Redux Style Guide

Introduction

This is the official style guide for writing Redux code. **It lists our recommended patterns, best practices, and suggested approaches for writing Redux applications.**

Both the Redux core library and most of the Redux documentation are unopinionated. There are many ways to use Redux, and much of the time there is no single "right" way to do things.

However, time and experience have shown that for some topics, certain approaches work better than others. In addition, many developers have asked us to provide official guidance to reduce decision fatigue.

With that in mind, **we've put together this list of recommendations to help you avoid errors, bikeshedding, and anti-patterns**. We also understand that team preferences vary and different projects have different requirements, so no style guide will fit all sizes. **You are encouraged to follow these recommendations, but take the time to evaluate your own situation and decide if they fit your needs**.

Finally, we'd like to thank the Vue documentation authors for writing the Vue Style Guide page, which was the inspiration for this page.

Rule Categories

We've divided these rules into three categories:

Priority A: Essential

These rules help prevent errors, so learn and abide by them at all costs. Exceptions may exist, but should be very rare and only be made by those with expert knowledge of both JavaScript and Redux.

Priority B: Strongly Recommended

These rules have been found to improve readability and/or developer experience in most projects. Your code will still run if you violate them, but violations should be rare and well-justified. **Follow these rules whenever it is reasonably possible**.

Priority C: Recommended

Where multiple, equally good options exist, an arbitrary choice can be made to ensure consistency. In these rules, **we describe each acceptable option and suggest a default choice**. That means you can feel free to make a different choice in your own codebase, as long as you're consistent and have a good reason. Please do have a good reason though!

Priority A Rules: Essential

Do Not Mutate State ESSENTIAL

Mutating state is the most common cause of bugs in Redux applications, including components failing to re-render properly, and will also break time-travel debugging in the Redux DevTools. **Actual mutation of state values should always be avoided**, both inside reducers and in all other application code.

Use tools such as `redux-immutable-state-invariant` to catch mutations during development, and Immer to avoid accidental mutations in state updates.

> **Note**: it is okay to modify *copies* of existing values - that is a normal part of writing immutable update logic. Also, if you are using the Immer library for immutable updates, writing "mutating" logic is acceptable because the real data isn't being mutated - Immer safely tracks changes and generates immutably-updated values internally.

Reducers Must Not Have Side Effects ESSENTIAL

Reducer functions should *only* depend on their `state` and `action` arguments, and should only calculate and return a new state value based on those arguments. **They must not execute any kind of asynchronous logic (AJAX calls, timeouts, promises), generate random values (`Date.now()`, `Math.random()`), modify variables outside the reducer, or run other code that affects things outside the scope of the reducer function**.

> **Note**: It is acceptable to have a reducer call other functions that are defined outside of itself, such as imports from libraries or utility functions, as long as they follow the same rules.

▶ **Detailed Explanation**

Do Not Put Non-Serializable Values in State or Actions ESSENTIAL

Avoid putting non-serializable values such as Promises, Symbols, Maps/Sets, functions, or class instances into the Redux store state or dispatched actions. This ensures that

capabilities such as debugging via the Redux DevTools will work as expected. It also ensures that the UI will update as expected.

> **Exception**: you may put non-serializable values in actions *if* the action will be intercepted and stopped by a middleware before it reaches the reducers. Middleware such as `redux-thunk` and `redux-promise` are examples of this.

Only One Redux Store Per App ESSENTIAL

A standard Redux application should only have a single Redux store instance, which will be used by the whole application. It should typically be defined in a separate file such as `store.js`.

Ideally, no app logic will import the store directly. It should be passed to a React component tree via `<Provider>`, or referenced indirectly via middleware such as thunks. In rare cases, you may need to import it into other logic files, but this should be a last resort.

Priority B Rules: Strongly Recommended

Use Redux Toolkit for Writing Redux Logic STRONGLY RECOMMENDED

Redux Toolkit is our recommended toolset for using Redux. It has functions that build in our suggested best practices, including setting up the store to catch mutations and enable the Redux DevTools Extension, simplifying immutable update logic with Immer, and more.

You are not required to use RTK with Redux, and you are free to use other approaches if desired, but **using RTK will simplify your logic and ensure that your application is set up with good defaults**.

Use Immer for Writing Immutable Updates STRONGLY RECOMMENDED

Writing immutable update logic by hand is frequently difficult and prone to errors. Immer allows you to write simpler immutable updates using "mutative" logic, and even freezes your state in development to catch mutations elsewhere in the app. **We recommend using Immer for writing immutable update logic, preferably as part of Redux Toolkit**.

Structure Files as Feature Folders with Single-File Logic STRONGLY RECOMMENDED

Redux itself does not care about how your application's folders and files are structured. However, co-locating logic for a given feature in one place typically makes it easier to maintain that code.

Because of this, **we recommend that most applications should structure files using a "feature folder" approach** (all files for a feature in the same folder). Within a given feature folder, **the Redux logic for that feature should be written as a single "slice" file**, preferably using the Redux Toolkit `createSlice` API. (This is also known as the "ducks" pattern). While older Redux codebases often used a "folder-by-type" approach with separate folders for "actions" and "reducers", keeping related logic together makes it easier to find and update that code.

▶ **Detailed Explanation: Example Folder Structure**

Put as Much Logic as Possible in Reducers STRONGLY RECOMMENDED

Wherever possible, **try to put as much of the logic for calculating a new state into the appropriate reducer, rather than in the code that prepares and dispatches the action** (like a click handler). This helps ensure that more of the actual app logic is easily testable, enables more effective use of time-travel debugging, and helps avoid common mistakes that can lead to mutations and bugs.

There are valid cases where some or all of the new state should be calculated first (such as generating a unique ID), but that should be kept to a minimum.

▶ **Detailed Explanation**

Reducers Should Own the State Shape STRONGLY RECOMMENDED

The Redux root state is owned and calculated by the single root reducer function. For maintainability, that reducer is intended to be split up by key/value "slices", with **each "slice reducer" being responsible for providing an initial value and calculating the updates to that slice of the state**.

In addition, slice reducers should exercise control over what other values are returned as part of the calculated state. **Minimize the use of "blind spreads/returns"** like `return action.payload` or `return {...state, ...action.payload}`, because those rely on the code that dispatched the action to correctly format the contents, and the reducer effectively gives up its ownership of what that state looks like. That can lead to bugs if the action contents are not correct.

> **Note**: A "spread return" reducer may be a reasonable choice for scenarios like editing data in a form, where writing a separate action type for each individual field would be time-consuming and of little benefit.

▶ **Detailed Explanation**

Name State Slices Based On the Stored Data STRONGLY RECOMMENDED

As mentioned in Reducers Should Own the State Shape , the standard approach for splitting reducer logic is based on "slices" of state. Correspondingly, `combineReducers` is the standard function for joining those slice reducers into a larger reducer function.

The key names in the object passed to `combineReducers` will define the names of the keys in the resulting state object. Be sure to name these keys after the data that is kept inside, and avoid use of the word "reducer" in the key names. Your object should look like `{users: {}, posts: {}}`, rather than `{usersReducer: {}, postsReducer: {}}`.

▶ **Detailed Explanation**

Organize State Structure Based on Data Types, Not Components

STRONGLY RECOMMENDED

Root state slices should be defined and named based on the major data types or areas of functionality in your application, not based on which specific components you have in your UI. This is because there is not a strict 1:1 correlation between data in the Redux store and components in the UI, and many components may need to access the same data. Think of the state tree as a sort of global database that any part of the app can access to read just the pieces of state needed in that component.

For example, a blogging app might need to track who is logged in, information on authors and posts, and perhaps some info on what screen is active. A good state structure might look like `{auth, posts, users, ui}`. A bad structure would be something like `{loginScreen, usersList, postsList}`.

Treat Reducers as State Machines STRONGLY RECOMMENDED

Many Redux reducers are written "unconditionally". They only look at the dispatched action and calculate a new state value, without basing any of the logic on what the current state might be. This can cause bugs, as some actions may not be "valid" conceptually at certain times depending on the rest of the app logic. For example, a "request succeeded" action should only have a new value calculated if the state says that it's already "loading", or an "update this item" action should only be dispatched if there is an item marked as "being edited".

To fix this, **treat reducers as "state machines", where the combination of both the current state *and* the dispatched action determines whether a new state value is actually calculated**, not just the action itself unconditionally.

▶ **Detailed Explanation**

Normalize Complex Nested/Relational State STRONGLY RECOMMENDED

Many applications need to cache complex data in the store. That data is often received in a nested form from an API, or has relations between different entities in the data (such as a blog that contains Users, Posts, and Comments).

Prefer storing that data in a "normalized" form in the store. This makes it easier to look up items based on their ID and update a single item in the store, and ultimately leads to better performance patterns.

Keep State Minimal and Derive Additional Values STRONGLY RECOMMENDED

Whenever possible, **keep the actual data in the Redux store as minimal as possible, and *derive* additional values from that state as needed**. This includes things like calculating filtered lists or summing up values. As an example, a todo app would keep an original list of todo objects in state, but derive a filtered list of todos outside the state whenever the state is updated. Similarly, a check for whether all todos have been completed, or number of todos remaining, can be calculated outside the store as well.

This has several benefits:

- The actual state is easier to read
- Less logic is needed to calculate those additional values and keep them in sync with the rest of the data
- The original state is still there as a reference and isn't being replaced

Deriving data is often done in "selector" functions, which can encapsulate the logic for doing the derived data calculations. In order to improve performance, these selectors can be *memoized* to cache previous results, using libraries like `reselect` and `proxy-memoize`.

Model Actions as Events, Not Setters STRONGLY RECOMMENDED

Redux does not care what the contents of the `action.type` field are - it just has to be defined. It is legal to write action types in present tense (`"users/update"`), past tense (`"users/updated"`), described as an event (`"upload/progress"`), or treated as a "setter" (`"users/setUserName"`). It is up to you to determine what a given action means in your application, and how you model those actions.

However, **we recommend trying to treat actions more as "describing events that occurred", rather than "setters"**. Treating actions as "events" generally leads to more meaningful action names, fewer total actions being dispatched, and a more meaningful action

log history. Writing "setters" often results in too many individual action types, too many dispatches, and an action log that is less meaningful.

▶ **Detailed Explanation**

Write Meaningful Action Names STRONGLY RECOMMENDED

The `action.type` field serves two main purposes:

- Reducer logic checks the action type to see if this action should be handled to calculate a new state
- Action types are shown in the Redux DevTools history log for you to read

Per Model Actions as "Events", the actual contents of the `type` field do not matter to Redux itself. However, the `type` value *does* matter to you, the developer. **Actions should be written with meaningful, informative, descriptive type fields**. Ideally, you should be able to read through a list of dispatched action types, and have a good understanding of what happened in the application without even looking at the contents of each action. Avoid using very generic action names like `"SET_DATA"` or `"UPDATE_STORE"`, as they don't provide meaningful information on what happened.

Allow Many Reducers to Respond to the Same Action STRONGLY RECOMMENDED

Redux reducer logic is intended to be split into many smaller reducers, each independently updating their own portion of the state tree, and all composed back together to form the root reducer function. When a given action is dispatched, it might be handled by all, some, or none of the reducers.

As part of this, you are encouraged to **have many reducer functions all handle the same action separately** if possible. In practice, experience has shown that most actions are typically only handled by a single reducer function, which is fine. But, modeling actions as "events" and allowing many reducers to respond to those actions will typically allow your application's codebase to scale better, and minimize the number of times you need to dispatch multiple actions to accomplish one meaningful update.

Avoid Dispatching Many Actions Sequentially STRONGLY RECOMMENDED

Avoid dispatching many actions in a row to accomplish a larger conceptual "transaction". This is legal, but will usually result in multiple relatively expensive UI updates, and some of the intermediate states could be potentially invalid by other parts of the application logic. Prefer dispatching a single "event"-type action that results in all of the

appropriate state updates at once, or consider use of action batching addons to dispatch multiple actions with only a single UI update at the end.

▶ **Detailed Explanation**

Evaluate Where Each Piece of State Should Live STRONGLY RECOMMENDED

The "Three Principles of Redux" says that "the state of your whole application is stored in a single tree". This phrasing has been over-interpreted. It does not mean that literally *every* value in the entire app *must* be kept in the Redux store. Instead, **there should be a single place to find all values that *you* consider to be global and app-wide**. Values that are "local" should generally be kept in the nearest UI component instead.

Because of this, it is up to you as a developer to decide what state should actually live in the Redux store, and what should stay in component state. **Use these rules of thumb to help evaluate each piece of state and decide where it should live**.

Use the React-Redux Hooks API STRONGLY RECOMMENDED

Prefer using the React-Redux hooks API (`useSelector` and `useDispatch`) as the default way to interact with a Redux store from your React components. While the classic `connect` API still works fine and will continue to be supported, the hooks API is generally easier to use in several ways. The hooks have less indirection, less code to write, and are simpler to use with TypeScript than `connect` is.

The hooks API does introduce some different tradeoffs than `connect` does in terms of performance and data flow, but we now recommend them as the default.

▶ **Detailed Explanation**

Connect More Components to Read Data from the Store STRONGLY RECOMMENDED

Prefer having more UI components subscribed to the Redux store and reading data at a more granular level. This typically leads to better UI performance, as fewer components will need to render when a given piece of state changes.

For example, rather than just connecting a `<UserList>` component and reading the entire array of users, have `<UserList>` retrieve a list of all user IDs, render list items as `<UserListItem userId={userId}>`, and have `<UserListItem>` be connected and extract its own user entry from the store.

This applies for both the React-Redux `connect()` API and the `useSelector()` hook.

Use the Object Shorthand Form of `mapDispatch` with `connect`

STRONGLY RECOMMENDED

The `mapDispatch` argument to `connect` can be defined as either a function that receives `dispatch` as an argument, or an object containing action creators. **We recommend always using the "object shorthand" form of `mapDispatch`**, as it simplifies the code considerably. There is almost never a real need to write `mapDispatch` as a function.

Call `useSelector` Multiple Times in Function Components

STRONGLY RECOMMENDED

When retrieving data using the `useSelector` hook, prefer calling `useSelector` many times and retrieving smaller amounts of data, instead of having a single larger `useSelector` call that returns multiple results in an object. Unlike `mapState`, `useSelector` is not required to return an object, and having selectors read smaller values means it is less likely that a given state change will cause this component to render.

However, try to find an appropriate balance of granularity. If a single component does need all fields in a slice of the state , just write one `useSelector` that returns that whole slice instead of separate selectors for each individual field.

Use Static Typing

STRONGLY RECOMMENDED

Use a static type system like TypeScript or Flow rather than plain JavaScript. The type systems will catch many common mistakes, improve the documentation of your code, and ultimately lead to better long-term maintainability. While Redux and React-Redux were originally designed with plain JS in mind, both work well with TS and Flow. Redux Toolkit is specifically written in TS and is designed to provide good type safety with a minimal amount of additional type declarations.

Use the Redux DevTools Extension for Debugging

STRONGLY RECOMMENDED

Configure your Redux store to enable debugging with the Redux DevTools Extension. It allows you to view:

- The history log of dispatched actions
- The contents of each action
- The final state after an action was dispatched
- The diff in the state after an action
- The function stack trace showing the code where the action was actually dispatched

In addition, the DevTools allows you to do "time-travel debugging", stepping back and forth in the action history to see the entire app state and UI at different points in time.

Redux was specifically designed to enable this kind of debugging, and the DevTools are one of the most powerful reasons to use Redux.

Use Plain JavaScript Objects for State STRONGLY RECOMMENDED

Prefer using plain JavaScript objects and arrays for your state tree, rather than specialized libraries like Immutable.js. While there are some potential benefits to using Immutable.js, most of the commonly stated goals such as easy reference comparisons are a property of immutable updates in general, and do not require a specific library. This also keeps bundle sizes smaller and reduces complexity from data type conversions.

As mentioned above, we specifically recommend using Immer if you want to simplify immutable update logic, specifically as part of Redux Toolkit.

▶ **Detailed Explanation**

Priority C Rules: Recommended

Write Action Types as `domain/eventName` RECOMMENDED

The original Redux docs and examples generally used a "SCREAMING_SNAKE_CASE" convention for defining action types, such as `"ADD_TODO"` and `"INCREMENT"`. This matches typical conventions in most programming languages for declaring constant values. The downside is that the uppercase strings can be hard to read.

Other communities have adopted other conventions, usually with some indication of the "feature" or "domain" the action is related to, and the specific action type. The NgRx community typically uses a pattern like `"[Domain] Action Type"`, such as `"[Login Page] Login"`. Other patterns like `"domain:action"` have been used as well.

Redux Toolkit's `createSlice` function currently generates action types that look like `"domain/action"`, such as `"todos/addTodo"`. Going forward, **we suggest using the `"domain/action"` convention for readability**.

Write Actions Using the Flux Standard Action Convention RECOMMENDED

The original "Flux Architecture" documentation only specified that action objects should have a `type` field, and did not give any further guidance on what kinds of fields or naming conventions should be used for fields in actions. To provide consistency, Andrew Clark created

a convention called "Flux Standard Actions" early in Redux's development. Summarized, the FSA convention says that actions:

- Should always put their data into a `payload` field
- May have a `meta` field for additional info
- May have an `error` field to indicate the action represents a failure of some kind

Many libraries in the Redux ecosystem have adopted the FSA convention, and Redux Toolkit generates action creators that match the FSA format.

Prefer using FSA-formatted actions for consistency.

> **Note**: The FSA spec says that "error" actions should set `error: true`, and use the same action type as the "valid" form of the action. In practice, most developers write separate action types for the "success" and "error" cases. Either is acceptable.

Use Action Creators RECOMMENDED

"Action creator" functions started with the original "Flux Architecture" approach. With Redux, action creators are not strictly required. Components and other logic can always call `dispatch({type: "some/action"})` with the action object written inline.

However, using action creators provides consistency, especially in cases where some kind of preparation or additional logic is needed to fill in the contents of the action (such as generating a unique ID).

Prefer using action creators for dispatching any actions. However, rather than writing action creators by hand, **we recommend using the `createSlice` function from Redux Toolkit, which will generate action creators and action types automatically**.

Use Thunks for Async Logic RECOMMENDED

Redux was designed to be extensible, and the middleware API was specifically created to allow different forms of async logic to be plugged into the Redux store. That way, users wouldn't be forced to learn a specific library like RxJS if it wasn't appropriate for their needs.

This led to a wide variety of Redux async middleware addons being created, and that in turn has caused confusion and questions over which async middleware should be used.

We recommend using the Redux Thunk middleware by default, as it is sufficient for most typical use cases (such as basic AJAX data fetching). In addition, use of the `async/await` syntax in thunks makes them easier to read.

If you have truly complex async workflows that involve things like cancelation, debouncing, running logic after a given action was dispatched, or "background-thread"-type behavior, then consider adding more powerful async middleware like Redux-Saga or Redux-Observable.

Move Complex Logic Outside Components RECOMMENDED

We have traditionally suggested keeping as much logic as possible outside components. That was partly due to encouraging the "container/presentational" pattern, where many components simply accept data as props and display UI accordingly, but also because dealing with async logic in class component lifecycle methods can become difficult to maintain.

We still encourage moving complex synchronous or async logic outside components, usually into thunks. This is especially true if the logic needs to read from the store state.

However, **the use of React hooks does make it somewhat easier to manage logic like data fetching directly inside a component**, and this may replace the need for thunks in some cases.

Use Selector Functions to Read from Store State RECOMMENDED

"Selector functions" are a powerful tool for encapsulating reading values from the Redux store state and deriving further data from those values. In addition, libraries like Reselect enable creating memoized selector functions that only recalculate results when the inputs have changed, which is an important aspect of optimizing performance.

We strongly recommend using memoized selector functions for reading store state whenever possible, and recommend creating those selectors with Reselect.

However, don't feel that you *must* write selector functions for every field in your state. Find a reasonable balance for granularity, based on how often fields are accessed and updated, and how much actual benefit the selectors are providing in your application.

Name Selector Functions as `selectThing` RECOMMENDED

We recommend prefixing selector function names with the word `select`, combined with a description of the value being selected. Examples of this would be `selectTodos`, `selectVisibleTodos`, and `selectTodoById`.

Avoid Putting Form State In Redux RECOMMENDED

Most form state shouldn't go in Redux. In most use cases, the data is not truly global, is not being cached, and is not being used by multiple components at once. In addition, connecting forms to Redux often involves dispatching actions on every single change event, which causes

performance overhead and provides no real benefit. (You probably don't need to time-travel backwards one character from `name: "Mark"` to `name: "Mar"`.)

Even if the data ultimately ends up in Redux, prefer keeping the form edits themselves in local component state, and only dispatching an action to update the Redux store once the user has completed the form.

There are use cases when keeping form state in Redux does actually make sense, such as WYSIWYG live previews of edited item attributes. But, in most cases, this isn't necessary.