Redux

Version 2.12 · verified 2026-08-11

The one-way data flow and why it scales, then practical Redux Toolkit — store, slices, selectors, thunks and RTK Query in a React app.

Contents

Mental model

Redux holds shared state in one immutable tree that can only change by dispatching a plain-object action to a pure reducer. That single constraint buys a named, ordered, replayable history of every change, which is what keeps a large app debuggable. Redux Toolkit is how you write this today — hand-written action types, `switch` reducers and manual spreading are legacy boilerplate, and RTK deletes them without changing the model underneath.

The one-way data flow

Step	What happens
`dispatch(action)`	A plain object names what happened
Reducer	`(state, action) => nextState`, pure
Store	Holds the tree, notifies subscribers

Step	What happens
`useSelector`	Reads a value, re-renders if it changed

The three principles are the whole architecture: state lives “in an object tree within a single store”, “the only way to change the state is to emit an action”, and the transformation is written as pure reducers. Because every change is centralized and ordered, “there are no subtle race conditions to watch out for”, and since actions are plain objects they “can be logged, serialized, stored, and later replayed” — which is what makes DevTools time-travel possible at all.

```
UI event
  -> dispatch(action)
     -> reducer(state, action) -> next state
        -> store notifies subscribers
           -> each selector re-runs
              -> only changed ones re-render
```

> **Gotcha:** A reducer must be pure — no `fetch`, no `Date.now()`, no `Math.random()`, no mutation of anything outside its own draft. Break that and replaying the same actions stops producing the same state, which costs you every debugging benefit you adopted Redux for.

Why it scales

Concern	How Redux answers it
Who changed this?	One named action, in the log
Re-render blast radius	Per-component selector subscription
Prop drilling	Any component reads the store directly
Reproducing a bug	Replay the recorded action list

The performance argument is the subscription model. Every `useSelector` is its own subscription: after a

dispatch each selector re-runs and its result is compared by reference, so only components whose selected value actually changed re-render. Context works the other way — every consumer re-renders whenever the provider value changes, so a single shared context object makes unrelated updates expensive. Redux also keeps update logic out of components entirely, which is why a growing codebase does not turn into deeper prop chains.

```
Context:  provider value changes
          -> every consumer re-renders

Redux:    action dispatched
          -> selectors re-run (cheap)
          -> only changed results re-render
```

> **Note:** The re-render win depends on selecting narrowly. A selector that returns the whole slice re-renders its component on every change to that slice, which is exactly the behaviour you were avoiding.

Store setup

`configureStore` gives you	Instead of
Combined slice reducers	`combineReducers` by hand
`redux-thunk` preinstalled	`applyMiddleware` wiring
DevTools connected	`composeWithDevTools`
Mutation + serializability checks	Silent bugs in development

One call replaces the entire legacy setup: it “calls `combineReducers` to merge slice reducers”, adds the thunk middleware, and connects the DevTools extension. The development-only middleware is the part people underrate — it catches accidental mutation and non-serializable values such as `Date` s, class

instances or promises before they reach the store. Wrap the app once in a `<Provider>`.

```
import { configureStore } from "@reduxjs/toolkit";
import { Provider } from "react-redux";
import todos from "./todosSlice";

export const store = configureStore({
  reducer: { todos },
});

// Once, at the root of the app:
// <Provider store={store}><App /></Provider>
```

> **Tip:** Derive your types from the store rather than writing them: `type RootState = ReturnType<typeof store.getState>` and `type AppDispatch = typeof store.dispatch`. Export typed `useAppSelector` / `useAppDispatch` hooks once and every read is inferred.

Slices

`createSlice` field	Produces
`name`	The action-type prefix, e.g. `todos/`
`initialState`	The slice's starting value
`reducers`	Case reducers **and** their action creators
`extraReducers`	Handlers for actions defined elsewhere

`createSlice` "automatically generates action creators and action types that correspond to the reducers", so writing `added` gives you `slice.actions.added()` dispatching type `todos/added`. Immer backs the reducers, which is why "mutating" code is allowed and still produces an immutable update. One rule: never mix mutating the draft and returning a new value in the same reducer.

```
import { createSlice } from "@reduxjs/toolkit";

const slice = createSlice({
  name: "todos",
  initialState: { items: [], loading: false },
  reducers: {
    added(state, action) {
      state.items.push(action.payload); // Immer
    },
    cleared: (state) => { state.items = []; },
  },
});
export const { added, cleared } = slice.actions;
export default slice.reducer;
```

> **Gotcha:** The Immer draft only exists inside `createSlice` and `createReducer`. The same `state.items.push(x)` written in a component, a selector or a plain reducer mutates real state, and because the reference never changes, nothing re-renders.

Reading state

Selector returns	Result on each dispatch
A primitive	Re-render only when it changes
An existing reference	Stable, no re-render
A new object or array	Re-renders every single time

`useSelector` re-runs after every dispatched action and "uses strict `===` reference equality checks by default, not shallow equality". So a selector that builds a fresh object is a guaranteed re-render: "returning a new object every time will *always* force a re-render". The

three fixes, in order of preference: call `useSelector` once per value, memoize with `createSelector`, or pass `shallowEqual` as the second argument.

```
// Bad: a new object each run, so it always
renders.
const view = useSelector((s) => ({
  n: s.todos.items.length,
  busy: s.todos.loading,
}));

// Good: one value per call, compared by ===.
const n = useSelector((s) =>
s.todos.items.length);
const busy = useSelector((s) => s.todos.loading);
```

> **Gotcha:** `.filter()`, `.map()` and `.slice()` inside a selector hit the same trap without an object literal in sight — each call returns a brand-new array. Derived lists need `createSelector`, not a plain arrow function.

Dispatch and async thunks

What you are doing	Where it goes
A synchronous update	A case reducer in the slice
A request	`createAsyncThunk`
Firing it from the UI	`dispatch(thunk(arg))`
Loading and error flags	`extraReducers` lifecycle cases

`useDispatch` returns the store's `dispatch`, and its reference is stable across renders. `createAsyncThunk` wraps a promise and dispatches `pending`, `fulfilled` and `rejected` around it — but "it does not generate any reducer functions, since it does not know what data

you're fetching", so you decide what each state means in `extraReducers`. Inside the payload creator, `thunkAPI` carries `dispatch`, `getState`, `rejectWithValue` and an abort `signal`.

```
export const fetchTodos = createAsyncThunk(
  "todos/fetch",
  async (id) => {
    const r = await fetch(`/api/todos/${id}`);
    return r.json(); // -> action.payload
  },
);
// In the slice, handle the lifecycle actions:
const extraReducers = (b) =>
  b.addCase(fetchTodos.pending, (s) => {
    s.loading = true;
  }).addCase(fetchTodos.fulfilled, (s, a) => {
    s.items = a.payload;
    s.loading = false;
  });
```

> **Gotcha:** `dispatch(fetchTodos(id))` resolves whether the request succeeded or failed — the rejection is an action, not a thrown error. Chain `.unwrap()` when the calling component needs to `await` the result and branch on failure.

Derived data

Selector	Memoize it?
`s => s.todos.items`	No — the same reference already
`items.filter(...)`	Yes — a new array each call
Takes an argument per component	Yes, via a factory

Unmemoized selectors "recalculate after every dispatched action", even when nothing they read has changed. `createSelector` takes input selectors plus a result function and recomputes only when an input

changes, returning the identical reference otherwise. Its default cache size is **1**, which is the detail that bites: one shared instance called with different arguments from several components memoizes nothing. Keep selectors in the slice file, beside the reducer that owns the shape.

```
import { createSelector } from "@reduxjs/toolkit";

const selectItems = (s) => s.todos.items;

// Recomputes only when items changes; otherwise
// hands back the very same array reference.
export const selectDone = createSelector(
  [selectItems],
  (items) => items.filter((t) => t.done),
);
```

> **Gotcha:** A list that renders `selectByCategory(state, cat)` for five categories thrashes a shared memoized selector — each call evicts the previous one. Build one instance per component with a factory wrapped in `useMemo`.

RTK Query

Piece	What it does
`b.query`	A read endpoint, giving `useGetXQuery()`
`b.mutation`	A write endpoint, giving a trigger hook
`isLoading` / `isFetching`	First load / any request in flight
`providesTags`	This query supplies a tag
`invalidatesTags`	This mutation refetches those queries
`refetch()`	Force a request now

RTK Query is “a powerful data fetching and caching tool” that replaces the thunk-plus-slice-plus-loading-flag ritual: you declare endpoints and it generates the hooks. Identical queries from different components de-duplicate into one request and one cache entry — “any future request that produces the same `queryCacheKey` will be de-duped against the original” — and the entry is dropped when the last subscriber unmounts. Add `api.reducerPath` and `api.middleware` to `configureStore` once.

```
// createApi from "@reduxjs/toolkit/query/react"
export const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: "/api/" }),
  endpoints: (b) => ({
    getTodos: b.query({ query: () => "todos" }),
  }),
});
export const { useGetTodosQuery } = api;
```

```
function Todos() {
  const { data, isLoading } = useGetTodosQuery();
  if (isLoading) return <p>Loading…</p>;
  return <p>{data.length} todos</p>;
}
```

Tags are the practical payoff: rather than refetching by hand after every write, a mutation names the tags it invalidates and every mounted query providing them refetches itself. Start with a plain string tag, then narrow to `{ type: "Todo", id }` when invalidating the whole list is too blunt. For instant feedback, update the cache inside the mutation's `onQueryStarted` with `updateQueryData` and roll it back if the promise rejects.

```
// Inside endpoints: (b) => ({ … })
getTodos: b.query({
  query: () => "todos",
  providesTags: ["Todo"],
}),
```

```
addTodo: b.mutation({
  query: (body) => ({
    url: "todos", method: "POST", body,
  }),
  invalidatesTags: ["Todo"],
}),
```

> **Gotcha:** `isLoading` is false during a refetch — only `isFetching` is true. A spinner wired to `isLoading` never appears on updates; one wired to `isFetching` flashes on every poll. `data` survives both, so render stale data rather than a blank screen.

When to reach for Redux

State	Where it belongs
One component's UI	`useState`
Server data	RTK Query, or React Query

State	Where it belongs
Shared and frequently updated	The Redux store
Rarely-changing config or theme	Context

The official advice is deliberately conservative — “don’t use Redux until you have problems with vanilla React” — and the signs it fits are concrete: lots of state needed in many places, updated frequently, with complex update logic, in a medium or large codebase worked on by several people. Much of what feels like “we need Redux” is really server-cache pain, so reach for RTK Query first and keep the store for state your client genuinely owns: selections, drafts, wizards, undo stacks.

```
useState        one component
lifted state    a couple of siblings
context         rarely-changing, app-wide
RTK Query       anything the server owns
```

```
Redux store     shared client state, complex
updates
```

> **Gotcha:** Hand-rolling server data into slices rebuilds loading flags, request de-duplication, cache lifetimes and invalidation — all of which RTK Query already ships. That reinvention, not Redux itself, is where most “Redux is too much boilerplate” complaints actually come from.

Further reading

- Redux — Three Principles
- Redux Toolkit — Getting Started
- createSlice
- createAsyncThunk
- RTK Query overview
- React Redux hooks
- Deriving data with selectors