Redux Toolkit and JavaScript Collections: A Beginner-Friendly Guide

November 12, 2025

Contents

1 Redux Toolkit: `createSlice`

1.1 What it is

`createSlice` helps you define a piece of Redux state with:

- a slice **name**
- the **initial state**
- **reducers** (how state changes)
- auto-generated **action creators**
- optional **selectors**

It uses Immer, so you can write state changes with simple assignments; Redux still keeps state immutable behind the scenes.

1.2 Why use it

- Less boilerplate than hand-written Redux.
- Clear, single place for a feature's state and logic.
- Safer, concise update code thanks to Immer.

1.3 How it looks in this project

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

type UiState = {
  filter: "all" | "open" | "done"
  selectedTodoId: string | null
}

const initialState: UiState = { filter: "all", selectedTodoId: null }

export const uiSlice = createSlice({
  name: "todosUi",
  initialState,
  reducers: {
    selectTodo: (state, action) => {
      state.selectedTodoId = action.payload
    },
    setFilter: (state, action) => {
      state.filter = action.payload
    }
  },
  selectors: {
    selectFilter: state => state.filter,
    selectSelectedTodoId: state => state.selectedTodoId
  }
})

export const { selectTodo, setFilter } = uiSlice.actions
export const { selectFilter, selectSelectedTodoId } = uiSlice.selectors
export const uiReducer = uiSlice.reducer
```

Listing 1: UI slice with reducers and selectors

Explanation: This slice declares UI state (`filter`, `selectedTodoId`), reducers using Immer-style assignments, and colocated selectors. It also exports typed action creators and the reducer for store integration.

1.4 Where it plugs into the store

```
import { configureStore } from "@reduxjs/toolkit"
import { todosApi } from "./services/todosApi"
import { uiReducer } from "./features/todos/uiSlice"

export const store = configureStore({
  reducer: {
    [todosApi.reducerPath]: todosApi.reducer,
    todosUi: uiReducer
  },
  middleware: getDefault => getDefault().concat(todosApi.middleware)
})
```

Listing 2: Store integration with RTK Query and slice reducer

Explanation: The store mounts RTK Query's cache at `todosApi.reducerPath` and the UI slice at `todosUi`. The RTK Query middleware enables caching, invalidation, polling, and request deduplication.

Quick tips

- Keep each slice focused on one feature.
- Reducers should be fast, side-effect free.
- Put async/data fetching in RTK Query (next section).

2 RTK Query: `createApi`

2.1 What it is

`createApi` is Redux Toolkit's built-in data fetching and caching. You describe endpoints; it generates React hooks and handles caching and refetching for you.[1]

2.2 Why use it

- Less code: hooks are generated for you.
- Smart cache: avoids unnecessary network calls.
- Easy updates: tag invalidation refreshes the right data.

[1] `@reduxjs/toolkit/query/react` provides the React hooks.

2.3 How it looks in this project

```
import { createApi, fetchBaseQuery } from "@reduxjs/toolkit/query/react"

export const todosApi = createApi({
  reducerPath: "todosApi",
  baseQuery: fetchBaseQuery({ baseUrl: "http://localhost:4000" }),
  tagTypes: ["History", "Todos"],
  endpoints: builder => ({
    getTodos: builder.query({
      query: () => "/todos",
      providesTags: (result) =>
        result
          ? [
              ...result.map(t => ({ type: "Todos", id: t.id })),
              { type: "Todos", id: "LIST" }
            ]
          : [{ type: "Todos", id: "LIST" }]
    }),
    addTodo: builder.mutation({
      query: (title) => ({ url: "/todos", method: "POST", body: { title } }),
      invalidatesTags: [{ type: "Todos", id: "LIST" }]
    }),
    // ... other mutations and queries
  })
})

export const { useGetTodosQuery, useAddTodoMutation } = todosApi
```

Listing 3: Todos API with tags and invalidation

Explanation: `getTodos` provides tags per item and a list tag; `addTodo` invalidates the list tag so `getTodos` re-fetches. Hooks `useGetTodosQuery` and `useAddTodoMutation` are generated for components.

2.4 Caching in simple words

- **providesTags**: queries say “I filled these tags.”
- **invalidatesTags**: mutations say “These tags are stale.”
- When tags are invalidated, matching queries re-fetch automatically.

Quick tips

- Use queries for reads, mutations for writes.
- Drive UI from hook state: `data`, `isLoading`, `error`.
- Invalidate tags on writes so the UI stays fresh.

3 JavaScript `Map`, `WeakMap`, and `Set`

3.1 `Map`: key-value store

Think “dictionary”: any value can be a key (including objects). Keeps items in the order you added them. **Equality rules**: primitive keys (e.g., `string`, `number`, `boolean`, `bigint`, `symbol`) are compared by value (SameValueZero), while object keys are compared by reference. Using strings as keys is safe and deterministic; using objects as keys requires using the exact same object reference.

Common operations

- **CRUD**: `set(k, v)`, `get(k)`, `has(k)`, `delete(k)`, `clear()`.
- **Iteration**: `map.keys()`, `map.values()`, `map.entries()`, `for...of`.
- **Size**: `map.size` (O(1)).

```
const m = new Map()
const user = { id: "42" }
m.set(user, ["a", "b"])
m.set("threshold", 3)

for (const [k, v] of m) {
  // preserves insertion order
}
```

Listing 4: Map basics and iteration

Explanation: `Map` accepts heterogeneous keys (object and string here), keeps insertion order, and iterates as `[key, value]` pairs. String keys compare by value; object keys compare by reference. Prefer `Map` when you need non-string keys or want to avoid key coercion of plain objects.

Examples

- **Cache by composite input**: key by an options object reference to memoize an expensive computation.
- **Stable ID lookup with order**: map task IDs to metadata while preserving creation order.
- **Cross-graph associations**: map DOM nodes or external handles to app data without mutating the sources.

3.2 `WeakMap`: private data for objects

Like `Map`, but keys must be objects, and they are held *weakly*. If the object is no longer used elsewhere, the entry disappears automatically (helps prevent memory leaks).

Notes

- Keys must be objects; primitives are invalid.
- Non-enumerable by design: no `size`, no iteration; prevents observation of GC.

- Ideal for private per-instance metadata without memory leaks.

```
const meta = new WeakMap()

class Widget {
  constructor(config) {
    meta.set(this, { clicks: 0, config })
  }
  click() {
    meta.get(this).clicks++
  }
}
```

Listing 5: WeakMap for private metadata

Explanation: `WeakMap` *attaches per-instance metadata without preventing garbage collection. It's not iterable and has no* `size`*, which avoids observing GC behavior.*

Examples

- **Per-instance private state**: store internal counters/config for class instances without exposing fields.
- **DOM element metadata**: attach observers or measured layout data keyed by elements; entries vanish when elements are removed.
- **Memoization keyed by objects**: cache derived results for specific object identities without leaks.

3.3 `Set`: list of unique values

Stores unique values (no duplicates), in insertion order. Perfect for tags, selected IDs, etc.

Common operations

- **CRUD**: `add(v)`, `has(v)`, `delete(v)`, `clear()`.
- **Iteration**: `set.values()`, `set.entries()`, `for...of`; preserves insertion order.
- **Size**: `set.size` (O(1)).

```
const tags = new Set(["urgent", "work", "urgent"])
// => {"urgent", "work"}

tags.add("home")
tags.has("work") // true
for (const t of tags) {
  // iterate unique tags
}
```

Listing 6: Set for de-duplication and membership

Explanation: `Set` *deduplicates values automatically and supports fast membership tests with* `has`*. It preserves insertion order when iterating.*

Examples

- **Tags/labels**: maintain a unique set of labels from user input.
- **Selections**: store selected item IDs for quick membership tests and toggling.
- **Deduplication**: remove duplicates from arrays while preserving order (`[...new Set(arr)]`).

3.4 Which one to use when

- **Map**: dictionary where keys can be anything (including objects).
- **WeakMap**: attach private data to objects without leaking memory.
- **Set**: list of unique items.

4 Putting it together in a component

```
import { useGetTodosQuery, useAddTodoMutation } from "./services/todosApi"
import { selectFilter, setFilter } from "./features/todos/uiSlice"
import { useSelector, useDispatch } from "react-redux"

export function TodoApp() {
  const dispatch = useDispatch()
  const filter = useSelector(state => selectFilter(state.todosUi))
  const { data: todos = [], isLoading } = useGetTodosQuery()
  const [addTodo, { isLoading: isAdding }] = useAddTodoMutation()

  return (
    // render todos; dispatch(setFilter("open"))
    // call addTodo("title")
  )
}
```

Listing 7: Query + mutation with generated hooks

Explanation: `useGetTodosQuery` exposes `data` and loading flags; `useAddTodoMutation` returns a trigger and mutation state. UI state is read via `useSelector` and modified via `dispatch`.

5 React `useSelector`: selecting state safely and efficiently

5.1 What it is

`useSelector` reads data from the Redux store in a React component. The component re-renders when the selected value changes by *strict* equality (===) or a provided equality function.

5.2 Professional guidance

- **Select the smallest shape** the component needs (primitives or stable objects) to minimize re-renders.

- **Use memoized selectors** (e.g., `reselect`) for derived data to avoid recomputation and stabilize references.
- **Avoid inline object/array construction** inside the selector; it creates new references every render.
- **Prefer slice selectors** colocated with the slice to centralize state knowledge.

5.3 Type-safe patterns

Option A: pass slice state into generated selectors

```
import { useSelector } from "react-redux"
import type { RootState } from "./store"
import { selectFilter } from "./features/todos/uiSlice"

export function FilterChip() {
  const filter = useSelector((state: RootState) => selectFilter(state.todosUi))
  return <span>{filter}</span>
}
```

Listing 8: Using slice selectors with root state

Explanation: This pattern scopes the selector to the `todosUi` *slice and retains RootState type safety. Returning a primitive minimizes unnecessary re-renders.*

Option B: typed selector hook (project standard)

```
import { useSelector, type TypedUseSelectorHook } from "react-redux"
import type { RootState } from "./store"

export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector
```

Listing 9: Define a typed selector hook once

Explanation: A typed selector hook removes repeated RootState annotations at call sites and ensures selectors remain type-safe across the app.

```
import { useAppSelector } from "./useAppSelector"
import { selectSelectedTodoId } from "./features/todos/uiSlice"

export function SelectedInfo() {
  const selectedId = useAppSelector(state => selectSelectedTodoId(state.todosUi))
  return <div>{selectedId ?? "None selected"}</div>
}
```

Listing 10: Consume with typed selector hook

Explanation: Consuming the typed hook improves ergonomics. Returning a primitive (`string|null`*) keeps equality checks cheap and stable.*

5.4 Performance notes

- The default comparison is strict equality. If you must return shallow objects, you can use: `useSelector(selectFn, shallowEqual)`.

- For non-trivial derived data, use `createSelector` (from `reselect`) to memoize:

```
import { createSelector } from "reselect"
import type { RootState } from "./store"

const selectTodos = (state: RootState) => state.todosApi // RTKQ cache slice
// Example: derive counts from UI-filtered todos (sketch; adapt to your shape)
export const selectOpenCount = createSelector(
  [
    (state: RootState) => state.todosUi.filter,
    (state: RootState) => state
  ],
  (filter, state) => {
    // compute from state and filter; return a primitive for stable equality
    return 0
  }
)
```

Listing 11: Memoized derived selector with reselect

Explanation: `createSelector` memoizes derived results based on inputs. Returning primitives (numbers/strings/booleans) helps `useSelector` avoid re-renders. Adapt the computation to your actual state shape.

6 Cheat sheet

- **createSlice**: define state + reducers + actions (+ selectors).
- **createApi**: define queries/mutations; use generated hooks.
- **Tags**: queries `providesTags`, mutations `invalidatesTags`.
- **useSelector**: select minimal stable values; memoize derived data.
- **Map**: key-value by any key; **WeakMap**: object keys, GC-safe; **Set**: unique values.