same object and read the store from anywhere, React or not. Actions are ordinary functions that call `set` — there are no action objects, no reducers and no context. The whole library is that one idea, plus middleware.

## Creating a store

| Piece | What it is |
|---|---|
| `create(fn)` | Returns a bound hook |
| `set` | Merges a partial state, or an updater |
| `get` | Reads current state inside an action |
| Actions | Plain functions, stored beside the data |

Zustand describes itself as “a small, fast and scalable bearbones state-management solution using simplified flux principles” with an API “based on hooks”. The state creator receives `set`, `get` and the store, and whatever object you return **is** the state — so the convention is to keep actions in it, next to the values they change, making a feature one object. The returned hook also carries `getState`, `setState`, `getInitialState` and `subscribe`.

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

```

```
export const useBears = create((set, get) => ({
  bears: 0,
  add: () => set((s) => ({ bears: s.bears + 1 })),
  reset: () => set({ bears: 0 }),
  double: () => set({ bears: get().bears * 2 }),
}));
```

> **Tip:** TypeScript needs the curried form, `create<State>()(fn)`. The single call cannot infer the state, because the generic is invariant — it is both produced and consumed by the creator — so TypeScript gives up and infers `unknown`.

## Selecting state

| Call | Subscribes to |
|---|---|
| `useBears()` | The whole store — every change |
| `useBears(s => s.bears)` | One value, compared with `Object.is` |
| `useBears(s => s.add)` | An action; its identity never changes |

The selector re-runs after every store update and its result is compared with `Object.is`; the component re-renders only when that comparison fails. So the unit of subscription is the selector, not the store: select one value per call and unrelated fields can churn all day without touching your component. Actions are stable references, so selecting them is free.

```
function Counter() {
  const bears = useBears((s) => s.bears);
  const add = useBears((s) => s.add);
  return <button onClick={add}>{bears}</button>;
}
```

> **Gotcha:** `useBears()` with no selector subscribes to the entire store, so every unrelated field re-renders the component. It is the most common Zustand performance bug, and it looks like working code.

## useShallow

| Selector returns | Result |
|---|---|
| A primitive | Renders only when it changes |
| A fresh object or array each run | Renders on every store change |
| The same wrapped in `useShallow` | Renders only if shallow-unequal |

“The computed selector will cause a rerender if the output has changed according to `Object.is`” — and `Object.keys(state)`, a `.filter()` or a grouped `{ a, b }` object is a new reference every single run. `useShallow` from `zustand/react/shallow` memoizes the selector using a shallow comparison, which is the fix for both grouped selections and derived arrays. It compares one level deep only; nested objects are still compared by reference.

```
import { useShallow } from "zustand/react/shallow";

// Re-renders only when one of the two changes.
const { bears, fishes } = useBears(
  useShallow((s) => ({
    bears: s.bears,
    fishes: s.fishes,
  })),
);
```

> **Gotcha:** v5's `create` "does not support customizing equality function" — the old second argument to the hook is gone. Use `useShallow`, or `createWithEqualityFn` from `zustand/traditional` if you need the v4 behaviour back.

## Updating state

| Call | Effect |
|---|---|
| `set({ a: 1 })` | Shallow-merges into the root |
| `set(s => ({ n: s.n + 1 }))` | Updater form, from current state |
| `set(next, true)` | Replaces the whole state object |
| Nested value | Spread each level, or use `immer` |

`set` takes a partial state "and it will be shallowly merged with the existing state in the store" — merged at the **root only**. Anything deeper has to be rebuilt by hand, which is where the spread pyramid comes

from, and why the docs point at Immer for nested shapes. The updater form is the one to reach for whenever the next value depends on the current one.

```
// Shallow merge: other root keys are untouched.
set({ bears: 3 });

// Nested: every level must be copied.
set((s) => ({
  deep: {
    ...s.deep,
    nested: { ...s.deep.nested, count: 1 },
  },
}));
```

> **Gotcha:** `set(next, true)` replaces the state object outright — and your actions live in that object, so anything you omit is gone. The store keeps working until the first click calls a function that no longer exists.

## Middleware

| Middleware | Gives you |
|---|---|
| `persist` | Save and rehydrate from storage |
| `devtools` | The Redux DevTools timeline |
| `immer` | Mutating syntax for nested updates |

| Middleware | Gives you |
| --- | --- |
| `subscribeWithSelector` | `subscribe(selector, listener)` |
| `combine` | Inferred types without writing generics |

Middleware wrap the state creator, so they compose by nesting and the order matters — `devtools(persist(immer(fn)))` is the usual stack. `persist` defaults to `createJSONStorage(() => localStorage)`, evaluated lazily so it does not explode during server rendering. `partialize` chooses what gets written, `version` plus `migrate` handle a changed shape, and `skipHydration` defers rehydration until you call it.

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

export const useAuth = create(
  persist(
    (set) => ({
      token: null,
      clear: () => set({ token: null }),
    }),
    {
      name: "auth", // storage key
      partialize: (s) => ({ token: s.token }),
      version: 1,
    },
```

```
  ),
);
```

> **Warning:** Persisted state is trusted blindly. `createJSONStorage` "does not perform any runtime validation", so the stored value is cast to your state type — corrupt, stale or tampered data walks straight into the store. Validate it in `migrate` or a custom storage.

## Outside React

| API | Use it for |
|---|---|
| `useStore.getState()` | Read once, without subscribing |
| `useStore.setState(p)` | Write from anywhere |
| `useStore.subscribe(cb)` | React to changes outside React |
| `createStore` (vanilla) | A store with no React at all |

Because the hook *is* the store, event handlers, tests, socket clients and plain modules can read and write without a component. `subscribe` registers "a callback that fires whenever the store's state updates", which is the basis of transient updates: let a high-frequency value — pointer position, scroll offset — drive a ref or the DOM directly and never re-render. `zustand/vanilla`'s `createStore` gives the same store with no React dependency.

```
// Read and write outside a component.
const { bears } = useBears.getState();
useBears.setState({ bears: bears + 1 });

// Transient: a side effect, not a render.
const unsub = useBears.subscribe((s) => {
  ref.current.textContent = String(s.bears);
});
```

> **Gotcha:** `getState()` is a snapshot, not a subscription. Reading it during render gives a value that is correct once and then never updates again — the component has told React nothing about what it depends on.

## Slices pattern

| Step | Shape |
|---|---|
| Write a slice | `(set, get) => ({ … })` |
| Combine them | `create((...a) => ({ ...slice(...a) }))` |
| Cross-slice action | Read or set another slice's key |

"You can divide your main store into smaller individual stores to achieve modularity." A slice is just a fragment of a state creator, and the bound store spreads them with `(...a)` so each one receives the same `set`, `get` and store instance. They all land in a single flat state object,

which is exactly why a bear slice can decrement `fishes` — the modularity is in your files, not in the runtime state.

```
export const createFishSlice = (set) => ({
  fishes: 0,
  addFish: () =>
    set((s) => ({ fishes: s.fishes + 1 })),
});

export const useBoundStore = create((...a) => ({
  ...createBearSlice(...a),
  ...createFishSlice(...a),
}));
```

> **Gotcha:** Slices share one namespace. Two of them declaring `loading` or `error` silently overwrite each other at spread time, and the winner is whichever is spread last — prefix the keys, or nest each slice under its own object.

## SSR and per-request stores

| Rule | Why |
|---|---|
| No module-level store on a server | It is shared across requests |
| Create the store per request | Concurrent users, separate state |
| Hand it down through context | The hook cannot be a global |
| Server Components never touch it | They have no hooks or state |

On a server the module-scope store that makes Zustand pleasant becomes a leak: “the store should be created per request and should not be shared across requests”, and “React Server Components should not read from or write to the store”. The pattern is a client provider that builds a vanilla store once into a ref and exposes it through context, with `useStore` reading it. Server and client must start from the same state or hydration breaks; with `persist`, that means `skipHydration` and rehydrating in an effect.

```
"use client";
// createStore from "zustand/vanilla",
// useStore from "zustand"
const Ctx = createContext(null);

export function StoreProvider({ children }) {
  const ref = useRef(null);
  if (!ref.current) ref.current = makeStore();
  return (
    <Ctx value={ref.current}>{children}</Ctx>
  );
}
export const useCounter = (sel) =>
  useStore(useContext(Ctx), sel);
```

> **Gotcha:** A `create()` call at module scope on the server is shared by every request that instance handles, so one user's state can show up in another user's HTML. It is invisible in development, where you are the only request.

## Choosing Zustand

| What you need | Reach for |
|---|---|
| Small shared client state | Zustand |
| Enforced structure, big team | Redux Toolkit |
| Server data and caching | RTK Query, React Query |
| Rarely-changing config | Context |

Zustand keeps the useful half of flux — a single store, explicit updates, a DevTools timeline — and drops providers, action constants and reducer indirection. What you give up is enforcement: any module can call `setState`, so discipline lives in your conventions rather than in the library. That trade is excellent for a focused store and gets worse as the number of people writing to it grows.

```
useState      one component
Zustand       shared client state, little ceremony
Redux Toolkit shared client state, enforced shape
RTK Query     anything the server owns
```

> **Tip:** Keep actions inside the store rather than calling `setState` from components. The docs describe both styles, but colocating them keeps one searchable list of every way the state can change — which is the property that makes a store debuggable at all.

## Further reading

- Zustand documentation
- create API
- Updating state
- useShallow
- persist middleware
- Slices pattern
- Setup with Next.js
- Migrating to v5