Jotai

Version 2.20 · verified 2026-08-25

Atomic state for React — atoms and derived atoms, write atoms as actions, async and Suspense, stores and Providers, persistence and SSR.

Contents

Mental model 2

Creating atoms 2

Using atoms in components 4

Derived atoms 6

Write atoms as actions 7

Async atoms and Suspense 9

Stores and Provider 11

Utilities 13

SSR and Next.js 16

Choosing Jotai 17

Further reading 19

Mental model

An atom is a definition, not a value. `atom(0)` returns an immutable config object that holds no state; the value lives in a store, keyed by that object's identity. Components subscribe to individual atoms, so a render is triggered by the atoms you actually read, not by a slice of one big object. Atoms derive from other atoms into a graph, and Jotai recomputes only the part of that graph a change touches.

Creating atoms

Call	Creates
`atom(0)`	Primitive, writable
`atom(get => …)`	Read-only derived
`atom(read, write)`	Read-write derived
`atom(null, write)`	Write-only action
`a.debugLabel = "count"`	A name for debugging
`a.onMount = fn`	First-subscription hook

“An atom config is an immutable object. The atom config object doesn’t hold a value. The atom value exists in a store.” That is why atoms are normally created once at module scope and exported: the config *is* the key, so two `atom(0)` calls are two unrelated pieces of state. Deriving is the same function — pass a read function instead of a value.

```
import { atom } from "jotai";

// A definition, not a value: the 0 lives in
// a store, keyed by this object.
export const countAtom = atom(0);

// Recomputed when countAtom changes.
export const doubledAtom = atom(
  (get) => get(countAtom) * 2,
);
```

> **Gotcha:** `useAtom(atom(0))` loops forever. "Referential equality is important" — an inline `atom()` is a new key on every render, so the component reads fresh state, re-renders, and does it again. Create atoms at module scope, or wrap them in `useMemo`.

Using atoms in components

Hook	Returns	Subscribes
`useAtom(a)`	`[value, setValue]`	Yes
`useAtomValue(a)`	`value`	Yes
`useSetAtom(a)`	`setValue`	No

`useAtom` mirrors `useState` and returns a tuple. `setValue` "takes just one argument, which will be passed to the write function of the atom as the third parameter" — for a primitive atom that is the next value or an updater. The subscription unit is the atom, so a

component re-renders only when an atom it reads changes; nothing else in the app matters.

```
function Counter() {
  const [count, setCount] = useAtom(countAtom);
  const doubled = useAtomValue(doubledAtom);
  const inc = () => setCount((c) => c + 1);
  return (
    <button onClick={inc}>
      {count} / {doubled}
    </button>
  );
}
```

> **Gotcha:** `const [, setCount] = useAtom(countAtom)` still subscribes. The component re-renders on every update of a value it never reads — the docs call these "unnecessary rerenders". `useSetAtom` is the fix.

Derived atoms

Piece	Behavior
`get(a)` in `read`	Reads `a` and tracks it
Dependency set	Recollected on every run
`atom(read)`	Read-only — no setter
`atom(read, write)`	Derived and writable

“`get` in the read function is to read the atom value. It’s reactive and read dependencies are tracked.” Dependencies are discovered by running the function, never declared: on first read Jotai records which atoms were touched, and “every time we invoke the ‘read’ function, we refresh the dependencies and dependents.” Derived atoms are cached and shared — ten components reading one derived atom compute it once.

```
const filterAtom = atom("all");
const todosAtom = atom([]);
const doneAtom = atom([]);
```

```
// Tracks doneAtom only while the filter is
// "done"; otherwise it tracks todosAtom.
const visibleAtom = atom((get) =>
  get(filterAtom) === "done"
    ? get(doneAtom)
    : get(todosAtom),
);
```

> **Gotcha:** A branch that skips a `get` drops that dependency for as long as the branch is not taken, so the atom stops reacting to it. That is correct behavior, and it surprises everyone who expects the dependency list to be fixed at creation time.

Write atoms as actions

Piece	Behavior
`atom(null, write)`	Write-only; `null` by convention

Piece	Behavior
`set(a, v)`	Writes, invoking `a`'s write
`set(a, (p) => …)`	Updater form
`get` inside `write`	Reads, but is *not* tracked
`write(get, set, ...args)`	Takes any number of args

A write function is where multi-atom updates and business rules live: it can read anything, set as many atoms as it wants, be async, and it receives whatever arguments the caller passed. Passing `null` as the read argument gives an atom with no value of its own — an action you invoke with `useSetAtom`, which is Jotai's answer to a reducer.

```
const countAtom = atom(0);


// An action atom: no value, just behavior.
const addAtom = atom(null, (get, set, by) => {
```

```
  if (get(countAtom) + by < 0) return;
  set(countAtom, (c) => c + by);
});

// const add = useSetAtom(addAtom);
// add(5);
```

> **Gotcha:** `get` in a write function "is also to read atom value, but it's not tracked". It samples the value at call time and creates no dependency — which is exactly why an action atom never re-renders the component holding it.

Async atoms and Suspense

Piece	Behavior
`atom(async (get) => …)`	Suspends the reader
`loadable(a)`	Three states as data
`unwrap(a, fallback)`	Sync value, no suspend
`{ signal }` read option	Aborts a stale run

“Jotai is inherently leveraging `Suspense` to handle asynchronous flows”: a read function that returns a promise suspends every component reading it, and those components see the resolved value, not the promise. The read function’s second argument carries an `AbortSignal` that fires before a new run starts, so a stale fetch cancels itself. Wrap the atom in `loadable` to read `loading`, `hasData` and `hasError` as a plain value instead of a boundary; `unwrap` does the same with a fallback, but still throws on error.

```
const userIdAtom = atom(1);
const userAtom = atom(async (get, opts) => {
  const id = get(userIdAtom);
  const res = await fetch(`/api/users/${id}`, {
    signal: opts.signal,
  });
  return res.json();
});
```

```
// useAtomValue(userAtom) returns the user;
// the reader must sit under a
// <Suspense fallback="…"> boundary.
```

> **Gotcha:** An atom deriving from an async atom gets a promise, not a value — `get` in a read function does not resolve it. The derived atom has to `await get(userAtom)`, and so becomes async itself; async propagates up the graph until something suspends.

Stores and Provider

API	Use it for
`getDefaultStore()`	The provider-less global store
`createStore()`	A fresh, isolated store
`store.get(a)`, `store.set(a, v)`	Read and write outside React

API	Use it for
`store.sub(a, cb)`	Subscribe; returns unsubscribe
`<Provider store={s}>`	Scope a subtree to `s`
`useStore()`	The store in scope

With no `Provider`, every atom resolves against one module-global default store — the mode most apps start in. A `Provider` holds its own store for its subtree, which is how you get “a different state for each sub tree”, accept initial values, or “clear all atoms by remounting”. The store’s three methods work with no React involved, so timers, socket handlers and tests can read and write the same atoms.

```
const store = createStore(); // from "jotai"


store.set(countAtom, 1);
const unsub = store.sub(countAtom, () => {
  console.log("now", store.get(countAtom));
```

```
});

const Root = () => (
  <Provider store={store}>
    <App />
  </Provider>
);
```

> **Gotcha:** One atom config under two Providers is two independent values, and a component rendered outside the Provider silently reads the default store instead. The symptom is an update that “doesn’t arrive” in a component that looks like it is subscribed.

Utilities

From `jotai/utils`	Gives you
`atomWithStorage(key, v)`	A `localStorage`-backed atom

From `jotai/utils`	Gives you
`createJSONStorage(fn)`	Custom or guarded storage
`atomWithReset` and `RESET`	A value you can reset
`selectAtom(a, sel)`	A narrowed, compared slice
`splitAtom(listAtom)`	One writable atom per item
`atomFamily(fn)`	One atom per parameter

`atomWithStorage` persists through `JSON.stringify`, and its default storage subscribes to `storage` events, so two tabs stay in sync for free. `getOnInit` is `false` by default: the first render gets `initialValue` and the stored value arrives after — set it to `true` when the stored value must be there on the first paint. `selectAtom` needs “both the base atom and the selector to be stable”, so define both outside the component or memoize them.

```
import {
  atomWithStorage,
  createJSONStorage,
} from "jotai/utils";

const themeAtom = atomWithStorage(
  "theme",
  "light",
  createJSONStorage(() => localStorage),
  { getOnInit: true },
);
```

> **Warning:** “Internally, atomFamily is just a Map whose key is a param and whose value is an atom config. Unless you explicitly remove unused params, this leads to memory leaks.” Call `remove(param)` or `setShouldRemove`. It is also deprecated — `jotai-family` is the drop-in replacement, and v3 removes the built-in.

SSR and Next.js

Rule	Why
Wrap the app in `<Provider>`	The default store outlives a request
`useHydrateAtoms(values)`	Seeds atoms from server data
Guard storage access	No `localStorage` on the server
No promises during SSR	Async atoms never resolve there

Provider-less mode is a server-side bug: the global store “is kept alive and is shared between multiple requests, which can lead to bugs and security risks” — one user’s state can surface in another’s HTML. A `Provider` at the root scopes the store to the app instance, which the server recreates per request. `useHydrateAtoms` is a client hook, so the component needs `"use client"` in the App Router.

```
"use client";
// useHydrateAtoms from "jotai/utils"

function Counter({ initial }) {
  useHydrateAtoms([[countAtom, initial]]);
  const [count] = useAtom(countAtom);
  return <span>{count}</span>;
}
```

> **Gotcha:** “Atoms can only be hydrated once per store.” A later render with a different prop is ignored, so a value that changes between navigations needs a store per page, not a second hydrate.

Choosing Jotai

What you need	Reach for
A `useState` + `useContext` replacement	Jotai

What you need	Reach for
A simple module-level store	Zustand
Enforced structure and an action log	Redux Toolkit
Suspense-driven async state	Jotai

“Jotai was born to solve extra re-render issues in React.” Context forces a choice between one big value that re-renders everything and a tower of providers; Jotai replaces both with atoms, and “optimize[s] renders based on atom dependency”, which “avoids the need for memoization”. The cost is that state is spread across many small configs: there is no single object to inspect, and no dispatch log to replay. Jotai is bottom-up and context-first; Zustand is one top-down store, module-first.

```
useState       one component
Jotai          many pieces, derived graph
Zustand        one module store, selectors
Redux Toolkit  enforced shape, action log
```

> **Tip:** Set `debugLabel` on atoms you expect to debug. Atoms are identified by object identity, not by a name, so without a label devtools and warnings have nothing readable to show you.

Further reading

- Jotai documentation
- atom API
- useAtom, useAtomValue, useSetAtom
- Store API
- Provider and useStore
- Async utilities
- atomWithStorage
- useHydrateAtoms for SSR
- Using Jotai with Next.js
- Comparison with other libraries