Introduction

State management is a central challenge in modern React applications. As projects grow in size and complexity, managing shared data, side effects, and predictable updates becomes critical. React provides built-in tools such as useState and useReducer for local component state. However, for large-scale applications, Redux offers a centralized and predictable state container that improves scalability, debugging, and team collaboration.

This documentation explores when to use Redux instead of React's built-in tools, the tradeoffs between simplicity and scalability, and the role of middleware in handling side effects and asynchronous operations.

Redux vs React's Built-in State Management

Redux over useState or useReducer

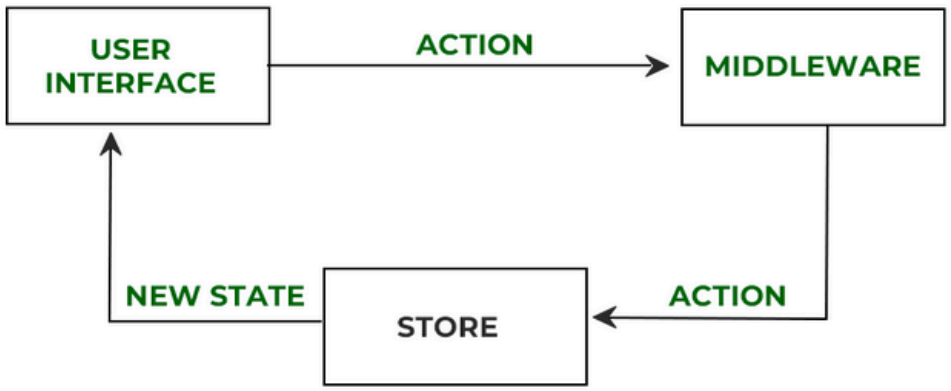
Redux becomes the preferred choice over useState or useReducer when an application grows beyond simple component-level state and starts requiring shared data across multiple parts of the UI. While React's built-in hooks are great for managing local state with minimal setup, they can quickly become limiting in larger apps. Prop drilling, scattered logic, and performance issues from unnecessary re-renders are common challenges when scaling with hooks alone. Redux solves these problems by centralizing state in a single store, making updates predictable and easier to trace. It also supports middleware, which allows for clean handling of side effects like API calls or logging. For teams working on complex applications, Redux offers better structure, debugging tools, and long-term maintainability, even if it comes with a steeper learning curve at first.

Trade-offs in complexity vs scalability.

There's a clear trade-off between simplicity and scalability when choosing a state management approach. Redux introduces more complexity upfront because it requires boilerplate, a deeper understanding of patterns, and a steeper learning curve. However, this investment pays off in larger applications where scalability, maintainability, and predictable state flow become essential. On the other hand, useState and useReducer offer a faster and simpler setup, making them ideal for small components or lightweight applications. But as the app grows, this approach can lead to prop drilling and performance issues due to unnecessary re-renders, especially when state updates become frequent or deeply nested.

Approach	Pros	Cons	Best For
useState/useReducer	Simple setup, minimal boilerplate, great for local state	Hard to scale, prone to prop drilling	Small or medium apps, isolated components
Redux	Centralized state, predictable flow, powerful debugging	More setup and conceptual overhead	Large apps, shared or async state management

Middleware in Redux
What role does middleware (e.g., Redux Thunk) play?
Middlewares are essential for handling side effects and enhancing the functionality of Redux. They are used to intercept actions sent to the Redux store and modify them before they reach the reducer or after they are dispatched. Allows Redux to stay synchronous as its core, while extending its capabilities for real-world applications.



Real-world scenarios where middleware is necessary.
We can add a small middleware if we want to add some logging to our application that lets us see the content of each action in the console when it's dispatched, and see what the state is after the action has been handled by the reducers.

In the code, a middleware has the next structure:

```
export const myMiddlware = (api) => (next) => (action) => {
  /* do something */
};
```

A middleware can do anything it wants when it sees a dispatched action:

- Log something to the console
- Set timeouts
- Make asynchronous API calls
- Modify the action
- Pause the action or even stop it entirely

In particular, middleware is intended to contain logic with side effects. In addition, middleware can modify dispatch to accept things that are not plain action objects.

Conclusion
Before this research, I was not completely sure when it was better to use useReducer or Reedux. I learned that as long as the application is not handling complex state logic, it is fine to keep using React's built-in tools like useState or useReducer.
I also learned about real examples of when middleware is useful in Redux and how it helps manage side effects, such as API calls or logging, in a cleaner and more organized way.

And, it’s important to understand that Redux is not always necessary, but it becomes a strong and scalable solution when an application grows and needs better structure, predictability, and teamwork.

References

- React state management - useReducer vs Redux | Saeloun Blog
- Bridging the Gap between React's useState, useReducer, and Redux | Strings and Things
- What are Middlewares in React Redux? - GeeksforGeeks
- Redux Fundamentals, Part 4: Store
- Middleware | Redux
- React Hooks vs Redux - GeeksforGeeks
- React Hooks Does Not Replace Redux: When to Use Which, or Both? | by Zavagezong | Medium
- ¿Que es un middleware en redux? - DEV Community