React Reducer CheatSheet

by Ndeye Fatou Diop

Mistakes to avoid with useReducer

- **Mutating the state:** Always return a new object from the reducer.
- **Inconsistent action names:** Stick to a convention for your actions names.
- **Impure reducer function 🛑:** Given a state/action, it should always return the same result.
- **Not handling unknown actions**

```
function reducer(state,
action) {
  switch (action.type) {
    case "increment":
      return { count:
state.count + 1 };
    default:
      // ✅ Safe fallback
      return state;
  }
}
```

Tip: Use useImmerReducer

useImmerReducer (from **Immer**) simplifies state updates by allowing you to directly "mutate" state in the reducer.

What is useReducer?

- useReducer is a React Hook used for managing complex state logic, especially when state depends on previous state.

```
const [state, dispatch] = useReducer(reducer, initialState);
```

- reducer: A function to determine the new state based on the action.
- initialState: The starting value of the state.
- dispatch: A function to trigger state updates.

How to use useReducer (example Counter app)

```
const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    default:
      throw new Error("Unknown action type");
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: "increment" })}>
        +
      </button>
      <button onClick={() => dispatch({ type: "decrement" })}>
        -
      </button>
    </div>
  );
}
```

When to Use useReducer vs useState

Use useState	Use useReducer
Simple state logic	Complex state logic with multiple transitions (the reducer function is easily testable)
Independent state values	Interdependent state values
“Fixed” state structure	“Dynamic” state structure

Combine useReducer + context for efficient state management

You can combine reducers and context in vanilla React to manage state effectively. By saving the dispatch function in context, you enable state updates from anywhere in your app.

```
const TodosContext = createContext(null);
const TodosDispatchContext = createContext(null);

export function TodosApp() {
  const [todos, dispatch] = useReducer(todosReducer, []);

  return (
    <TodosContext.Provider value={todos}>
      <TodosDispatchContext.Provider value={dispatch}>
        ...
      </TodosDispatchContext.Provider>
    </TodosContext.Provider>
  );
}
```