React State Management

Version 19.2 · verified 2026-10-24

Classify state before choosing a tool — local, Context, server cache, global store, atoms — and which approach fits which project.

Contents

Mental model 2

Kinds of state 2

The local-first ladder 3

Context: share, don't store 4

Server state: the biggest win 5

Client global stores 7

Atomic state: Jotai 8

URL and form state 9

Choosing by project 10

Further reading 11

Mental model

Most of what people call “state” is really a **cache of data that lives on a server** — treat it as a cache you borrow, not something you own. The rest is genuine client state, and nearly all of it is local to one component or a small subtree; only a little is truly global. Classify each piece first, then reach for the smallest tool that fits. A global store is the last resort, not the default.

Kinds of state

Kind	Lives in	Reach for
Server cache	API / database	TanStack Query
Local UI	one component	`useState`
Shared UI	a subtree	lift up, Context
Global client	whole app	Zustand, Redux
Form	the inputs	`useState`, RHF
URL	the address bar	the router

Classify by where the source of truth lives and who needs to read it. The most common mistake is storing fetched server data in a client store and syncing it by hand — that is a cache pretending to be state. Pulling server data out into a query library and keeping URL state in the URL removes most of what teams think of as “state management” before any store is chosen.

> **Gotcha:** Two copies of the same fact drift apart. If a value can be computed from props, existing state, or the URL, derive it during render instead of storing a second copy.

The local-first ladder

Need	Use
One independent value	`useState`
Next value from previous	updater function
Many fields move together	`useReducer`
A child needs parent state	lift up
Reset state on prop change	`key`

Start local. Lift state only to the **closest** common ancestor that needs it — no higher. Move from `useState` to `useReducer` when several fields change together or many handlers touch the same state. These built-ins carry most apps; a library is not a starting point.

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

function Child({ n }: { n: number }) {
  return <p>{n}</p>;
}

export function Parent() {
  const [n, setN] = useState(0);
```

```
  return (
    <>
      <button onClick={() => setN(n + 1)}>+</button>
      <Child n={n} />
    </>
  );
}
```

> **Gotcha:** Lifting too high is its own bug. State hoisted to the root re-renders the whole tree on every change — keep it at the lowest owner that still covers every reader.

Context: share, don't store

Piece	Role
`createContext(default)`	defines the channel
`<Ctx value={v}>`	provides to the subtree
`useContext(Ctx)`	reads nearest provider

Context solves **prop-drilling** — passing a value deep without threading it through every layer. It is transport, not a store: no selectors, no middleware, and every consumer re-renders when the value changes by `Object.is`. Pair it with `useReducer` for a small app-wide store; for large or fast-changing global state, a store library re-renders far less.

```tsx
import { createContext, useReducer } from "react";

type S = { count: number };
type A = { type: "inc" };
const Ctx = createContext<S | null>(null);
const step = (s: S, _: A): S =>
  ({ count: s.count + 1 });

export function Provider(p: { children: unknown }) {
  const [s] = useReducer(step, { count: 0 });
  return <Ctx value={s}>{p.children as any}</Ctx>;
}
```

> **Gotcha:** A fresh object in `value` each render re-renders every consumer every time. Memoize the value, or split a slow read context from a stable write context.

Server state: the biggest win

Concern	A query library gives you
Caching	a keyed cache
Staleness	stale-while-revalidate
Status	loading / error flags
Deduping	one request per key
Freshness	background refetch

Server data is asynchronous, shared, and goes stale — properties no reducer models well. A query library (TanStack Query, or RTK Query if you already run Redux) keys each request, caches the result, dedupes concurrent callers, and refetches in the background. Adopting one usually deletes most of a global store and nearly every data-fetching Effect.

```
import { useQuery } from "@tanstack/react-query";
type User = { name: string };

export function Profile({ id }: { id: string }) {
  const q = useQuery({
    queryKey: ["user", id],
    queryFn: (): Promise<User> =>
      fetch(`/api/users/${id}`)
        .then((r) => r.json()),
  });
  if (q.isPending) return <p>Loading…</p>;
  return <p>{q.data?.name}</p>;
}
```

> **Tip:** Don’t copy query data into a client store. Read it from the cache wherever you need it — the cache is the single source of truth, and copying it reintroduces the sync bug you removed.

Client global stores

Library	Shape	Fits
Redux Toolkit	one store, slices	big apps, strict flow
Zustand	many hook stores	minimal boilerplate

Reach for a global client store only for state that is genuinely app-wide and shared by distant components: session and auth, theme, a shopping cart, cross-cutting UI. Redux Toolkit brings structure, time-travel devtools, and middleware at the cost of ceremony; Zustand is a store that is just a hook — small and unopinionated. Both are solid; choose by team size and how much structure you want enforced.

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

type Store = { count: number; inc: () => void };

const useStore = create<Store>()((set) => ({
  count: 0,
  inc: () => set((s) => ({ count: s.count + 1 })),
}));

export function Counter() {
  const n = useStore((s) => s.count);
  return <p>{n}</p>;
}
```

> **Gotcha:** Subscribing to the whole store re-renders on every change. Select the slice you read (`s => s.count`), not the entire state object.

Atomic state: Jotai

Concept	Meaning
`atom(init)`	the smallest unit of state
derived `atom`	computed from other atoms
`useAtom`	read and write, like `useState`

Atomic libraries (Jotai, Recoil-style) build state **bottom-up** from tiny atoms instead of one large store. A component subscribes to only the atoms it reads, so updates are fine-grained with no selector boilerplate, and derived atoms recompute on their own. Good when state is highly interrelated or you want Context-like ergonomics without the re-render cost.

```
import { atom, useAtom } from "jotai";
const countAtom = atom(0);
const doubleAtom = atom(
  (get) => get(countAtom) * 2,
);

export function Counter() {
  const [n, setN] = useAtom(countAtom);
  const [double] = useAtom(doubleAtom);
```

```
  return (
    <button onClick={() => setN(n + 1)}>
      {n} / {double}
    </button>
  );
}
```

> **Note:** Atoms are identified by reference, not by a string key. Define each as a module-level `const` — never create one during render, or it resets every time.

URL and form state

State	Put it in
Filters, tab, page	the URL
Current route	the router
Input values	form state / RHF

Two kinds of state get forced into stores where they don't belong. Anything a user should be able to bookmark, share, or reload into — the active tab, filters, pagination, a search query — belongs in the **URL**, owned by the router. Form input is ephemeral and local: `useState` handles small forms, and a form library (React Hook Form) handles large ones with validation and fewer re-renders.

```tsx
import { useState } from "react";

// Ephemeral and local — no global store.
export function Search() {
  const [q, setQ] = useState("");
  return (
    <input
      value={q}
      onChange={(e) => setQ(e.target.value)}
    />
  );
}
```

> **Gotcha:** Mirroring URL state into a store and syncing both means two sources of truth that disagree on back/forward. Read from the URL and derive; don't duplicate it.

Choosing by project

Project	Likely needs	Start with
Static / marketing	almost none	`useState`
Dashboard / CRUD	server cache	Query + `useState`
Large SPA, teams	structured global	Redux + Query
Editor / tool	fine-grained client	Zustand/Jotai + Query
Next.js app	server + a little	RSC + Query + store

Decide from the state you actually have, not the app's ambition. Nearly every data-driven app wants a server-cache library first; add a client store only when distant components share client state. Most apps never need Redux. Co-locate state as low as it goes and promote it only when a second component truly needs it.

> **Tip:** Add a library when you feel the pain it removes — prop-drilling, manual cache sync, tangled Effects — not at project start. Adding one later is cheap; unwinding a premature one is not.

Further reading

- Managing State
- Choosing the State Structure
- Sharing State Between Components
- Scaling Up with Reducer and Context
- You Might Not Need an Effect
- TanStack Query overview
- Redux — when (not) to use it