Understanding Redux with React: A Guide to State Management

This application uses **Redux Toolkit** and **React Redux** to manage the global count state, strictly following the established Redux flow: **Action → Reducer → Store → React Component**.

Redux Coding Flow and Reasons

The implementation is broken down into four distinct steps, detailing the logical flow for state management.

1. Actions (redux/action/counterAction.ts)

- **Purpose:** Actions are plain JavaScript objects that serve as "messages" describing *what* happened in the application. They are the only way to send data to the Redux store.
- **Implementation:** Redux Toolkit's **createAction** is used to define action creators (increment, decrement, setValue). These functions return actions with a unique type string (e.g., "counter/increment") that the Reducer uses to identify the intent.

2. Reducer (redux/reducer/counterReducer.ts)

- **Purpose:** Reducers are **pure functions** that take the current state and an action, and return a *new state*. They specify *how* the state changes in response to an action.
- **Implementation:** Redux Toolkit's **createReducer** is used with an immutable **builder pattern**.
    - The Reducer listens for specific actions (addCase).
    - Redux Toolkit uses the **Immer** library, allowing the developer to write "mutating" logic (state.count += 1) that is safely translated into immutable updates behind the scenes.
    - The setValue action demonstrates how to use the action.payload to set the new state value.

3. Store Configuration (redux/store.ts)

- **Purpose:** The Store is the single source of truth that holds the entire state tree. It is responsible for dispatching actions and running the reducers.
- **Implementation:**
    - **configureStore** (from Redux Toolkit) initializes the store, setting up the root state structure (e.g., counter: counterReducer).
    - It also automatically bundles essential middleware (like Redux Thunk for async logic) and sets up DevTools integration.
    - TypeScript types (RootState and AppDispatch) are exported to ensure type safety across the entire application.

4. React Integration (Components)

- **Purpose:** Connecting React components to the global Redux Store to read state and dispatch updates.
- **Implementation:**
  - **Provider (in index.tsx):** The entire application is wrapped in the <Provider> component, passing the store instance via props. This makes the store accessible via hooks to all child components.
  - **Hooks (App.tsx, CountView.tsx):** Components use specialized hooks to interact with the store.

Why useSelector and useDispatch? (React Redux Hooks)

These two hooks are the primary tools for connecting functional components to the Redux Store.

useSelector

Concept	Description
Purpose	To **read** specific data from the Redux state.
Mechanism	You pass a **selector function** (e.g., (state: RootState) => state.counter.count).
Benefit	The component automatically **subscribes** to that specific piece of state. When *only* that data changes, the component re-renders. This is highly optimized and prevents "prop drilling."

useDispatch

Concept	Description
Purpose	To get the dispatch function, which is used to **send** actions to the store.
Mechanism	You call const dispatch = useDispatch() once in the component.
Benefit	Components request state changes by calling dispatch(action). This enforces the **separation of concerns**, as the components only declare *what* they want to happen, and the Reducers handle *how* the state changes.

Naming Conventions Used

Element	Example	Description
Action	increment, setValue	Defines the event that occurred.
Action Type	"counter/increment"	The unique identifier string for the action.
Reducer	counterReducer	The function defining state logic for a feature (slice).
State Slice	counter	The key under which the reducer's state is stored in the global RootState.
Types	RootState, AppDispatch	TypeScript types for the state shape and the dispatch function.