**React and UI State libraries optimization**

[[Kevin Wong](https://pragmaticgeek.medium.com/?source=post_page-----bf9a1103a71d--------------------------------)](https://pragmaticgeek.medium.com/?source=post_page-----bf9a1103a71d--------------------------------)

[Kevin Wong](https://pragmaticgeek.medium.com/?source=post_page-----bf9a1103a71d--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F36636f636b9e&operation=register&redirect=https%3A%2F%2Fpragmaticgeek.medium.com%2Freact-and-ui-state-libraries-optimization-bf9a1103a71d&user=Kevin+Wong&userId=36636f636b9e&source=post_page-36636f636b9e----bf9a1103a71d---------------------post_header-----------)

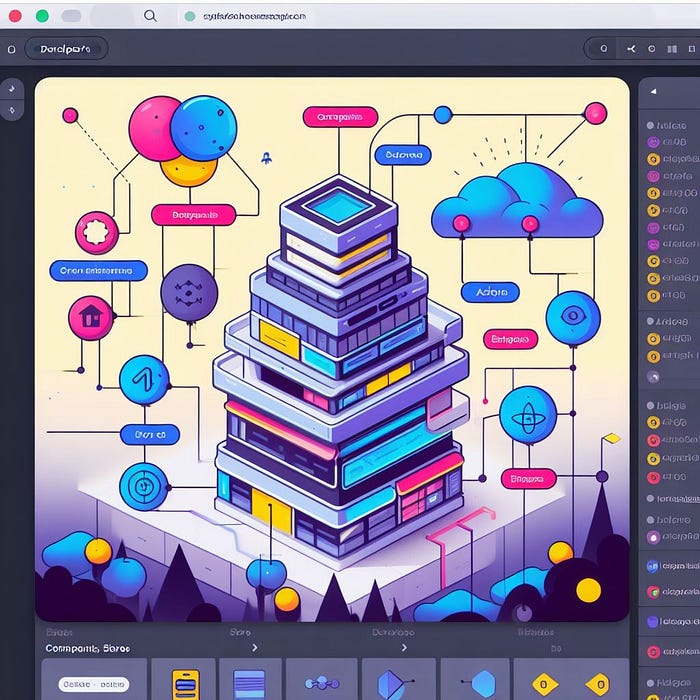
7 min read

·

Dec 6

51

1



Building a modern web application invariably involves managing UI state. As applications evolve, the complexity of UI state and its sub-states often grows, leading to performance issues and potential bugs if not carefully planned. In the React development community, Redux, once the undisputed UI state libraries of choice, but now has faced criticism for its bulkiness, maintainability challenges, and re-rendering performance concerns. In response, React developer community has sought alternatives that improve on performance and optimization.

There has been alternative libraries and patterns since then like:

* mobx
* RxJS
* Recoil
* PushState
* React’s native *useReducer* hook
* React’s native *useContext* and *useState* combination hooks

Each of them have drawbacks such as bundle size of the libraries or the efficiency of the library. More-so, in the native React hooks, they solve a smaller problem at a smaller scale but once scaled up to a complex application, it becomes very challenging to manage the state using just the native hooks.

After looking at what some alternatives, two libraries that caught my attention:

* **Zustand** (<https://github.com/pmndrs/zustand>)

*A small, fast and scalable bearbones state-management solution using simplified flux principles. Has a comfy API based on hooks, isn’t boilerplatey or opinionated.*

* **Jotai** (<https://jotai.org/docs/extensions/redux>)

*Jotai scales from a simple useState replacement to an enterprise TypeScript application.*

What does **Zustand** and **Jotai** solve that Redux or the alternatives doesn’t solve?

* aimed to be minimalistic to solve merged UI State
* atomic state and segregation of finer state notification (transient update) by design that leads to better re-rendering performance
* framework/library agnostic

What appeals to me about this concept is its ability to atomically segregate each piece of UI state for updates as required. For instance, while working on a significant feature for work, I found that each sub-feature only required a subset of the UI application state. These subsets can be treated atomically in relation to the domain model or subdomain models.

As the application scaled, a remarkable realization unfolds — the need for a composed molecule state derived from atomic states to map data dependencies. This essentially transforms atomic states, much like Lego pieces, into a larger molecule state that precisely corresponds to all the data and data-dependency mapping required for the feature module. This proves to be a powerful concept, especially in the context of microfrontend architecture. In UI applications, the *View* component is often decomposed into finer atomic Design System components and assembled into composite components. However, the *Model* part of the UI system tends to be neglected, resulting in a cumbersome pile of monolithic tech debt.

*Jotai* excels in the realm of building atomic to molecule state, particularly as web front-end engineers adopt an iterative approach towards completed applications. *Jotai* introduces the concept of building more granularly, eliminating the need to design the entire UI state upfront. Instead, it allows developers to understand how to selectively separate the state and data relationships as they progress. In contrast, the *Redux* approach feels aligned to a waterfall model, potentially resulting in significant time loss from an engineering standpoint where the UI state must be planned upfront to have success. Moreover, using the *useSelector* hook with *Redux* is not without penalties as the system scales up, leading to unforeseen maintenance nightmares due to cherry-picked state scattered throughout the application codebase. Deleting or mutating the structure of the UI state can pose significant problems in the future. The atomic/separation of state update (in for both *Zustand* and *Jotai*) offers an effective alternative, enabling developers to work in reverse and replace monolithic UI state solutions such as *Redux* or older *Flux* libraries.

**Demo**

I have prepared a Codesandbox demo of *Redux* vs *Jotai* vs *Zustand* vs *native React hooks* to showcase UI state performance. Make sure to open the console to see what is updating an re-rendering to understand the performance impact of each option.

Codesandbox Demo

Github link to the demo

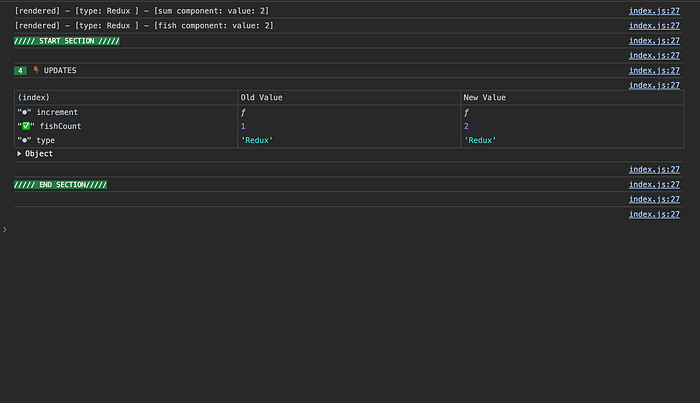
**[GitHub - pragmaticgeek/UI-State-Demo: The purpose of this repository is to provide a demo to…](https://github.com/pragmaticgeek/UI-State-Demo?source=post_page-----bf9a1103a71d--------------------------------" \t "_blank)**

[The purpose of this repository is to provide a demo to compare UI state libraries and strategies. As well as exploring…](https://github.com/pragmaticgeek/UI-State-Demo?source=post_page-----bf9a1103a71d--------------------------------" \t "_blank)

[github.com](https://github.com/pragmaticgeek/UI-State-Demo?source=post_page-----bf9a1103a71d--------------------------------" \t "_blank)

**Redux**

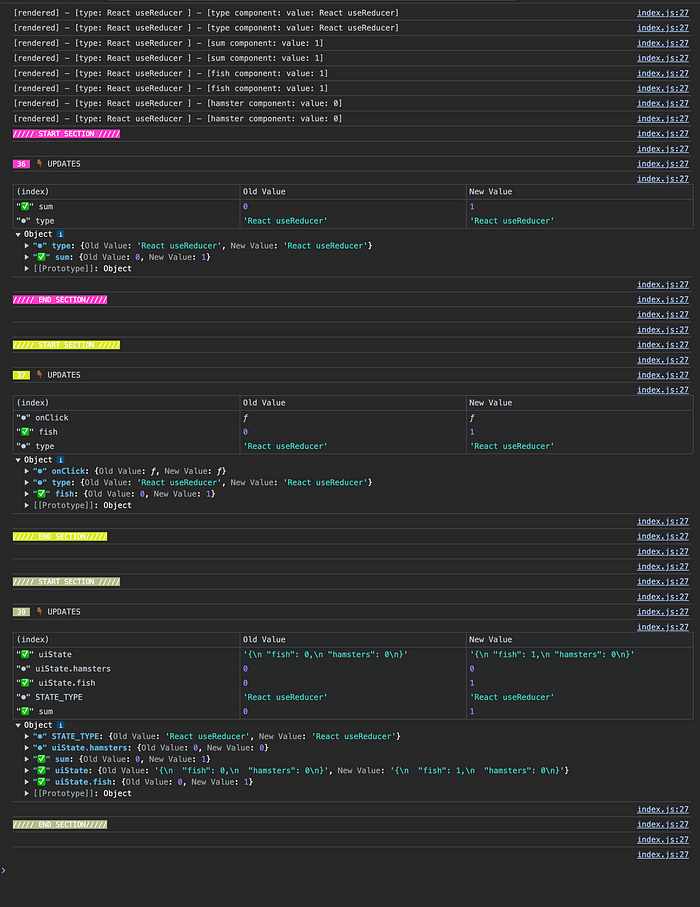
As you can see from the demo, by using *useSelector*hook, we can target what gets updated. So you can see when fish is incremented, Redux will first render with current state value again and then new changes will cause another re-render. It does work as intended to reduce re-rendering. But as mentioned, as the system grows, the *useSelector* hook usage will be sprinkled to many parts of the codebase with other developers having very little insight of the impact while mutating the state structure or deleting a sub-state from the reducer.



Redux with useSelector hook to separate the state updates

**useReducer hook**

This is probably the second worst option evaluated. Incrementing either fish or hamsters not only causes itself to change but also to *TypeComponent, SumComponent*and both the *FishComponent* and *HamsterComponent.*



useReducer hook re-rendering

From the console output, you can see my useReducer implementation is setting a new object that is returned for every time the state mutates in the reducer. Perhaps using ImmutableJS will help reduce an object reference update but that is an extra library to tack on with and more complexity added on to solve the performance problem. So as the system grows, this option is not very desirable.

const reducer = (state, action) => {  
 switch (action.type) {  
 case actions.HAMSTERS\_INCREMENT:  
 return { ...state, hamsters: state.hamsters + 1 };  
 case actions.FISH\_INCREMENT:  
 return { ...state, fish: state.fish + 1 };  
 default:  
 return state;  
 }  
};

**useContext with useState combination**

This one is probably the worst one evaluated. Very similar to the above *useReducer* implementation where the stored state gets updated as a new object, this implementation isn’t easily solved by using ImmutableJS to retain the same object reference. As you can see from the console, even the callbacks references are different causing an update causing re-rendering to happen. Thus, whenever any value within the state changes, so will all the downstream state. This is a pattern I have seen many React developer use but fail to understand the heavy impact that it comes with.



useContext with useState pattern causing multiple re-rendering

**Zustand**

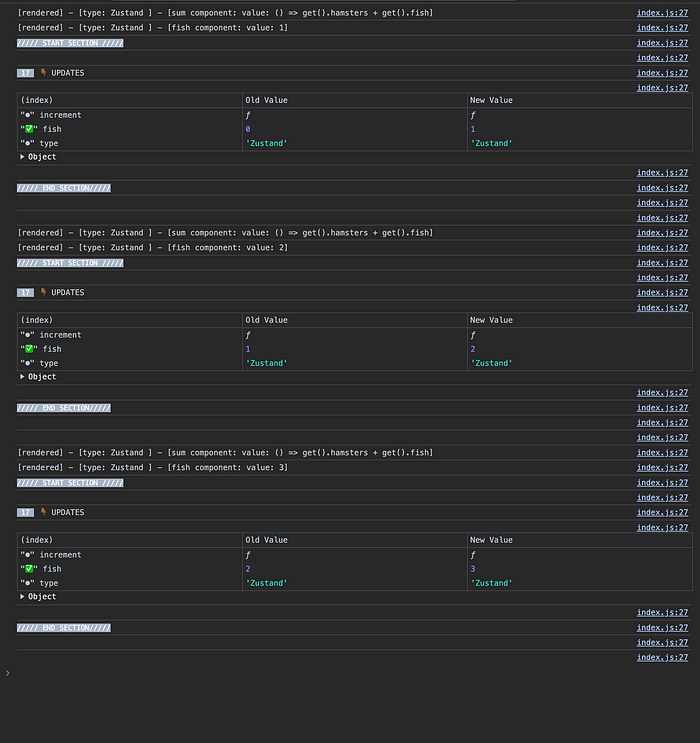
Like *Redux*, *Zustand* is almost a direct replacement of *Redux.* Zustand encourages the [use of multiple sliced stores](https://github.com/pmndrs/zustand/blob/main/docs/guides/slices-pattern.md) instead of one giant store like traditional flux state design to achieve modularity. Moreover, the sliced stores can be combined to form a more complex bounded (composite) store.

For example from *Zustand’s* documented slices pattern:

import { create } from 'zustand'  
import { createBearSlice } from './bearSlice'  
import { createFishSlice } from './fishSlice'  
  
export const useBoundStore = create((...a) => ({  
 ...createBearSlice(...a),  
 ...createFishSlice(...a),  
}))

This gives the developer a finer control of atomic states and the bounded store can be where data dependency mapping can be captured and be understood by other developers working on a common UI state.

Looking at the performance. I expected no less than only updating the targeted component with the updated pieces of state



Zustand re-rendering on selected state changes

**Jotai**

Like Zustand, when we add a fish or add a hamster to the store, only the Sum component and the count of either fish or hamster gets updated. This is because Jotai only transiently update the component when fish or hamster atom state changes and everything else are not affected from being re-rendered. So as the UI state gets more complicated, working from the ground up to build composite state is ideal. What differs from Zustand is that Jotai encourages to go bottom-up by defining atom states and build backwards up to a composite state that becomes the global UI state.

As expected, triggering a change to either Fish or Hamster only affected the counter itself and the sum to be re-rendered and nothing else is changed like *Zustand*or *Redux* with *useSelector* hook.



Jotai re-rendering on atomic state changes

**Final thoughts**

So now you are convinced with the performance gains from segregated state update approach. So what is better? **Jotai** or **Zustand**?

**Zustand**

* use *Zustand* if you are migrating an application that is already in Redux or Flux inspired UI state libraries, you will have a better time with the top down state approach and API similar to Redux and flux
* When migrating from Redux, no more Provider is needed or need for a Context/Provider HOC pattern this is a big plus and the state can be accessible through React hooks anywhere

**Jotai**

* consider using this if you are green-fielding a new project and iteratively building out the project.
* state can be define atomically from the start.
* lots of extensions for the heavy lifting boilerplate code