TL;DR

We have started a series of contributions to the Aave IPFS UI to improve the state management system of the application.

Context

Usually, the contributions of BGD on Aave are linked to smart contracts, architecture/design, and on-chain security aspects. But actually, other parts of Aave like the front end are quite important too.

Often overlooked, blockchain front-end applications like the Aave IPFS one are in a quite special position within the JS ecosystem, so we were/are trying to define particular issues and find modern solutions and approaches to make it more performant, and easier to maintain/contribute to. This is especially important for us at the moment, as some of the upcoming features involve changes on the front end, so by contributing to it, the delivery process is faster.

The problem

The current Aave IPFS user interface, although following modern React best practices, has some issues:

- Its extensive use of context makes it hard to separate visual components from business logic.
- It is hard to split business logic into multiple files since it's mostly located in the context's provider files.
- Due to the extensive use of hooks, it is kind of hard to compose multiple actions together without triggering React state changes and re-renders; which is not optimal performance-wise.

This list of issues is not unique to the Aave interface; it is a consequence of current trends of front-end development trying to reduce the client-side complexity by offloading the logic back to the server. React server components, Apollo Graphql, and similar tools are good examples of this trend.

But the typical Web3 application has a completely opposite set of requirements, i.e:

- IPFS deployment build, which should work without a server.
- Complicated logic for fetching and mapping the data from blockchain's RPC. Combining data from different smart contract methods, combining freshly fetched data with cache provided from a different source.
- Wallet-related logic: different types of wallets, signing, network switching, etc.

And that's where the bulk of the app complexity goes. The UI aspect, although important, is usually way less complicated than all the above.

The solution

Our strategy was to research what modern React state management solutions have to offer and see if it's possible with minimal effort to develop an easy-to-follow and simple architecture, with the following requirements:

1. The global store is separated from component states, with the option to select only a portion of the state to avoid additional re-renders when fetching invisible data.
2. Dedicated actions place. Most of the logic happens in side effects, so we need a way to always have dedicated side effects with options to chain multiple ones.
3. Minimalistic boilerplate with easy onboarding. The solution should be as default and generic as possible, so we can rely on library docs and learning materials.

Generally, there are 3 approaches to state management

Atomic state

(Jotai, Recoil)

Good

Very performant for dealing with computed state. Powerful selectors with automatic caching.

Bad

No place for side effects. Atomic libs were mostly invented for dealing with frequently changed user states. i.e. Text editors, graphics, canvas, etc. Each action is either the default state of an atom or a reaction for some atom change. No way to

chain events.

Proxy pattern

(Valtio, Mobx)

Good

The same benefit of single direction data flow, but without hustle with immutable state and selectors.

Powerful API, with generators in actions, transactional state changes (no intermediate changes).

Bad

Complex API, observable implementation is different in each lib, steep learning curve. Classes and HOC, most of the API moved to hooks.

Potential problems with new react versions and SSR Support concurrent features React / React 18 · Issue #2526 · mobxjs/mobx

Flux

(Zustand, Storeon, RTK)

Good

Action-based and very reliable and scalable.

Bad

No transactional state. Intermediate updates would flicker.

Boilerplate and immutability.

So, which one for Aave?

Atomic state is clearly trying to solve another problem, with the lack of global actions flow being a dealbreaker.

The proxy approach is awesome, but not simple. The implementation of an observable pattern is always different and although it does suites us well, the learning curve is steep.

Flux with Zustand is the winner so far.

- Dedicated place for “business” logic outside the component scope. Even without context.
- Super simple and fully typed. API is minimal and store manipulation is easy. Actions and state in the same place. Globally available state properties.
- Well equipped for selectors and caching. Mapping, sorting, and filtering data operations all have a dedicated place and can be memoized.
- Can be used on the client side, SSG and SSR if needed

How this affects the Aave IPFS UI anyway?

The best part about zustand is it can be gradually implemented. Regardless of how good is the solution, if it can’t be implemented gradually it won’t work for the Aave IPFS interface, being a production application dealing with a critical protocol.

So initially we have done a PR to move the majority of the Aave business logic related to pool, governance, and staking to zustand while still exposing the same interfaces to the rest of the app.

This approach allowed us to implement just enough groundwork to make concurrent contributions possible without huge PRs and conflicts.

You can take a look at the changes onhttps://github.com/aave/interface/pull/964

What’s next and how to contribute

Given this is a progressive improvement, we will submit PRs to migrate the remaining parts of Aave interface to zustand in small chunks.

It is important to say that, at BGD we plan to use the same architecture for any other Aave project, and we have created an example for inspiration https://github.com/bgd-labs/fe-shared-examples. If you are into front-end development, give us your feedback!

There is an open discussion on https://github.com/aave/interface/discussions/1293, where the community is welcome to comment. Meanwhile, we'll introduce more ideas on how to design and migrate the remaining parts of the app, like the transaction manager or wallet connection.

Stay tuned

Disclaimer: Aave Companies is the maintainer of the Aave IPFS UI codebase. Even if being open source, we appreciate their collaboration during our contributions