iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 11 **Issue:** IX **Month of publication:** September 2023

DOI: https://doi.org/10.22214/ijraset.2023.55677

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 11 Issue IX Sep 2023- Available at www.ijraset.com

A Comparative Study of State Management Libraries: Redux, MobX, Recoil, and Zustand

Jai Sehgal[1], Akash Yadhav[2]
[1] *Frontend Developer, Gurugram, India*
[2] *Software Developer, Gurugram, India*

Abstract: ***State management is an essential concern in the development of modern web applications. Various libraries have emerged to ease this problem, each with its own philosophy and implementation details. This paper provides a comprehensive evaluation and comparison of four prominent state management libraries: Redux, MobX, Recoil, and Zustand. We scrutinize each library in terms of usability, complexity, code maintainability, and performance, thereby providing developers with insights to make an informed choice.***

Keywords: State Management, Redux, MobX, Recoil, Zustand, Usability, Complexity, Performance, Code, Maintainability, Web Development

I. INTRODUCTION

The rapid evolution of web applications has rendered state management as one of the focal points in front-end development. Managing the state effectively is crucial for scalability, maintainability, and performance. Four state management libraries stand out due to their unique approaches and widespread adoption: Redux, MobX, Recoil, and Zustand.

II. BACKGROUND

Modern web development has witnessed exponential growth in complexity, driven by user demands for fast, interactive, and visually engaging applications. As developers strive to meet these demands, the need for effective state management has never been greater. The frontend state in a web application refers to the data that dictates the behaviour and appearance of the interface. It could include anything from simple UI flags, like whether a dropdown menu is open, to complex data sets retrieved from APIs. Effective state management is critical to ensuring that an application is not only functional but also efficient, reliable, and user-friendly. This underscores the significance of choosing an appropriate state management library—one that aligns well with the project's specific requirements.

III. SIGNIFICANCE

With the array of state management solutions available, making an informed choice is often daunting for developers. A wrong decision can lead to increased complexity, hamper maintainability, and degrade performance, thereby negatively affecting the user experience and development velocity. By providing a detailed comparison of popular state management libraries, this paper aims to serve as a guide for developers, aiding them in choosing a library that best suits their application's needs.

IV. LIBRARY OVERVIEW

To ensure a fair and comprehensive comparison, this study employs a multi-faceted approach. Firstly, a theoretical examination of each library's architecture and principles is undertaken. Secondly, sample code snippets are analyzed to highlight the intricacies of practical implementation. Finally, real-world performance benchmarks and case studies are referenced to present a holistic view.

V. PERFORMANCE EVALUATION

A. Redux

Redux is a predictable state container designed to help you write JavaScript applications that behave consistently across different environments. It promotes a single immutable state tree and a unidirectional data flow.

B. MobX

MobX uses reactive programming and is less opinionated about application structure. It provides a more intuitive way to deal with state management, making it quicker to implement.

C. Recoil

Recoil is developed by Facebook and aims to address the limitations of existing tools while adding features like derived state, asynchronous queries, and cross-app observability.

D. Zustand

Zustand offers a minimalistic approach focusing on simplicity and a small bundle size. It combines the best features from its competitors and aims to reduce boilerplate code.

VI. USABILITY

Managing the state effectively in a web application not only aids in achieving the desired functionality but also plays a crucial role in enhancing usability. This section provides an in-depth look into the usability aspects of four state management libraries: Redux, MobX, Recoil, and Zustand.

A. Redux

1) Pros

a) Mature Ecosystem: Redux boasts a mature and robust ecosystem. The library is well-documented, and there is an abundance of community-contributed middleware, tutorials, and boilerplate projects. This rich ecosystem can dramatically expedite the development process.

b) Predictability and Debugging: The predictable nature of Redux's state transitions, combined with its powerful debugging tools like "Redux DevTools," offers an intuitive debugging experience. You can trace every state change back to the action that caused it, allowing for easier testing and debugging.

2) Cons

a) Boilerplate Code: One of the biggest criticisms of Redux is the amount of boilerplate code required to implement even simple features. For small to medium projects or for developers just starting out, this can be overwhelming and can extend the development timeline.

b) Learning Curve: Redux requires understanding several concepts like reducers, actions, action creators, and middleware. This makes the library less approachable for beginners, who might find the plethora of terms and the overall architecture challenging to grasp.

B. MobX

1) Pros

a) Intuitive API: MobX offers an API that is straightforward and easy to learn. The observable and computed values in MobX significantly reduce boilerplate code, which is often required in other state management libraries.

b) Flexibility: Unlike Redux, MobX is less prescriptive about how to structure your application, providing developers more freedom. This can be advantageous for smaller teams and projects where quick prototyping is necessary.

2) Cons

a) Lack of Structure: The flexibility that MobX offers can be a double-edged sword. In large and complex applications, the lack of a strict pattern can lead to inconsistencies and make the codebase difficult to manage.

b) Debugging Complexity: While MobX allows for more dynamic interactions, this can complicate the debugging process as it might not be as straightforward to trace how a particular state change occurred.

C. Recoil

1) Pros

a) Fine-Grained Control: Recoil provides a more granular level of control over the state, facilitating direct manipulation of smaller state pieces. This allows for more optimized re-rendering and overall better performance.

b) *Built-in Asynchronous Handling:* Recoil supports asynchronous data queries and derived state out-of-the-box, simplifying tasks like data fetching and manipulation.

2) Cons

a) *Community Resources:* Being relatively new, Recoil has fewer community resources and tutorials available. This could be a hindrance for developers who rely on community support for learning and troubleshooting.

b) *Lack of Middleware Support:* Unlike Redux, Recoil does not have a built-in middleware support system, which can make implementing custom logging, caching, or other side-effects more challenging.

D. Zustand

1) Pros

a) *Simplicity:* Zustand offers a minimalistic API that is extremely easy to set up. It is suitable for projects where you want to get state management up and running quickly without the need to learn a lot of new concepts.

b) *Low Overhead:* The library has a small bundle size and almost no boilerplate code, making it a suitable choice for lightweight applications or for incorporating into projects incrementally.

2) Cons

a) *Limited Built-In Features:* Zustand is minimalistic by design, which means that it doesn't offer some of the more advanced features of its competitors out of the box. For complex state logic, additional custom coding may be required.

b) *Scalability:* While Zustand is perfect for small to medium-sized applications, its lack of structure and guidelines can make it less suitable for larger projects, where codebase maintainability becomes critical.

VII. COMPLEXITY

Complexity in state management libraries often correlates with the library's learning curve, the amount of boilerplate code, and the intellectual overhead required to manage application state effectively. Here, we dissect the complexity aspects of Redux, MobX, Recoil, and Zustand, supported by code examples for better understanding.

A. Redux

In Redux, the core principles are actions, reducers, and the store. While this promotes a clean architecture and makes the state predictable, it tends to increase complexity, especially for straightforward state changes.

1) Code Comparison

For adding an item to a list in Redux:

```
// Action
const ADD_ITEM = "ADD_ITEM";

export const addItem = item => ({
  type: ADD_ITEM,
  payload: item,
});

// Reducer
const initialState = {
  items: [],
};

export const itemReducer = (state = initialState, action) => {
  switch (action.type) {
    case ADD_ITEM:
      return { ...state, items: [...state.items, action.payload] };
    default:
```

```
    return state;
  }
};

// Store
import { createStore } from "redux";

const store = createStore(itemReducer);
```

2) Complexity Analysis

a) Boilerplate Code: As seen in the code example, the process of adding an item involves creating an action, a reducer, and finally dispatching the action through the store. This involves multiple files and increases the boilerplate code.

b) Immutability: The Redux architecture demands that reducers be pure functions, necessitating manual copying of states, which adds to the complexity.

B. MobX

MobX operates on a simpler principle of observables and actions, making it comparatively less complex for certain tasks.

1) Code Comparison

For adding an item to a list in MobX:

```
import { observable, action } from "mobx";

class ItemStore {
  @observable items = [];

  @action
  addItem(item) {
    this.items.push(item);
  }
}

const store = new ItemStore();
```

2) Complexity Analysis

a) Intuitive Syntax: The MobX example uses decorators to mark fields as observable and methods as actions, streamlining the code.

b) Implicit State Management: The complexity is reduced as MobX takes care of the immutability behind the scenes.

C. Recoil

Recoil works with atoms and selectors, allowing more granular control over the state.

1) Code Comparison

For adding an item to a list in Recoil:

```
import { atom, useRecoilState } from "recoil";

const itemState = atom({
  key: "itemState",
  default: [],
});

// In a React component
function ItemList() {
```

```
  const [items, setItems] = useRecoilState(itemState);

  const addItem = (item) => {
    setItems([...items, item]);
  };
}
```

2) Complexity Analysis

a) Fine-Grained Control: Atoms make it easier to manage specific slices of state, thereby reducing complexity for focused tasks.

b) Learning New Concepts: While atoms and selectors are powerful, they introduce new concepts that developers must learn, adding to initial complexity.

D. Zustand

Zustand aims for simplicity and minimum boilerplate, resulting in low complexity.

1) Code Comparison

For adding an item to a list in Zustand:

```
import create from "zustand";

const useStore = create((set) => ({
  items: [],
  addItem: (item) => set((state) => ({ items: [...state.items, item] })),
}));

// Usage
const addItem = useStore((state) => state.addItem);
```

2) Complexity Analysis:

a) Minimalistic API: Zustand's code example clearly shows a minimalistic approach that reduces boilerplate code.

b) Lack of Structure: The simplicity comes at a cost, as Zustand provides less guidance and structure, which could lead to complexities in large applications.

VIII. CODE QUALITY AND MAINTAINABILITY

The efficiency and success of a software project significantly rely on the quality of the codebase and how easily it can be maintained and extended. Code quality often encompasses readability, modularity, and the ease with which team members can understand and collaborate on the code. Maintainability involves considerations like how easily bugs can be identified and fixed, how straightforward it is to implement new features, and the effort required for debugging. In this section, we will examine these aspects in the context of four state management libraries: Redux, MobX, Recoil, and Zustand.

A. Redux

1) Code Quality

a) Structured and Explicit: Redux encourages a structured and explicit code architecture. The state transition logic is centrally located within reducers, making it easier to understand the overall flow and functionality.

b) Type-Safe: With the strong community adoption of TypeScript, Redux integrates seamlessly, providing a type-safe way to manage states. This improves code quality by identifying issues at compile-time.

2) Maintainability

a) Debugging Tools: Redux offers excellent debugging capabilities, especially with middlewares like Redux DevTools. This makes it easier to maintain by offering features like time-travel debugging.

b) Scalability: Redux scales well for large projects with multiple developers. Its strict architecture ensures a uniform codebase, making it easier to onboard new developers and maintain the code.

B. MobX

1) Code Quality

a) Readable Syntax: MobX code is often more concise and, thus, more readable. The use of observables and computed values can make the state transitions easily understandable.

b) Dynamic: MobX allows for more dynamic coding patterns, which can lead to elegant solutions but also potential pitfalls if not managed carefully.

2) Maintainability

a) Simplicity: MobX's simple and intuitive API makes it easy to add or change features, contributing to better maintainability.

b) Refactoring Risks: The freedom and flexibility that MobX provides can be a double-edged sword. Inconsistent implementations among team members can make the codebase harder to maintain.

C. Recoil

1) Code Quality

a) Component-Level State: Recoil's atom-based state management allows for very granular control, often within individual components. This ensures that components only re-render when absolutely necessary, which can improve performance and code quality.

b) Async Handling: Built-in asynchronous utilities make dealing with asynchronous operations clean and manageable.

2) Maintainability

a) High Cohesion: Recoil encourages localized state management, improving cohesion and making it easier to understand the functionality of individual components.

b) Limited Community Patterns: Being a newer library, community best practices are not as well-established, which might lead to varied coding patterns and thereby affecting maintainability.

D. Zustand

1) Code Quality

a) Minimal Boilerplate: With Zustand, you can achieve functionality with less code. The minimalistic approach often results in cleaner, less cluttered codebases.

b) Immutability: Zustand doesn't enforce immutability, allowing you to manage it as you see fit, which can be positive for code quality if managed carefully.

2) Maintainability

a) Simplicity for Small Projects: Zustand's simplicity makes it easy to maintain smaller projects, but it could be a limitation for larger, more complex applications.

b) Lack of Formal Structure: The lack of a rigid structure and guidelines could lead to inconsistencies in code, particularly in projects with multiple developers, thereby affecting maintainability.

IX. CONCLUSION

Each of the state management libraries offers unique advantages and disadvantages. Redux provides a structured and maintainable approach, MobX offers simplicity and quick development, Recoil allows granular state management with performance benefits, and Zustand offers minimalistic simplicity. The choice of library should be dictated by the project's specific requirements, including its complexity, the team's familiarity with the libraries, and specific performance considerations.

Choosing the right state management library requires a deep understanding of the project requirements and the trade-offs involved. It is hoped that this comparative study offers comprehensive insights to make an informed decision.

X. ACKNOWLEDGMENT

I would like to extend my sincere gratitude to all those who contributed to the success of this research paper. First and foremost, I thank my advisors for their continuous guidance and invaluable insights that shaped this work. My appreciation also goes to my peers and colleagues who reviewed the paper and offered constructive criticism.

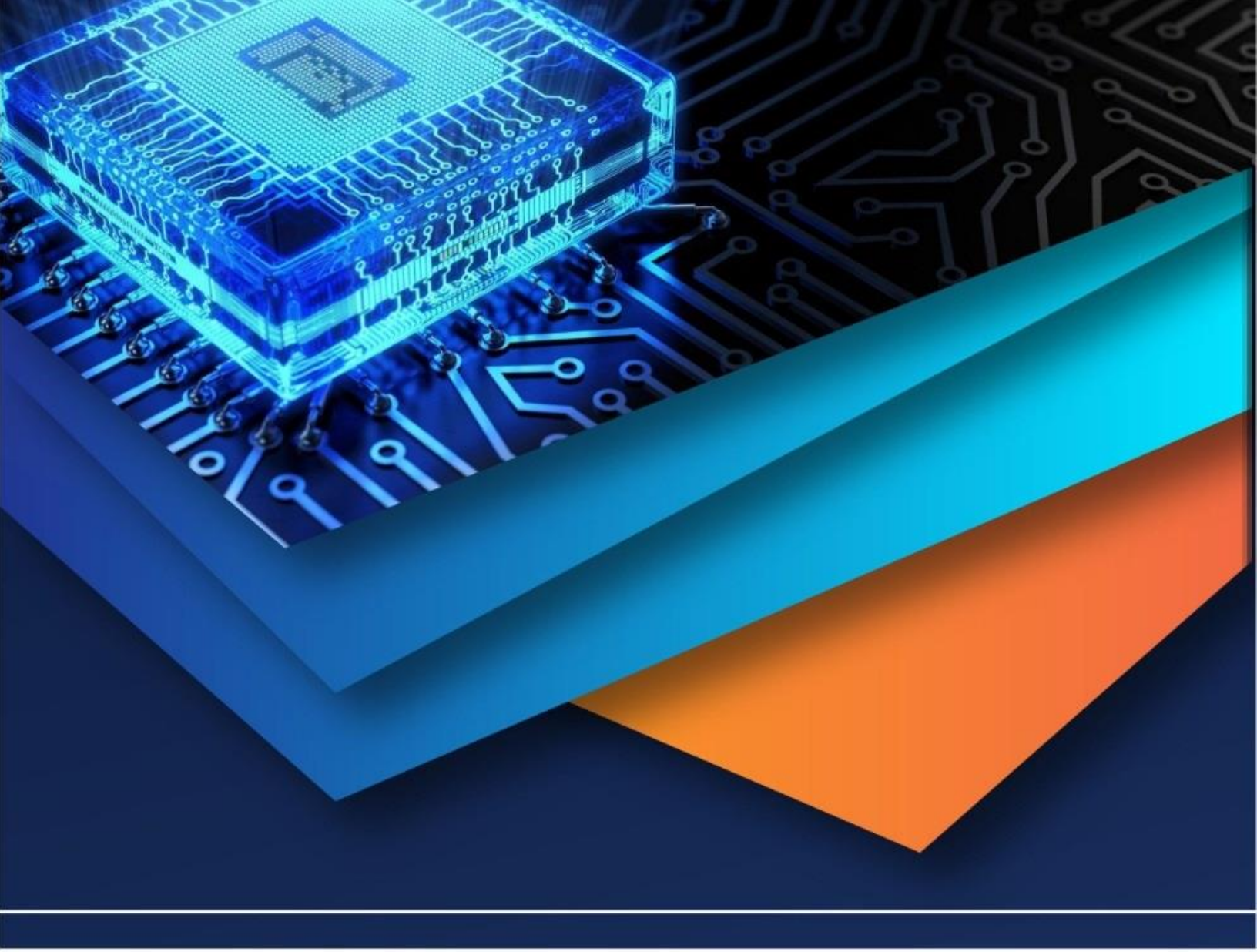
A special mention must be made to the developers and communities behind Redux, MobX, Recoil, and Zustand; their open-source contributions made this comparative study possible. Last but not least, I am grateful to the various online forums and educational resources that aided in my understanding of state management libraries. Thank you all for your support and encouragement.

REFERENCES

[1] Redux Official Documentation: https://redux.js.org/
[2] MobX Official Documentation: https://mobx.js.org/
[3] Recoil Official Documentation: https://recoiljs.org/
[4] Zustand Official Documentation: https://github.com/pmndrs/zustand

ABOUT THE AUTHORS

	**Jai Sehgal,** a passionate software engineer holding a Bachelors in Technology in Computer Science and Engineering. who is passionate about working with new tech in the market and also published another paper on topic Image Noise Reduction with Autoencoder using Tensor FlowImage Noise Reduction with Autoencoder using Tensor Flow International Journal of Science and Research (IJSR) · Oct 1, 2020
	**Akash Yadav** Software Engineer with a Bachelors in Technology on Computer Science and Engineering who likes to work on data and related fields. Has published another paper on Real-Time Image Processing using Flutter and Tflite Packages.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY