**ScopeTreasure Courses**

REACT COURSES · LESSON 5.20 · CODE SCOPE SERIES

# Redux Toolkit

Modern Redux without the boilerplate: configureStore, createSlice, useSelector and useDispatch, and a cart slice that adds and removes items live.

## Introduction

Lesson 5.11 taught Redux's ideas — one store, actions, pure reducers. Written the old way, those ideas drowned in boilerplate: action-type constants, switch statements, manual immutable updates, store setup plumbing. **Redux Toolkit (RTK)** is the official modern way: the same ideas, with the ceremony removed — and the package the Redux team tells every new app to use. *(Roman Urdu: Redux Toolkit wohi Redux hai, bas purani lambi formality khatam — kam code, wohi nizam.)*

## Key points

- **Setup is two moves:** `configureStore({ reducer: { cart: cartReducer } })` builds the store (with DevTools and good defaults built in), and the app is wrapped in `<Provider store={store}>` — Context, working quietly underneath.
- **`createSlice` writes three things at once:** give it a `name`, an `initialState` and `reducers`, and it generates the reducer **and** the action creators for you. `cartSlice.actions.addItem` exists without you declaring a single action type.
- **Reducers may "mutate" — safely:** RTK bundles **Immer**, so inside a slice reducer you write `state.items.push(item)` and Immer produces the correct immutable copy behind the scenes. (Outside RTK this remains a bug — the permission belongs to Immer only.)

  *(Roman Urdu: Slice ke andar push likh sakte ho — Immer peeche asal copy khud bana deta hai.)*
- **Components read with `useSelector`:** `const items = useSelector((state) => state.cart.items);` — pick your slice's piece; the component re-renders when *that piece* changes.
- **Components change with `useDispatch`:** `const dispatch = useDispatch();` then `dispatch(addItem(product))` — the same dispatch verb as always, now carrying Toolkit-generated actions.
- **File habit:** one slice per file (`cartSlice.js`, `userSlice.js`), each exporting its reducer as default and its actions by name — the structure every modern Redux codebase shares.

## Code examples

### Example 1 — The cart slice: a whole feature in one small file

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

const cartSlice = createSlice({
  name: "cart",
  initialState: { items: [] },
  reducers: {
    addItem(state, action) {
      state.items.push(action.payload);
    },
    removeItem(state, action) {
      state.items = state.items.filter((item) => item.id !== action.payload);
    },
    clearCart(state) {
      state.items = [];
    },
  },
});

export const { addItem, removeItem, clearCart } = cartSlice.actions;
export default cartSlice.reducer;
```

What it does: One `createSlice` call replaces the old trio of action constants, action creators and a switch reducer. The reducers read like direct edits — push, filter, clear — while Immer guarantees the store still receives fresh immutable states. The generated actions come out at the bottom, named and ready.

**What you will see:** nothing visual yet — this file is the engine. Its whole API is three verbs: `addItem(product)`, `removeItem(id)`, `clearCart()`.

### Example 2 — The store and the Provider

```
// store.js
import { configureStore } from "@reduxjs/toolkit";
import cartReducer from "./cartSlice";

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

// main.jsx
import { Provider } from "react-redux";
import { store } from "./store";

root.render(
  <Provider store={store}>
    <App />
  </Provider>
);
```

What it does: `configureStore` assembles the slices into one state tree (`state.cart…`), and the `Provider` — the Context wrapper from react-redux — makes that store reachable from every component. Two small files and the app's shared state has a home.

**What you will see:** in Redux DevTools, a state tree shaped exactly like the configuration: `{ cart: { items: [] } }`, ready to fill.

### Example 3 — Reading and dispatching: badge and buttons

```
import { useSelector, useDispatch } from "react-redux";
import { addItem, removeItem } from "./cartSlice";

function Shop() {
  const items = useSelector((state) => state.cart.items);
  const dispatch = useDispatch();
  const total = items.reduce((sum, i) => sum + i.price, 0);

  return (
    <div>
      <p>Cart: {items.length} items · Rs {total}</p>
      <button onClick={() => dispatch(addItem({ id: 1, name: "Chai", price: 250 }))}>
        Add Chai
      </button>
      <ul>
        {items.map((item, idx) => (
          <li key={idx}>
            {item.name} — Rs {item.price}
            <button onClick={() => dispatch(removeItem(item.id))}>Remove</button>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

What it does: `useSelector` subscribes the component to the cart slice; `dispatch` sends the generated actions. The total is derived during render, never stored. Every part of the 5.11 triangle is present — with a fraction of the old code.

**What you will see:** each "Add Chai" grows the list and total together; Remove deletes its row; Redux DevTools shows the `cart/addItem` actions marching in order, replayable one by one.

## Quick recap

- **RTK = Redux without ceremony:** `configureStore` + `<Provider>` for the store, **`createSlice`** generating reducer + actions together.
- Inside slice reducers, **Immer** lets you write pushes and assignments while the store still gets immutable updates.
- Components live on **`useSelector`** (read a slice) and **`useDispatch`** (send a generated action).

## Common mistakes

1. **Mutating state outside Immer's reach** — treating a selector result as editable, or mutating inside a plain hand-written reducer, breaks Redux's change detection. The fix: "mutation style" is legal *only* inside `createSlice` reducers (and only on the draft state they receive).
2. **Selecting the whole state** — `useSelector(state => state)` re-renders the component on *every* action anywhere in the app. The fix: select the smallest piece the component actually shows.
3. **Dispatching during render** — calling `dispatch` in the component body fires actions mid-render and loops. The fix: dispatch lives in event handlers and effects, exactly like setState discipline.

## Practice

1. Add a `userSlice` beside the cart: state `{ name: "", loggedIn: false }`, reducers `login` (takes the name as payload) and `logout`. Register it in the store, and build a small header that shows "Hello, {name}" or a Login button by selecting the user slice.
2. Give the cart slice quantities: `addItem` should raise `qty` when the id is already in the cart instead of adding a duplicate row. Update the Shop list to show `name × qty` and make the total use `price * qty`.

## Practice quiz

**1. What does createSlice give you from one definition?**

- Only the reducer
- The reducer and the action creators together, from a name, an initialState and reducers; cartSlice.actions.addItem exists without declaring a single action type
- Only the action creators
- A whole app

**Answer:** The reducer and the action creators together, from a name, an initialState and reducers; cartSlice.actions.addItem exists without declaring a single action type

**2. Why may a slice reducer write state.items.push(item) when plain reducers may not mutate?**

- Because RTK removed the rule entirely
- RTK bundles Immer, which produces the correct immutable copy behind the scenes; outside createSlice reducers this remains a bug
- Because push is not a mutation
- Because slices are not real reducers

**Answer:** RTK bundles Immer, which produces the correct immutable copy behind the scenes; outside createSlice reducers this remains a bug

**3. How do components read and change RTK state?**

- With useState and useEffect

- Read with useSelector picking a slice's piece, change with useDispatch sending a generated action
- With props only
- With context directly

**Answer:** Read with useSelector picking a slice's piece, change with useDispatch sending a generated action

---

**Watch the Code Scope channel:** https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

**Read online:** https://scopetreasure.com/notes/react-redux-toolkit · More free tools and notes at https://scopetreasure.com