FlatFeeStack as a Decentralized Autonomous Organization

Semester Assignment

Department of Computer Science
OST – Eastern Switzerland University of Applied Sciences
Campus Rapperswil-Jona

Autumn Term 2022

Authors

Pascal Knecht & Andy Pfister

Supervision

Dr. Thomas Bocek

Co-Examiner

Dr. Guilherme Sperb Machado

December 21, 2022

Abstract

The FlatFeeStack website allows developers to easily and transparently support open-source projects by providing a flat fee of $120 per year. FlatFeeStack gives open-source contributors funds based on their contribution to the project, calculated using metrics like contributed lines of code. The platform accepts payment by credit card and makes payouts using cryptocurrencies.

The community should operate and further develop FlatFeeStack, just like open-source software. For handling payments, FlatFeeStack requires a legal entity. The concept of an association under Swiss law can fulfill these two demands. Which allows for the realization of a collaborative cooperation that can act as a legal entity. Modeling the association on the blockchain ensures a clean democratic process and enables cooperation beyond Swiss borders. Based on the concept of a decentralized autonomous organisation (DAO), smart contracts were written so that all legal requirements of a Swiss association are met and thus, all activities of an association can be performed via the blockchain. Additionally, a frontend is provided for simple interaction with the smart contracts. This frontend was integrated into the existing frontend of FlatFeeStack. Statutes have been developed that are specifically adapted to the concepts of the DAO and the implementation of smart contracts. These are required for the association to be able to act.

With all artifacts from this project, the "FlatFeeStack Association" is ready to be founded.

Keywords: Blockchain, Decentralized Autonomous Association, Technology and Law, Fintech.

Executive Summary

Initial situation

FlatFeeStack is a website that simplifies the process of supporting open-source projects with a fixed annual payment of $120. The platform uses metrics like lines of code to determine contributors' contributions, and accepts payments by credit card. Payouts are made in cryptocurrency.

Just like open-source software, FlatFeeStack should be developed and run by the community. In order to handle payments, FlatFeeStack needs a legal entity. A Swiss association can fulfill these requirements, enabling the creation of a collaborative entity.

Additionally, the solution to be implemented should leverage blockchain technologies.

Procedure & Technologies

The project started with research. Existing solutions for governance systems on the Ethereum blockchain were evaluated, as well as the association's organizational structure to combine Swiss law with blockchain technologies.

It was decided to use the existing Governor framework, provided by OpenZeppelin, and to use the Ethereum blockchain, as its the most common blockchain to do development on it. The association structure was kept close to the givens of the Governor framework. The Governor framework's mechanisms have only been altered as required by law. For instance, Governor allows handing in proposals at any time [1]. Under Swiss Law, there needs to be a set appointment where association members meet and vote on an agenda [2]. Therefore, it was decided to implement the ballot vote on the blockchain, as it can replace a physical assembly, and Swiss law doesn't specify the medium on which the ballot vote is hosted.

Once the fundamental structure was designed, the implementation of the smart contracts started. Using the Solidity language for the smart contracts with Hardhat for unit tests. In parallel, a frontend application was built to interact with the smart contracts in

the Ethereum blockchain. The frontend was integrated into the existing frontend for FlatFeeStack, therefore using Svelte as the main framework.

After developing the initial structure for the association, bylaws were written to specify the bridge between Swiss law and the code. The bylaws specify the system of the association in more detail and determine how they are represented in code.

Final thoughts

The outcome of this project is a good starting point to start the FlatFeeStack association. It also allows participants to elaborate further on building bridges between new Blockchain technologies and existing law.

The project uncovered several topics that were previously not yet discovered. For instance, every member has the right to contest an accepted proposal in court if the member didn't participate in the vote and the accepted proposal contradicts the law [2]. If the court rules in favor of the member, the association's council has to take action. What happens if they do not act? Further developments are documented in a later chapter (5.2).

The chosen Governor framework allows interoperability with the existing Governance ecosystem so that the association can communicate on a code level with others. The developed frontend allows for easier participation in the association even if a potential member has never interacted with Blockchain technologies. The created bylaws accompany the planned launch of the association and will help clarify the interactions between members and smart contracts.

Acknowledgment

We would like to thank the following people for helping with this semester assignment: Dr. Thomas Bocek for the guidance and supervision during the course of this semester assignment.

Dr. Guilherme Sperb Machado for sharing his valuable experience from launching the GrantShares DAO and his inputs on software development on the blockchain.

Dr. Stephan D. Meyer for providing legal counsel and his experience with Decentralised Autonomous Organisations projects.

Jamie Maier for proofreading the semester assignment.

Contents

Chapter 1

Introduction

With the FlatFeeStack website, donations become more accessible and more transparent. For 120 USD per year, any developer can support any open-source project - the donation is equally split among the projects. If a developer supports more projects, then each project will get proportionally less. For a company, this is easier to budget as it is a flat fee per developer. This means, for ten developers, the budget is 1200 USD - always. Since each developer in the company can support those libraries and frameworks that make their work more efficient, there is no organizational overhead in deciding which projects to support. Furthermore, the company has an up-to-date technical map of its IT landscape. The distribution of funds to the open-source contributor is based on how much code was modified by the developer. Since an open-source project has its code publicly available, these metrics can be calculated automatically by FlatFeeStack.

The funds are distributed according to the contribution. The FlatFeeStack Application calculates a contribution score based on lines of code and other metrics. Another advantage is that the donation payment is made with credit cards, while the payout is done with cryptocurrencies. Since the contribution and crypto currency transactions are publicly available, FlatFeeStack is fully transparent.

In order to process credit card payments and have a bank account, FlatFeeStack needs some kind of organization behind it to be a legal entity. Since FlatFeeStack is based in Switzerland, there are some options like AG, GmbH, Non-Profit, or a Association (*Verein*).

An association is, in the case of FlatFeeStack an ideal way to be organized. The association lives through its members, and everyone can co-determine where it should go in the future.

A similar concept to the *Association* exists in the blockchain world, which is called a Decentralized Autonomous Organization (DAO) [3]. A DAO is a member-owned community

without central leadership. Its rules are encoded into a computer program that runs on the blockchain.

To combine the concept of the DAO and the association, MME introduced a new term Decentralized Autonomous Association DAA (more on that in the chapter 3). So a DAA is the same as a DAO but with the requirement to be compliant with Swiss law.

1.1 Assignment

This thesis has two primary objectives:

- Design and develop a DAO which is consistent with Swiss law.
- Develop a PoC Web Application to easily interact with the DAO.

1.2 Basic Conditions

This work was done as part of a semester assignment (Studienarbeit). A time budget of 480 hours is reserved for the work on this assignment and will be rewarded with eight ECTS credits.

Chapter 2

Problem Analysis

Functional and non-functional requirements are essential components of any software system. They define the capabilities and characteristics that a system must possess in order to meet the needs of its users and stakeholders. This chapter, describes the different types of functional and non-functional requirements and how they are used to guide the design and development of a system.

2.1 Functional requirements

In this section are all functional requirements and personas listed, who are relevant for this FlatFeeStack project.

2.1.1 Persona

User: A user is a person who is not already part of the DAA, but has the intention of joining it.

Member: A member is part of the DAA. They have the right to vote and make proposals.

Council Member: A council member is a member who has further obligations. They must hold the ballot vote and represents the association to the outside.

2.1.2 Use-Cases

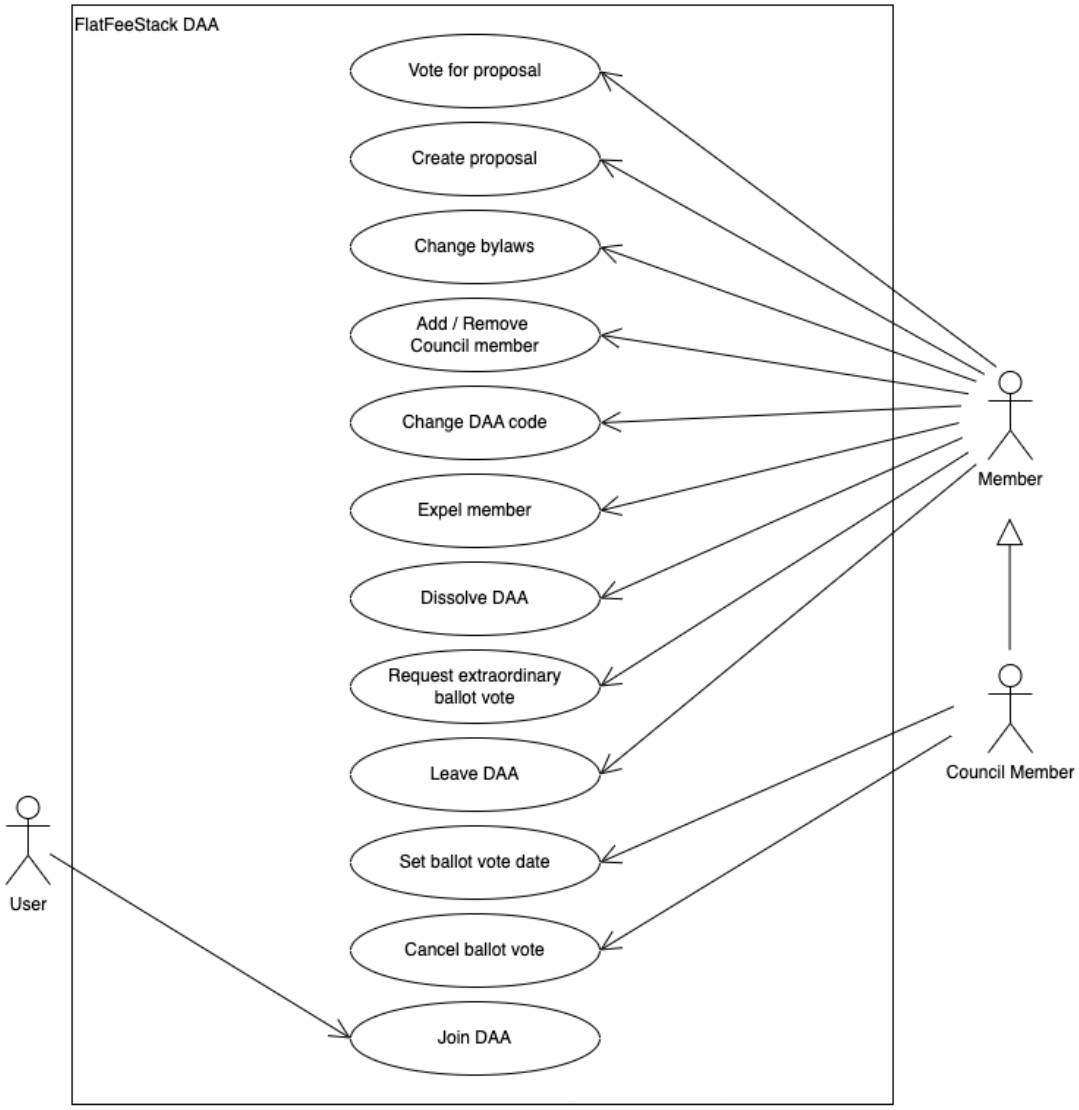

Figure 2.1: UseCase diagram

- **User can join DAA**
  A User can request to be a member of the DAA. It is necessary to do a KYC validation before a user is allowed to join.
- **Member can leave DAA**
  A member can leave the association with immediate effect.
- **Member can create and vote for proposal**
  A member can create a proposal. A member can vote for a proposal. A member can only vote once per proposal.

- **Member can propose and vote for new bylaws**
  If a member wants to change the bylaws they can create a proposal.
- **Member can propose and vote for a new DAA code version**
  If a member wants to change the DAA's underlying code, they can create a proposal.
- **Member can propose and vote to add or remove a council member**
  A member can create a proposal to add or remove a council member.
- **Member can propose and vote to dissolve the DAA**
  A member can create this specific proposal to dissolve the DAA. The specific steps to dissolve are written in the bylaws. It requires a 20% quorum and a simple majority.
- **Member can propose and vote to expel a member**
  A member can create a proposal to expel a member.
- **Define date for a ballot vote**
  A council member must set a date for next ballot vote. They must set the date at least one month before the vote.
- **Member can request an extraordinary ballot vote**
  Members can request an extraordinary ballot vote. This proposal passes if one fifth of all members accept it. The voting duration is two weeks.
- **Council member can cancel a ballot vote**
  If the council member has a good reason, they can cancel a ballot. There must be a replacement date. All agenda items must be moved to the next vote.

2.2 Non-functional requirements

The following non-functional requirements for this project have been identified.

1. Functionality: Each story is a functional component and does not break the functionality of others. Acceptance criteria: Unit Tests run for all subsystems in the main branch.
2. Portability: The application must be portable and able to run on multiple platforms or devices without requiring significant modification. Acceptance criteria: Manual testing on different devices and browsers.
3. Extensibility: The application must be easy to maintain and update over time, with clear documentation and support processes. Acceptance criteria: Time to fix bugs or to install updates.

4. Robustness: The application must be robust and able to handle unexpected or invalid input without crashing or behaving unexpectedly. Acceptance criteria: Manual testing with invalid or malicious input.
5. Code quality: The application must be written in clean, well-organized code that is easy to understand and maintain. Acceptance criteria: Static analysis run with each commit.

Chapter 3

Research

This chapter delves into the existing research on decentralized autonomous organizations (DAOs) and their smart contracts. It begins by reviewing the prototype DAA smart contract developed by MME Compliance AG, including its functionalities and dependencies on other smart contracts.

Next, the structure and requirements of Swiss association law are examined, as the DAA is intended to mimic the structure of an association under Swiss law.

Finally, the existing literature on DAOs and their potential applications in various industries is reviewed, including their advantages and challenges. This research provides a foundation for the design and development of the DAA in subsequent chapters.

3.1 Existing DAA smart contract

MME Compliance AG built a prototype of a decentralized autonomous organization [4]. They published a paper describing how the DAA mimics the existing Swiss law structure for an association and which function calls on the smart contract will issue what effect.

At a talk at the EthCC 2019 [5], the developers behind the smart contract itself gave more insights about the functionalities. The important slides for this assignment are the ones that show how different processes within an association are represented in the DAA. Figure 3.1 shows an overview of the different smart contracts that form the DAA.

The elected head of the association is here referred to as delegate, this semester assignment refers to it as council member.

The responsibilities for the different smart contracts are as follows:

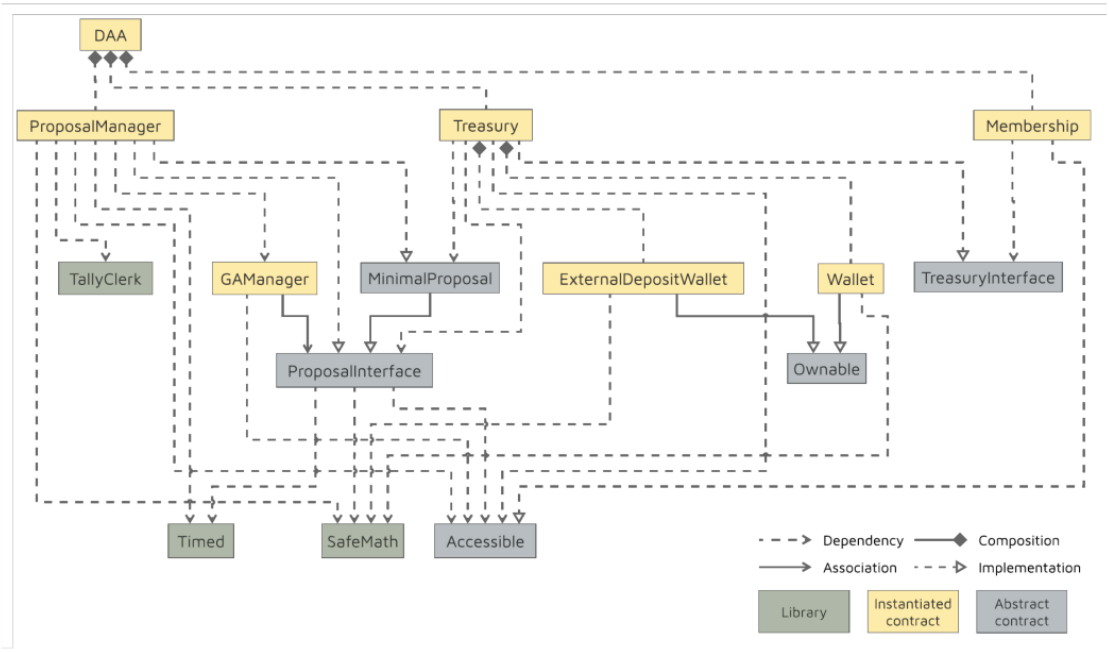

Figure 3.1: Smart contract architecture for a DAA

- Memberships: Keeps track of all members within the DAA including members that are part of the membership approval committee or the delegate.
- Proposal manager: As the name suggests, the proposal manager manages different kinds of proposals that a member can propose and vote on at the general assembly. The different kind of proposals are given by the proposal manager. The proposal manager depends on the Memberships contract to know which members are eligible to vote and how many votes are needed to make certain proposals pass or fail. Certain proposals, like expelling members, requires the applicant to be a delegate in the DAA.
- General assembly manager: The general assembly manager holds information about the general assembly of the association. It depends on both the Memberships and the Proposal manager contract. The Memberships contract once again is used to determine who is allowed to participate in the general assembly. The proposal manager is only used twice within the general assembly manager:
    1. In case a proposal for an extraordinary general assembly passed, the general assembly re-schedules the next general assembly according to the passed proposal.
    2. If a proposal for new bylaws passed, the general assembly manager saves a hash of its result.

- (Internal) Wallet: The internal wallet receives memberships fees. Via proposals, people can request to get money from this wallet.
- External wallet: If a proposal allows it, members outside the DAA can sponsor it. The external wallet is responsible to allocate received funds to proposals. A proposal can have one or more addresses where donations can be sent. Funds for a proposal need to be placed before voting concludes. If a proposal did not pass, the funds will be sent back to the originators.
- Treasury: The treasury contract is equivalent to an accountant: It manages the internal and external wallet. Members pay their contribution fee to this contract or can request to withdraw money from either wallet via a respective proposal. The treasury contract is owner of both wallets, nobody else is eligible to manage the funds.
- DAA: The DAA contract knows all the contracts listed above. Once all contracts are deployed and linked correctly, the DAA is officially active.

The code for the DAA is publicly available on Github [6].

3.2 Swiss Association Law

The DAA should mimic the structure of an association under Swiss law [2]. Generally, an association is a group of people who have a matching interest, like politics, sports, economy etc. (Article 60). An association allows this group to become a legal entity under the Swiss law if they fulfill certain requirements.

The requirements for the planned association are simple, as FlatFeeStack is neither subject to audit nor participates in commercial trade (Article 61). FlatFeeStack needs to establish bylaws, so-called *Statuten* and a board of directors. Once both are confirmed by the founding members of the association at the first general assembly meeting, the association is active and can be added to the trade register.

The Swiss Civil Law article 60 to 79 define additional rules that an association may follow. An association is allowed to override most of these rules (Article 63), except:

- There can be an extraordinary assembly if one fifth of all members request it. (Article 64, 3)
- A general assembly can be cancelled by a council member for important reasons (Article 65, 3)
- Members are not allowed to vote in proposals that involve themselves, their spouse or first line relatives. This also includes other associations. (Article 68)

- Members are only allowed to leave by the end of the year or, if the association defines one, by the end of a fiscal year. They need to announce it six months in advance (Article 70, 2)
- Accepted proposals that contradict either the bylaws or Swiss Law, can be challenged at court by a member who didn't participant in the vote, within a month. (Article 75)
- If the association is insolvent or the council members are no longer available, the association needs to be dissolved. (Article 77)

The integration of Swiss law and the DAO provides certain legal and organizational challenges [7]:

- The bylaws can override the exceptions listed above if they benefit the members. For example, it is allowed that the bylaws specify that members can leave at any time instead of only by the end of the calendar year.
- There are two possibilities for implementing a DAA. The first variant is to build a technical solution compliant with the law. The second variant is a DAO who is embedded into the association, but all the legal matters are happening off-chain (e.g. physical general assembly, voting, financials, etc.).
- The first general assembly establishes the association that confirms the council members and the bylaws. There is a possibility that the first general assembly is replaceable with a ballot vote on the blockchain. Ballot votes can also replace the general assembly held in person. Additional points were clarified in later mail exchange.
  - A council member announces the date for the next ballot vote.
  - Members can create proposals for the next ballot vote until a specific date.
  - After the submission deadline, the submitted proposals compose the ballot vote agenda.
  - Voting is open for a particular time for all proposals submitted.
  - Members need to have the possibility to add a reason to their vote.
  - Note that Swiss law doesn't specify what medium to use for the ballot vote. You just need to document the voting process including the medium in the bylaws. Additionally, you need to implement some kind of verification process when accepting new members so not everybody can just jump in and vote.
- bylaws can be in English.

- Hostile takeovers should be taken in consideration. Often, specific mechanisms in DAOs allow an attacker to take control of the DAO and ultimately drain its funds. To counteract this problem, membership requests should need an approval by a committee (who are existing members of the DAO) and also a KYC.

3.3 On-chain Governance

An essential part of the blockchain world is that the majority must agree to any further development or change to the system [8]. This means that no central body can decide alone.

Blockchains have a built-in governance systems which are built on the protocol level of the blockchain. This implementation varies from blockchain to blockchain. Developers can state code changes through proposals and each participant votes for or against this change.

At the level of smart contracts, the contract owner often holds the power of decision-making. If you want to delegate the decision-making to the community, you will implement a so-called governor contract.

3.3.1 OpenZeppelin Governor

OpenZeppelin is an organization that provides a framework of reusable smart contracts with high-security standards. For example, they provide smart contracts to easily implement an ERC-721 (NFT).

They also provide a set of contracts to have governance in smart contracts [1]. The contracts are very modular and can be easily extended via solidity inheritance.

The key functionalities that a governor contract needs to take care of are:

- Proposals
- Votes
- Counting

Structure of the Governor contracts

Core

The core module contains all the logic. It is abstract and requires a vote module and a counting module. You can either write them yourself or work with a provided one.

You also need to set some parameters:

- votingDelay: How long after a proposal is created should voting power be fixed. A large voting delay gives users time to unstake tokens if necessary.
- votingPeriod: how long the proposal is open to vote

Vote Modules

Defines how voting power is determined and how many votes are needed for a quorum. The idea is that the voting works with some kind of token. For ERC20 and ERC721, the necessary contracts are provided.

Counting Module

Defines the different options for voting. A simple module with the voting options against, for and abstain is provided.

Timelock

Depending on the decision taken on a proposal, you might disagree and want to leave the organization. Once a proposal is accepted, the content is transferred to a Timelock module. It holds off the execution of the decision for a specific time. This mechanism allows people to leave the organization in time before changes come into effect.

Proposal Lifecycle

A proposal is always a sequence of actions which will be executed if it passes the vote. Each action consists of a target address, calldata encoding a function call, and an amount of ETH to include.

When the proposal is active, members can cast their vote.

If the voting period is over, a quorum was reached and the majority voted in favor, the proposal can proceed to be executed.

If a timelock is in place, the actions must be queued, otherwise the proposal can be executed immediately. The actions are not stored in the contract to save gas. Instead, a hash of the proposal ID, function call, and function data is held in the chain. When calling `queue` or `execute` on the Timelock contract, the action needs to be passed again as a parameter. The hash will be calculated from the parameter and compared against the stored version.

Chapter 4

Solution

This solution chapter examines the structure and responsibilities of the FlatFeeStack association, including how it will fund its mission and address challenges such as vote manipulation and fraudulent proposals. It also describes the application architecture for the association, including the use of smart contracts and the integration with the existing frontend of FlatFeeStack. The chapter also discusses the implementation of various functionalities for the association, including membership management and proposal management.

4.1 Structure

This section will look at how the association is structured and its responsibilities. It also clarifies how the association can fund its mission.

The FlatFeeStack association is responsible to maintain the platform. This includes to pay for operating expenses, keep the platform up to date and develop new features.

In order to have enough funds available for those tasks the member pay a yearly membership fee. In addition, 1% of all project donations on FlatFeeStack will be sent to the association.

Every member can submit proposals to shape the future of the platform. Proposals can be everything as long as they contribute to the further development of the platform or the association. Figure 4.1 illustrates this process.

Every member has one vote for one proposal.

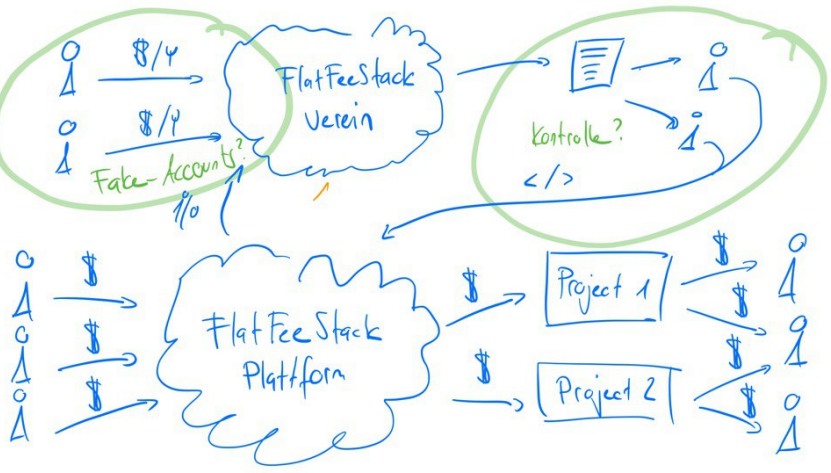

Figure 4.1: DAA structure

Some challenges came up with this approach:

- If the membership fee is low, one could create many Ethereum wallets and join the DAO to gain a majority of the votes. Since proposals are *free-text*, the attacker could propose to send all of those DAOs funds to themselves. This proposal would pass as the attacker has, as mentioned, a majority of the votes. Members have to be approved by multiple council members to mitigate this risk. This risk is also known as the Sybil attack.
- A member could propose to implement a feature for FlatFeeStack in exchange for some funds. However, the receiver of the money could just run away after receiving the money without doing any work. The risk is not mitigated with the proposed solution but could be solved as follows.:
  - Members cannot request funds prior to implementation, but rather can request money after they've done the work.

  - Instead of paying everything upfront, the DAO would send e.g. 50% of the requested fund. The remains would be sent once the DAO accepts implementation (via a separate proposal and vote).

- A user could join the association but loose interest over time and not participate anymore. This risk is mitigated by implementing a function in the smart contract that allows members to be removed if they do not pay their membership fee for a certain amount of time.

4.2 Application architecture

This chapter describes the implementation of the structure of the association, described in 4.1.

4.2.1 General

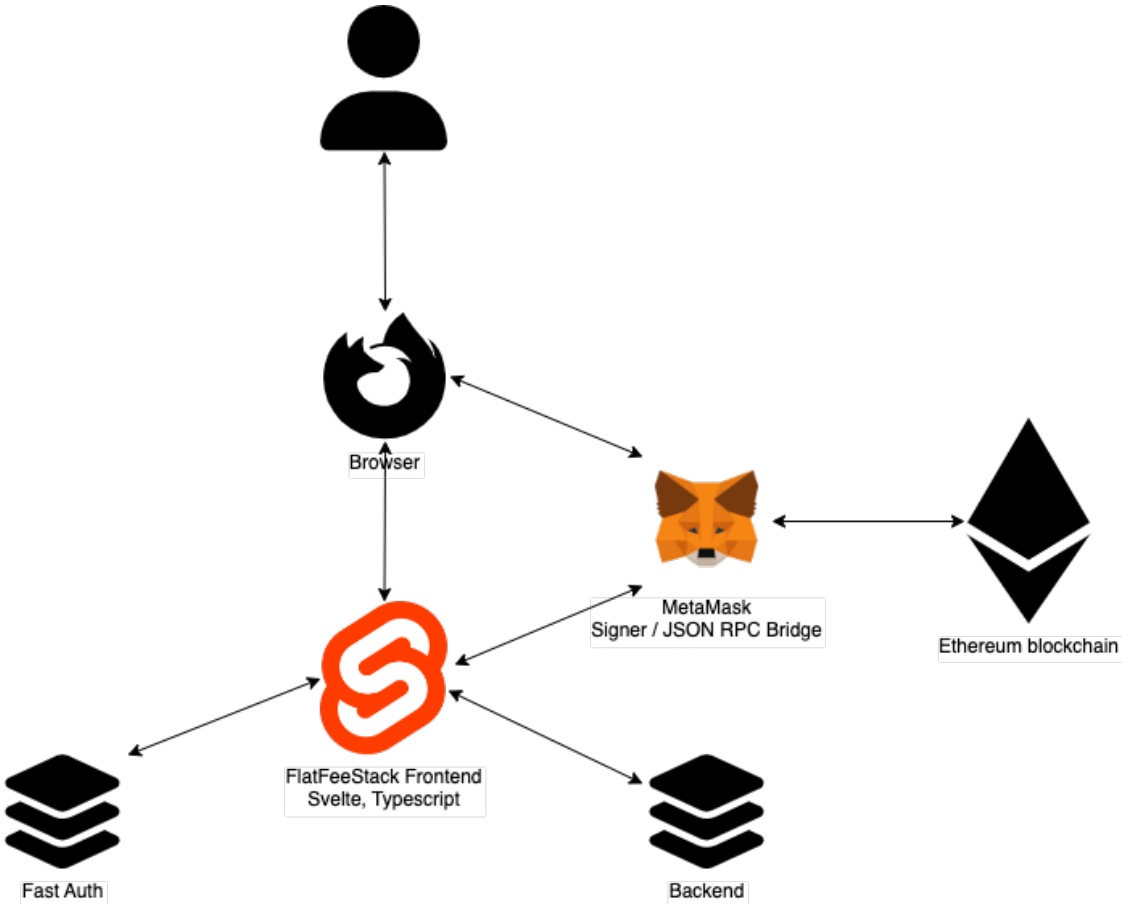

Figure 4.2: General application architecture

The general application architecture is similar to a typical decentralized application design.

- There is no backend application running to manipulate data. The frontend application connects directly to the Ethereum blockchain, where compiled versions of the smart contracts run.
- To connect to the Ethereum blockchain, a user needs a browser with MetaMask installed. MetaMask is a free browser plugin that enables users to interact with the Ethereum blockchain. MetaMask is used for two separate tasks.
    1. MetaMask injects an object into the JavaScript runtime of the browser that allows interaction with the Ethereum blockchain.
    2. If a user needs to perform an action (e.g., place a vote) within the application, it must be wrapped in a transaction. This transaction needs to be signed by the user. MetaMask holds the user's private key to their Ethereum accounts and allows them to sign transactions. Therefore, the frontend does not need to know the user's private key but can dispatch a prepared transaction to the MetaMask browser plugin with the command to sign it.
- The frontend integrates with the existing frontend of FlatFeeStack, which is written in Typescript and Svelte.
- There are two different services with which the frontend interacts: A backend and an authentication service. Those are required for the use of all functions of FlatFeeStack, but were created outside the semester assignment. More services are part of the FlatFeeStack ecosystem but are not part of figure 4.2 as they are irrelevant to the work. Complete documentation can be found in the semester assignment of Armend Lesi and Marco Endres, chapter 4.2 [9].

4.3 Smart contracts

The backend is not a traditional application, but rather a composite of several smart contracts running on the Ethereum blockchain. This creates some unique circumstances on how to draw the interaction between them.

- Solidity, the programming language for smart contracts on Ethereum, allows classes to inherit from multiple base classes. Inheritance is drawn with empty arrowheads in 4.3.
- Deploying a smart contract creates an instance of it. It is possible to deploy multiple instances of the same contract. However, the instances are not related, and there is

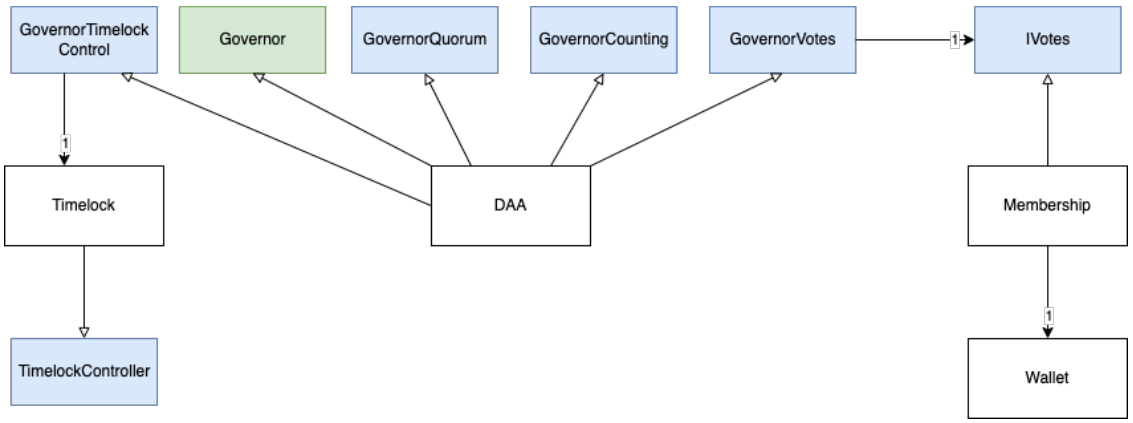

Figure 4.3: Smart contract relations

no easy way to discover all deployed instances of a particular smart contract on the Ethereum blockchain.

There are four smart contracts that compose together the DAA on the blockchain.

Generally, the implementation uses the Governor contracts from OpenZeppelin 3.3.1. This provides a foundation for everyday tasks within the association, such as creating proposals and voting on proposals. Usually, OpenZeppelin contracts can be referenced from other contracts without any modifications. However, to implement the ballot vote, the original Governor contract (highlighted in green) had to be copied and modifications had to be made. Since the Governor contract was built using inheritance, several other smart contracts had to be copied into the project. Those contracts are highlighted in blue. Irrelevant code in these contracts was removed, but the program logic was not modified.

The Governor contract wants a connection to another contract that implements the `IVotable` interface to determine the number of votes per member. Due to the unique requirements for joining the association compared to a regular DAO, a custom membership contract implementation that inherits from `IVotable` was necessary. On the other hand, this abstraction allows someone else to take the DAA contract and connect it to any other `IVotable`-compatible contract.

A separate wallet contract manages the funds for the association. This is for separation of concerns.

The fourth contract is the time lock contract. With the default Governor contract, once a proposal is accepted, any member can execute the proposal's instructions immediately. However, specific proposals might have drastic consequences on the DAO, and members who voted against the proposal, but got outvoted by other members, want to leave the association. With the time lock controller and the Governor module `GovernorTimelockControl`

, proposals are sent to a queue in the timelock controller after they are accepted. In the case of the FlatFeeStack DAO, the proposals need to be in this queue for a day before any member can execute them. This mechanism gives members who disagree with a proposal time to leave the DAO.

4.3.1 Upgradable Contracts

By design, smart contracts are immutable on the EVM. A contract's code cannot be changed once it has been deployed on the blockchain. Often software needs to be able to change, be it for bug fixing, further developments, etc.

For these scenarios, different mechanisms exists to be able to upgrade smart contracts [10]. OpenZeppelin uses the proxy pattern for all of their upgradable contracts. Because this project relies on OpenZeppelin contracts, the proxy pattern is the chosen mechanism for contract upgrades.

The user calls a proxy contract as shown in 4.4. This proxy is responsible for forwarding transactions to the logic contract and back. The logic contract contains all the logic to compute a transaction.

Figure 4.4: Proxy Pattern

Both the proxy and the logic contract are immutable. In case of an upgrade, the logic contract needs to be replaced with a new one as shown in 4.5. After that, the address of the new implementation is stored in the proxy contract.

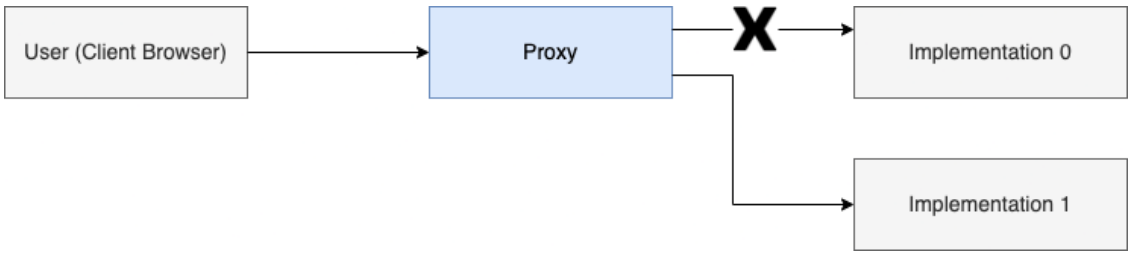

Figure 4.5: Proxy Pattern Upgrades

To make a contract upgradable, a developer has to pay attention to two things [11].

1. The contract must avoid storage collisions. Storage collision happens when the order of variables between different contract versions changes.
2. The contract can not use a constructor. Instead, an initialize function should be used.

4.3.2 Contract owner

Certain functions on the different smart contracts should only be called if a proposal is passed.

The Governor framework has this mechanism by default. It has an attribute named executor, which resolves to an address that refers to itself or, when using the `GovernorTimelockControl` module, to the timelock controller. When a member executes the proposal's content, the corresponding `execute` function executes the proposal contents in the name of the executor, not the member. When a protected function is called, the protected function checks if the caller is the same as in the executor property. And since proposal contents are executed in the name of the executor, the security check will pass.

Neither the Membership nor the Wallet contract know about the DAO contract. Also, this executor property, given by the Governor's framework, is private.

Therefore, the Membership and Wallet contract inherits from the Owner contract provided by OpenZeppelin. This adds a private property named owner to the smart contracts and a function to verify if the caller is the contract owner. Once the timelock controller is deployed, it owns both the Membership and Wallet contract. When executing a proposal, the caller is the timelock contract. And since the timelock controller cannot run anything that wasn't a successful proposal, the security mechanism of the contract owner is equal to the executor property of the DAO contract.

4.3.3 Development environment

The development environment for the smart contracts is based on Hardhat. It consists of different components for editing, compiling, debugging, and deploying your smart contracts and dApps, all of which work together to create a complete development environment [12].

Hardhat is based on NodeJS and can be installed using NPM. Once the package is installed, it allows you to set up new projects based on Javascript or Typescript. For the DAA project, the Typescript setup was chosen to have type-safety.

Hardhat gives the following directory structure:

```
daa
├── .github — Github Actions Workflows
├── contracts — Smart Contracts
├── deploy — Blockchain deployment scripts
├── deployments — Deployment state to different networks
├── scripts — Helpful scripts
└── test — Unit tests for smart contracts
```

Testing

The test directory is used for writing unit tests. The unit tests themselves are written in Typescript. Hardhat uses the chai and Mocha libraries for test assertions and adds a few additional ones specific to Blockchains, e.g., expectations that a particular operation will revert [13]. Hardhat compiles the Solidity code and boots up a temporary blockchain when running the tests. The temporary blockchain will be deleted after all tests are completed. The user is responsible for making sure the contracts are deployed prior to running the tests. For the DAA tests, this is done by having a `deployFixture` method in each test file that sets up the smart contracts and all relevant configurations.

Below you can find a unit test example written using Hardhat:

```
    describe("increaseAllowance", () => {
        it("cannot increase more than total balance of wallet", async () => ↵
{
            const { owner, otherAccount, wallet } = await deployFixture();

            await expect(
                wallet
                .connect(owner)
                .increaseAllowance(
                    otherAccount.address,
                    ethers.utils.parseEther("200.0")
                )
            ).to.be.revertedWith("Keep allowance below balance!");
        });
    });
```

Listing 4.1: A unit test for the Wallet contract using HardHat

As shown in listing 4.1, the fixtures are deployed first, which gives us back the owner of the smart contract and a metaclass object from ether.js of the Wallet contract [14]. Afterward, it is expected that calling `increaseAllowance` with the parameter `200 ETH` will revert with the message `Keep allowance below balance!`.

When pushing code changes to the GitHub repository, the tests are run with GitHub Actions. Additionally, GitHub Actions will report the code coverage.

Deployment

Hardhat offers support to deploy contracts to a local chain, test networks, or the main network using scripts written in Typescript. However, the deployment mechanism doesn't keep track of existing deployments, which is vital for the DAA, as the proxy contract address should stay the same with each deployment.

Therefore, it was decided to use the community-maintained hardhat-deploy plugin. This plugin saves the addresses of the deployed contracts in different JSON files, including the compiled code version that was deployed. Deployment scripts are idempotent, so rerunning them without any change won't trigger a new deployment [15].

The deployment scripts reside in the `deploy` folder, while the deployment states with the mentioned JSON files are saved to `deployments`.

Scripts

Hardhat allows one to writing scripts where one can access the Hardhat runtime environment. The DAA project contains several scripts.

- `addMember`: After deployment of the contract, this script allows to add a regular member directly to the DAA. This script is used for setting up the local development blockchain.
- `addSlots`: After deployment of the contract, this script creates a few voting slots and one proposal to fill the DAA with data. This script is used for setting up the local development blockchain.
- `mineBlocks`: The DAA is heavily dependent on time. This script allows to mine a certain amount of blocks to fast-forward time on the blockchain itself.
- `exportInterfacesToFrontend`: This script reads the ABIs of the smart contracts and exports them to the frontend repository, where they are used to generate the ether.js meta classes.
- `exportContractAddressesToPayout`: Since the frontend fetches the smart contract addresses from the payout service, this script allows to export the contract addresses stored in the JSON files from hardhat-deploy to the payout service.

4.3.4 Membership

Anyone can become a member of the FlatFeeStack DAO, but they should know blockchain technology or be interested to learn it.

The `Membership.sol` Contract implements everything regarding membership.

Types

There are two different types of Members, with various rights and obligations.

The first type are the council members ("Vorstand"). They are the head of the association and represent the association to the outside world. The bylaws contain a complete list of their responsibilities (4.6).

The association needs at minimum two council members because two different council members must approve new members to join the association.

The second type is the regular member.

States

There are four membership states: `nonMember`, `requesting`, `approvedByOne` and `isMember`.

If an Ethereum account address is not in the membership list, it has the status `nonMember`.

If a non-member wants to become part of the association, they can request a membership (`requestMembership`). This adds them to the membership list and assigns the status `requesting`.

Now a council member can approve the membership (`approveMembership(address _adr)`) and with that, the status `approvedByOne` is assigned. After that, a second, different council member must approve the membership for the second time. With that, the member has the status `isMember`.

Until the member has paid his membership fee, they cannot participate in votes or create proposals.

Fees

Every member has to pay membership fees (`payMembershipFee()`). They must be paid once a year and are 30'000 Gwei. A list inside the contract keeps track of the member's next membership fee payment date. If a new member pays their membership fees, they gain the right to vote and to create proposals. The voting rights are not revoked even if the member does not pay the following membership fees.

Voting Power

The implementation of contracts according to the OpenZeppelin Governor standard requires the integration of a module to determine an individual account's voting power and the total available voting power. In this case, every account has the same voting power. To become a votes module, the membership contract implements `IVotesUpgradeable`. With that also, two new structs are introduced `_voteCheckpoints` and `_totalCheckpoints`. The

`_voteCheckpoints` keeps track of the votes an account has. The `_totalCheckpoints` keeps track of all the voting power in total.

These structs work with a history. This means the data is stored together with the block-number in which the transaction ran. This gives the possibility to read voting power from past snapshots.

4.3.5 Ballot Vote

As mentioned in 3.2 a ballot vote is an excellent way to substitute the concept of a general assembly. It can be held digitally, which fits perfectly with this blockchain project.

A council member announces a Ballot Vote. They have to announce the vote at least one month before it takes place. After the announcement, all members have time to submit proposals (see 4.3.6) to this ballot vote. A ballot vote always has a start and an end. The council member can specify the start. The smart contract calculates the end. So the duration of a ballot vote is always the same.

A council member can cancel a ballot vote if they have a good reason (required by law). That must happen at least one day before the vote. Already existing proposals in that ballot vote will be moved to the following ballot vote. If there is no following vote the council member won't be able to cancel.

4.3.6 Proposals

Proposals are the instrument for the members of the organization to make changes. For example, a member could propose getting some money from the organization for their work.

Figure 4.6 shows the states that a proposal goes through. It had to be adjusted a bit, as in the original OpenZeppelin implementation (3.3.1). This was done because proposals need to be assigned to a ballot vote.

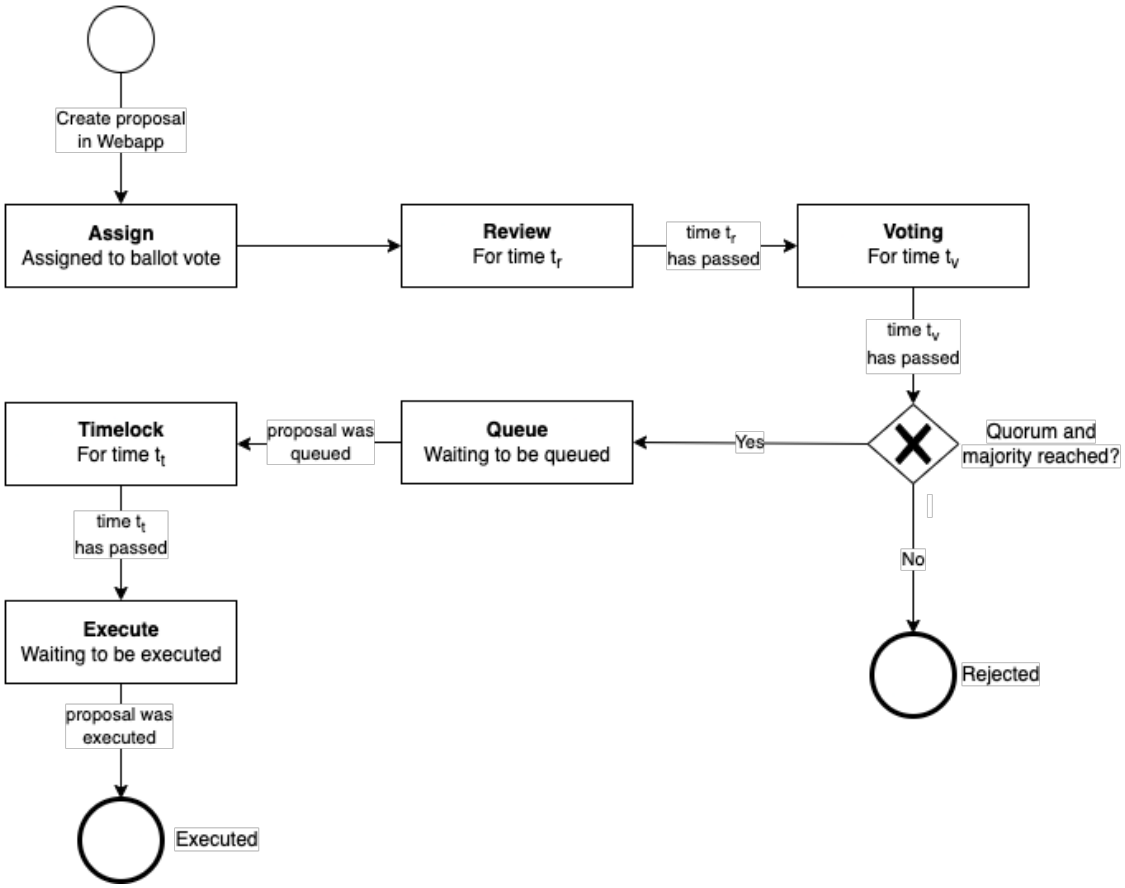

Figure 4.6: Proposal Lifecycle

Note that t_v, t_t, and t_r are adjustable variables in the smart contract and, therefore, can be changed via a proposal. t_r sets the minimum time limit before the vote within which proposals may be submitted.

Assign

The proposal will be assigned to the next possible ballot vote.

Review

Other members can review the proposal and form an opinion on whether they want to support it. They have got at minimum t_r amount of time for that.

Voting

All members have t_v time to cast a vote for the proposal. If they want, they can give a reason for a vote.

Queue

After the voting window is over and the proposal is approved, it can be queued into the timelock. It needs an external invocation. For example, a council member can queue all the proposals.

Timelock
The proposal is locked for t_t time and can't be executed. This is for security.

Execute
After the timelock is over, the proposal is ready to be executed. It needs an external invocation. For example, the proposer can execute his proposal.

Executed
The proposal is executed. This is the end of a proposal lifecycle.

Rejected
The community rejected the proposal. There is no change to adjust the proposal for further use. If something similar wants to be proposed, a new proposal has to be made.

4.3.7 Voting

Every member can cast one vote per proposal, and everyone has the same voting power. There are three types of votes:

- Against: If the member is against the proposal.
- For: If the member is in favor of the proposal.
- Abstain: If the member wishes to abstain from voting.

The votes are counted at the end, and a simple majority is sufficient for the proposal to be a success.

Founding Ballot Vote

Every association must have a founding meeting where the bylaws and the council member must be approved. To achieve that, a ballot vote will be created on the deployment of the smart contracts. This vote only consists of one proposal that updates the hash of the bylaws on the DAO (4.6.2). The vote takes place one week after the deployment to allow other members to join the association. A successful funding ballot vote establishes the association as an independent legal entity.

4.3.8 Time

The DAO relies on time for most of its functionalities. One can only vote during a particular time window. Members can create new proposals for a ballot vote until a specific time.

Ethereum generally knows two kinds of time: block numbers and timestamps. The block number is the number of the block relative to the first block. The timestamp to the block

gets added by the miner based on their system time and can be manipulated to a certain extent [16]. The DAO uses a mix of these two different time variations, below are the important ones:

- The next membership fee payment for members is saved as a timestamp.
- Everything in the DAO is saved and calculated using block numbers. This was given by the OpenZeppelin Governor.
- The time how long a proposal must be queued before execution is saved as a timestamp. This was given by the OpenZeppelin time lock conteract.

Roughly every twelve seconds, a new block gets created on the Ethereum network. When dealing with future block numbers, the frontend takes the difference between the current block number and the given future block. It multiplies it by twelve seconds to approximate the date when the miners will create this specific block. However, it's an approximation. Especially since the Ethereum merge, the time between blocks has gone down massively [17]. Therefore, it could be that this constant has to be adapted at some point depending on the average block time.

4.4 Frontend

This section focuses on the design and implementation of the frontend for the FlatFeeStack association.

4.4.1 User interface draft

One of the functional requirements is to build a frontend that allows users to interact with the association on the blockchain. The DAA part of the frontend will be integrated into the existing frontend of FlatFeeStack, so all information about FlatFeeStack is accessible from one location. Below are wireframes for the different views within the frontend application.

Figure 4.7 shows the base layout for the application. On the left side is the navigation to manage different aspects of the association. The existing frontend for FlatFeeStack also incorporates the navigation on the left side, using icons and text to describe the items. The center displays the upcoming and past ballot votes (named voting windows). Depending on the state of the voting window, members can vote for proposals (see figure 4.9) or submit new proposals (see figure 4.8).

Proposals will be grouped into voting windows. A voting window maps to the definition of a ballot vote required by law (3.2).

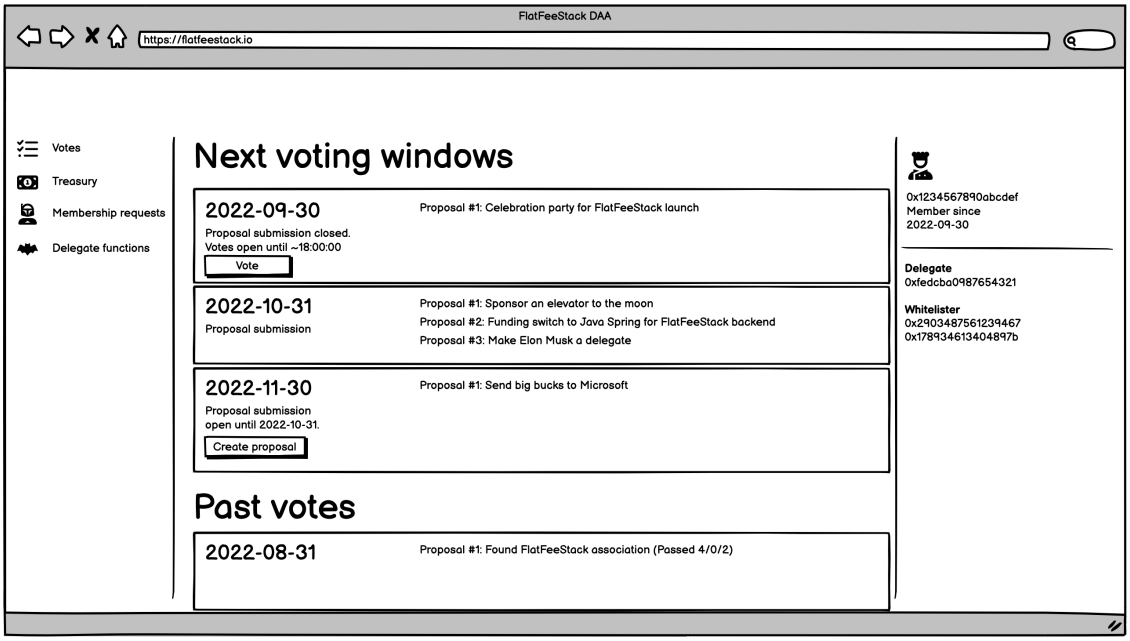

Figure 4.7: Start screen

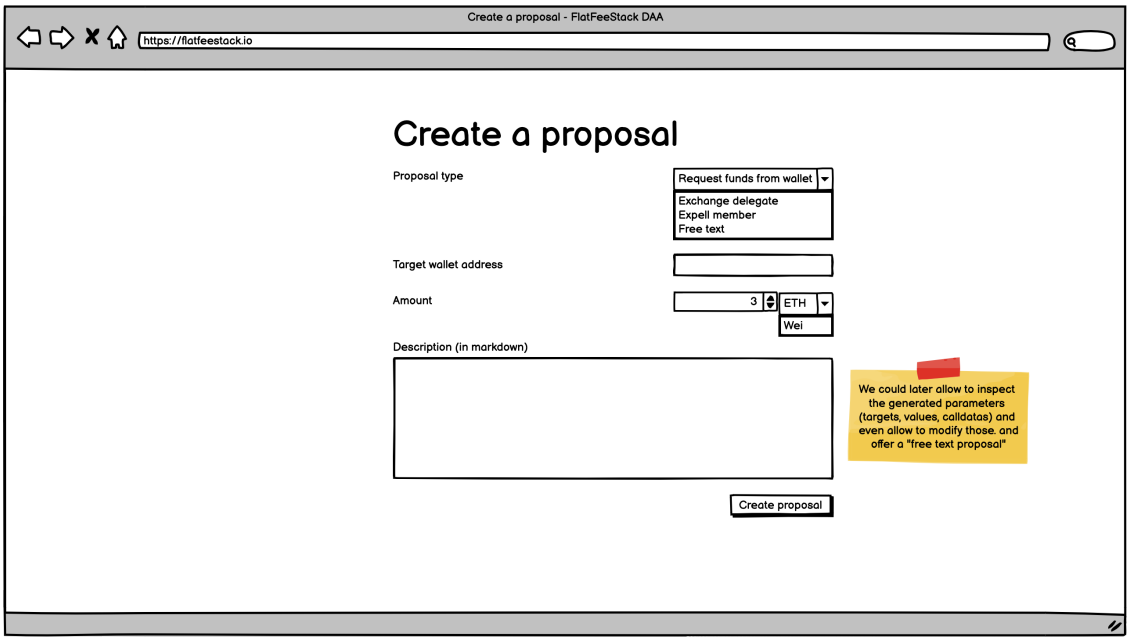

Figure 4.8: Create a proposal

The call to the blockchain to create a proposal in the association is quite complex. A simple proposal form abstracts the complexity for standard proposals, like requesting funds or an exchange of the representative. Therefore, the form changes the available fields based on the chosen proposal type, as shown in figure 4.8. The form will allow advanced users to influence the parameters sent to the blockchain directly.

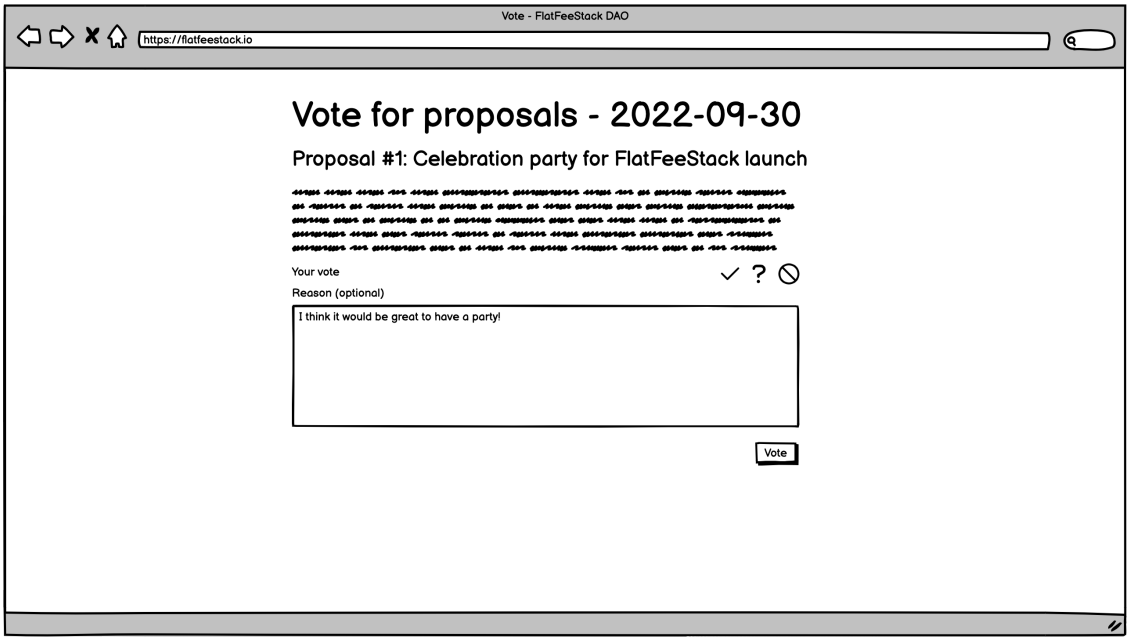

Figure 4.9: Vote for propoals

Figure 4.9 shows the screen where users can submit votes. It will list all votes; for each, the user can decide if they support the proposal, vote against it, or have no opinion about it. For each vote, a user can add a reason.

4.4.2 Implementation

General

As outlined in section 4.2, the frontend for the DAO is integrated into the existing frontend for FlatFeeStack. This frontend is built with the Svelte framework.

The frontend needs to know the addresses of the contracts on the blockchain and their application binary interface to interact with them. Additionally, as mentioned in section 4.2, MetaMask is needed to connect to the blockchain and sign transactions for actions a user executes. These interactions are enabled with the ether.js library, which was already part of the frontend application, as it has existing functionality that fetches and sends data to the Ethereum blockchain.

ether.js builds so-called JavaScript meta classes based on the ABIs and addresses for a smart contract [14]. When any method on those meta classes is called, ether.js builds the corresponding transaction and sends it to a provider. For FlatFeeStack, this is MetaMask. MetaMask inspects the transaction: If it is a write operation, it requests the user to confirm the operation and signs the transaction with the user's private key. Afterward, the transaction is sent to the JSON RPC interface of Ethereum, which queues the transaction

and allows it to be included in the next block. The transaction goes directly to an Ethereum node for evaluation [18].

The Svelte framework organizes frontend code into a tree of smaller code snippets called components. Data needed to render components is generally stored in each component separately. Once a component is removed from the DOM, its stored data is also lost.

To preserve data across components, Svelte provides a store functionality that is based on the observer pattern. Multiple components can subscribe to changes in the store. When navigating in the frontend, the state in stores is preserved. A store can have an `initialize` function when the first component subscribes to the store. For FlatFeeStack DAO, this helps initialize the mentioned ethers.js meta classes.

A significant part of the state needed to render the frontend of the DAO is preserved in those Svelte stores:

- The current signer fetched from MetaMask.
- The meta classes for the smart contracts.
- Any data that is needed across multiple components, like the list of council members or a list of proposals.

This documentation will mention when Svelte components read or write data to Svelte stores.

One significant difference between traditional applications that interact with a regular backend application and the FlatFeeStack DAO frontend is the shape of information. For traditional applications, the entire state of an object is served in one reply, and the main job of the frontend is to arrange the given information on the computer screen. However, depending on the requested view, the FlatFeeStack DAO frontend needs to combine information from multiple requests to get a complete state of the objects. For example, a first call checks if the signer is known to the membership contract to retrieve the status of the current logged-in signer (`getMembershipStatus`). A second call in parallel retrieves the list of council members. If the signer is a member of the DAO, the frontend will check if they are also a council member to display the correct membership status in the frontend. Fetching this kind of information is usually just one call to the backend in a traditional application. Therefore, some frontend components execute many parallel calls to the blockchain and do the heavy lifting with object transformation to get a renderable state.

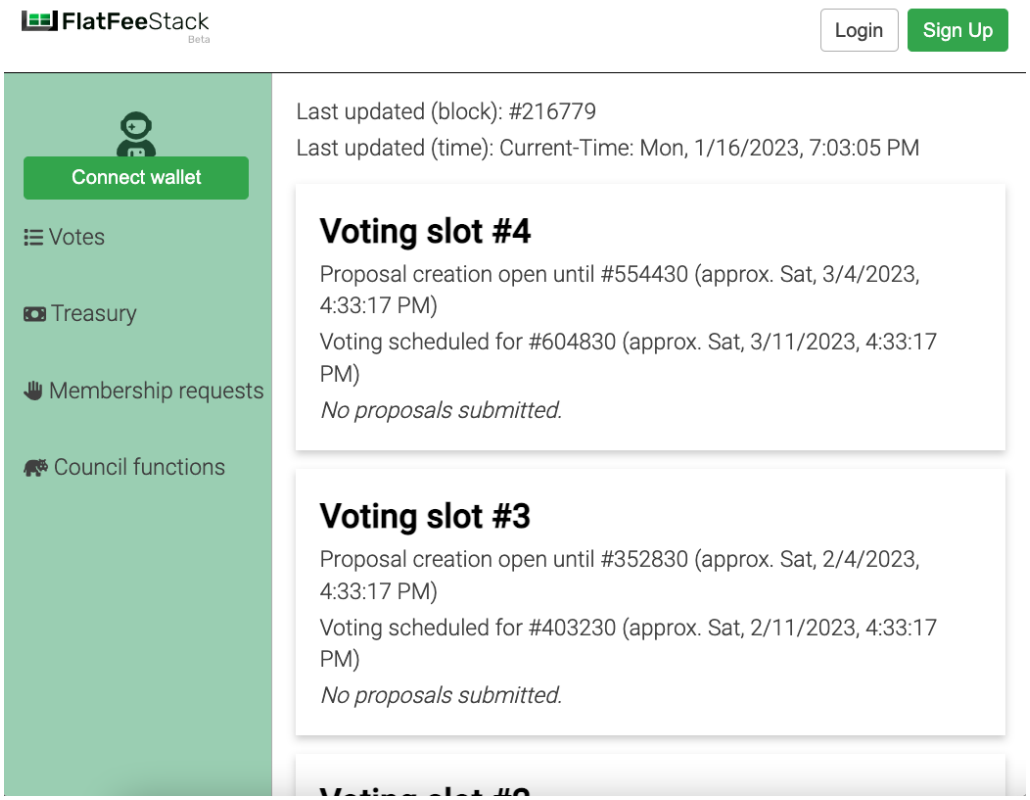

Figure 4.10: Navigation with MetaMask wallet not connected

Navigation

The navigation component is rendered in all views of the DAO frontend. Views request to be rendered inside the Navigation component, which places their content on the right-hand side. The navigation also displays the currently connected user and navigation options.

The navigation component is vital to the frontend as it pre-fills several essential stores. Once a user clicks on the *Connect wallet* button shown in figure 4.10, the frontend saves a signer object from ether.js into a Svelte store. This signer object allows identification of the user's Ethereum address, which is used in several other filtering mechanisms or to check access in different views (e.g., council member functions). Before the connection to the Ethereum wallet is established, an empty signer is used to read data from the blockchain.

Additionally, when the signer object changes, the mentioned ether.js smart contract ether.js contract meta classes are re-initialized so transactions from ether.js can be sent to MetaMask.

With a connected wallet as shown in figure 4.11, the navigation displays the Ethereum address of the connected user, allows to inspect the current membership status (4.4.2) or to leave the DAO.

Overview of voting slots and proposals

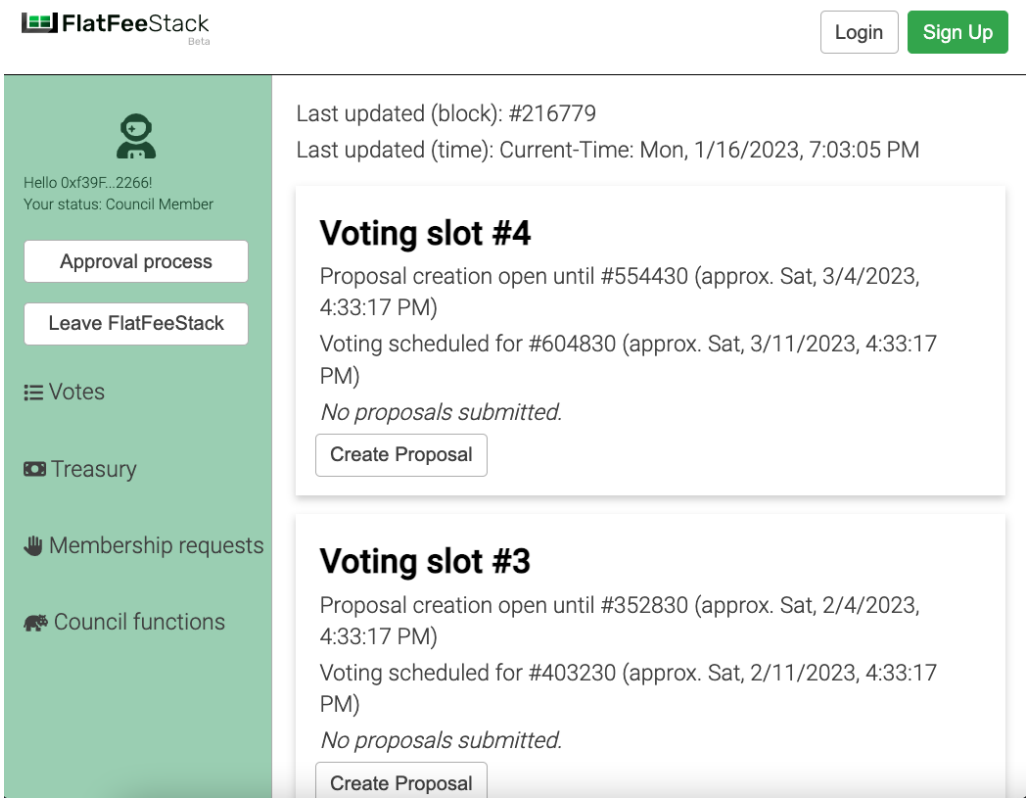

Figure 4.11: Overview of the voting slots with the MetaMask wallet connected

As shown in figure 4.11, the start page of the FlatFeeStack DAO frontend is the overview of voting slots and proposals. The data is fetched in multiple calls:

1. As the blockchain cannot return array properties, the frontend first asks for the number of voting slots.
2. For the received number of voting slots, it receives their effective voting start date.
3. Based on this start date, the frontend fetches a list of proposal identifiers for each voting slot.
4. Finally, for the list of proposal identifiers, the blockchain is searched for `ProposalCreated` events that match the retrieved identifiers. The `ProposalCreated` event contains complete information about the proposal, like the proposer or the description.

The retrieved voting slots and proposals are saved in respective stores.

Depending on the current time, the user can execute various functions on this page.

- Create a new proposal for a voting slot if the slot still allows adding proposals. The corresponding button takes the user to the create proposals form (4.4.2).

- The user can navigate to the voting form if the voting is open for a specific slot (4.4.2).
- The user can navigate to *Execute proposals* if voting is concluded (4.4.2).

Create proposal form

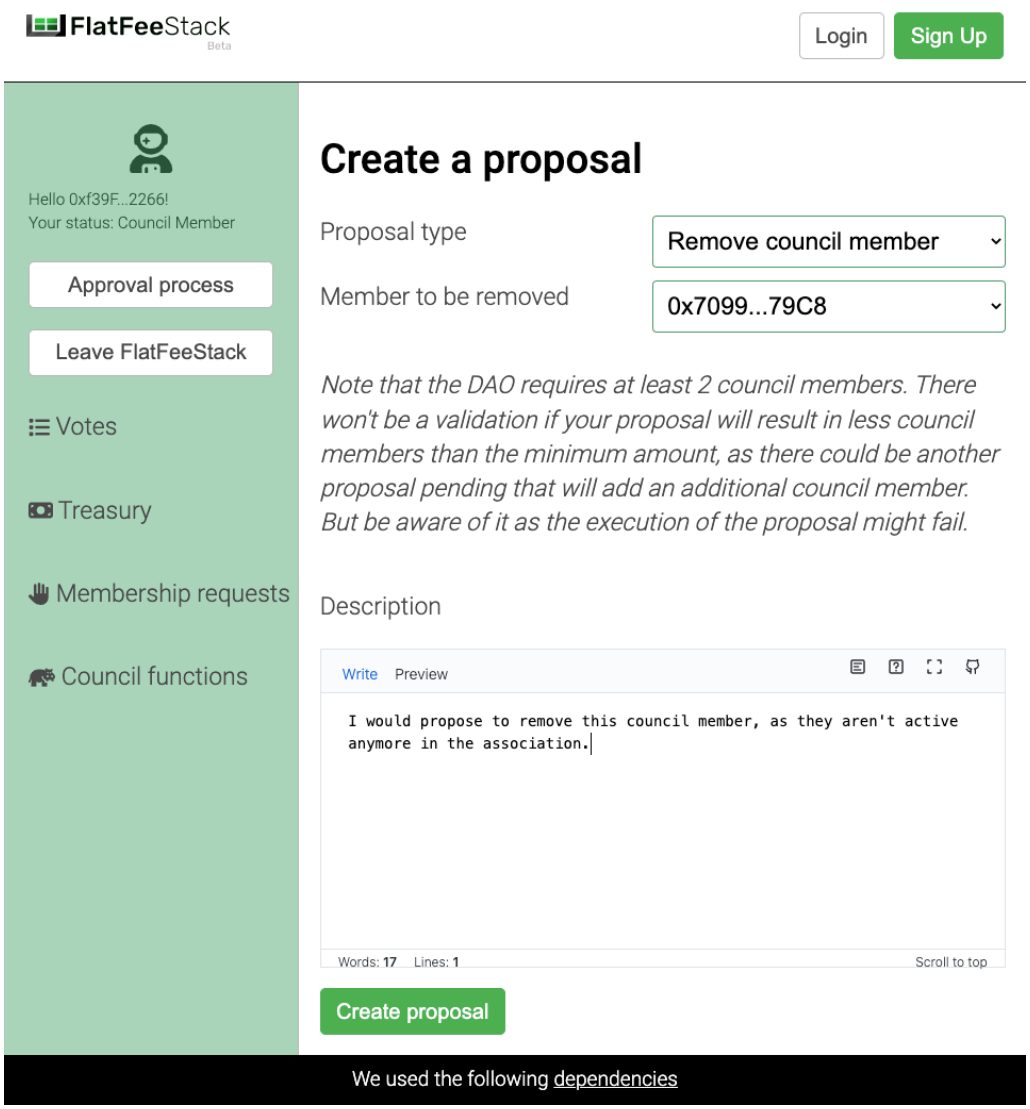

Figure 4.12: Create proposal form, filled out to propose removing a council member

As shown in figure 4.12, the create proposal form allows users to create a new proposal in a simplified form. Usually, they need to provide target addresses that should be called and encoded versions of the functions and parameters that should be called on those addresses (see details in section 3.3.1).

The proposal form abstracts this for typical proposals, like requesting funds from the DAO or removing a member. When choosing a different proposal type in the form, the Svelte component renders a child component that implements the required fields for this proposal. For example, in the case of requesting funds, the child component provides two fields: the target wallet address and the amount. Each child component validates the fields on its own.

If the fields are valid, the child component encodes the information as required by the smart contract. It communicates the updated state via Svelte's two-way binding to the parent component. The parent component blocks the form submission if the child component does not communicate valid encoded function calls. This abstraction allows us to implement new proposal types quickly, as the required data by each proposal type are de-coupled from each other.

Vote view

As shown in figure 4.13, the vote view aims to mimic a ballot vote: Each proposal for the selected voting slot is listed with its proposer and description. A user can click on their voting choice (either disapprove, approve or abstain) and, optionally, give a reason for their vote. At the end of the page, all the votes can be submitted at once.

The view accesses the proposal and voting slot stores, loaded already when the user opens up the start page. However, the view needs to check if the user is eligible to vote, if the voting period for the selected slot is open, and if they have already voted, as votes cannot be revised even if the voting slot is still open. This information is stored locally in the component.

The cast votes are fetched from the corresponding events. However, it is only possible to find votes cast on the Ethereum blockchain by the signer's address, not by proposal identifier. Therefore, in the vote view component, the frontend filters the retrieved events using the proposal identifiers from the selected voting slot. If the connected user has already voted on one proposal, the view is pre-filled with their choice and reason. Submission for proposals where the signer has already cast their vote is blocked to prevent errors.

Execute proposals view

As explained earlier in section 4.3, accepted proposals are not available for immediate execution but need to be submitted to an execution queue. After a specific time, they can be executed.

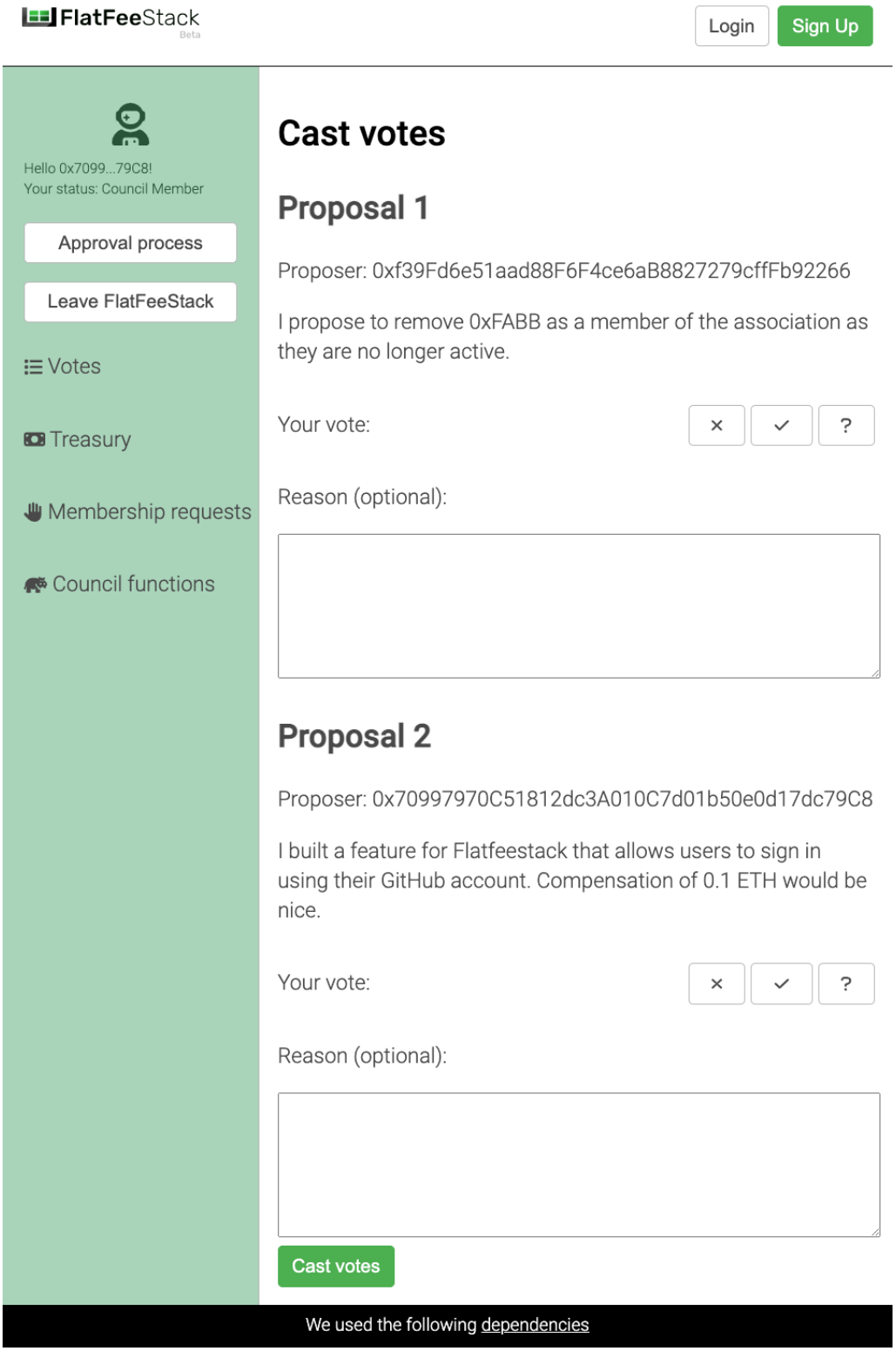

Figure 4.13: View to cast votes.

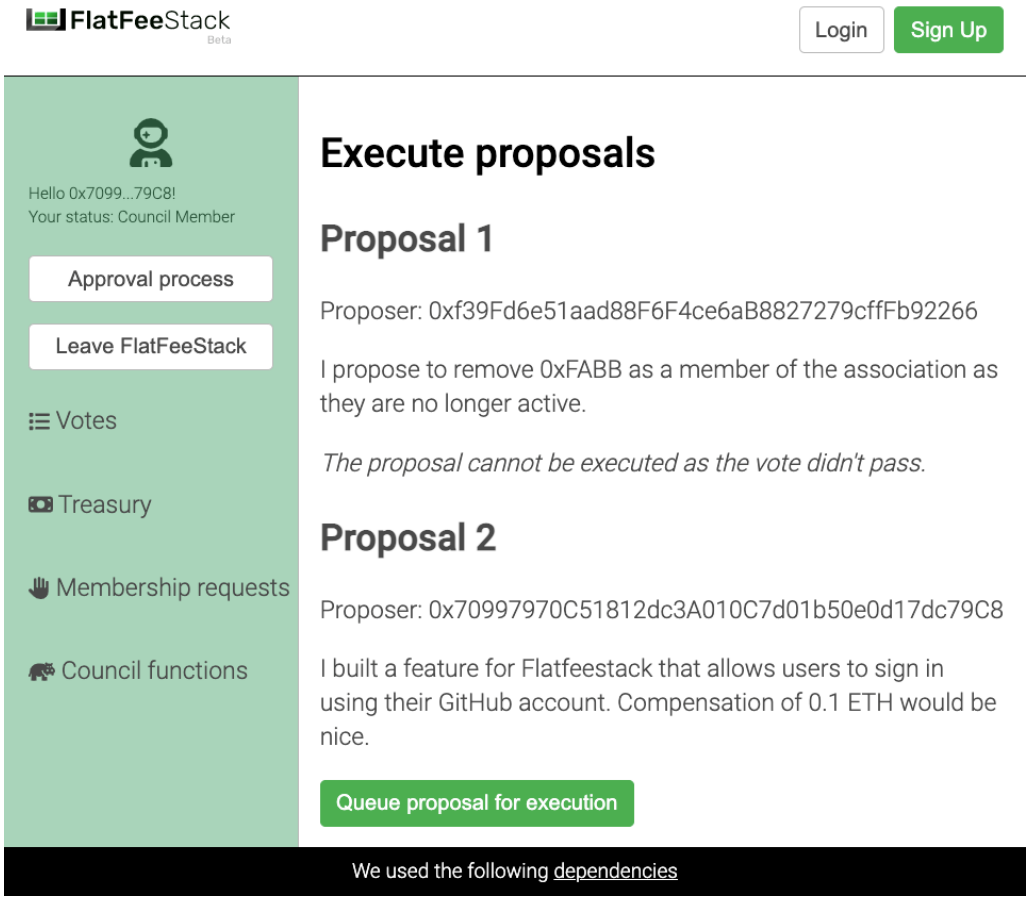

Figure 4.14: View to execute proposals. Vote for the first proposal did not pass, so no further action is available, but the second proposal can be queued.

The Execute proposal view shown in figure 4.14 illustrates this workflow. This view is available after voting for a slot is concluded. It reads data from the voting slot and proposal store but fetches the individual state for each proposal and a property called proposal eta. This property returns the seconds until the proposal can be executed.

The view covers the following states of a proposal:

- Defeated: The view mentions that the vote for the proposal did not pass, and no actions are available on this page.
- The vote for the proposal passed. The view displays a button to send the proposal to the execution queue.
- Queued: If the proposal eta is 0, the view displays a button to execute the proposal. Otherwise, it displays the remaining time.
- Executed: The view mentions that the proposals have been executed already.

Treasury

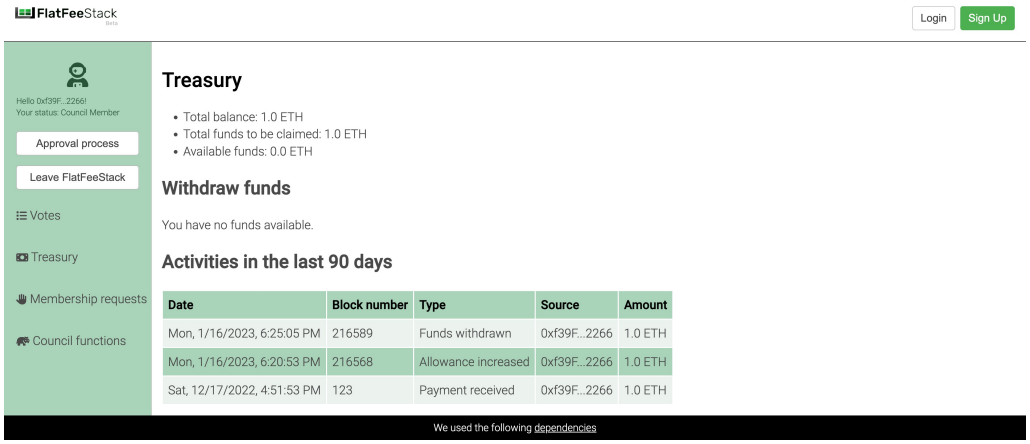

Figure 4.15: Treasury view

The treasury view shown in figure 4.15 displays information from the Wallet contract. It is divided into three sections.

The first part shows the current total of funds in the Wallet. Some of these funds can be locked for members using the `Request funds` proposal, which is displayed in the second. The last number shows how many funds are not allocated and could therefore be used to pay for various things.

The second section allows the user to withdraw funds, if they have any.

The third section overviews all Wallet transactions within the last 90 days. The Wallet contract emits three relevant events that the frontend will fetch for this view.

Note that this Wallet view can only be accessed when somebody is an association member. While the information is publicly available on the blockchain, the treasury's money should not be the primary incentive to join the association. Keeping this information secret in the user interface should help with it.

Council member view

A council member has additional privileges in the DAO. Those additional functions can be managed from the council member's view shown in figure 4.16 and are accessible from the navigation.

This view covers two functions:

- The council member can create a new voting slot. The view will check if the voting slot has been announced a month in advance.

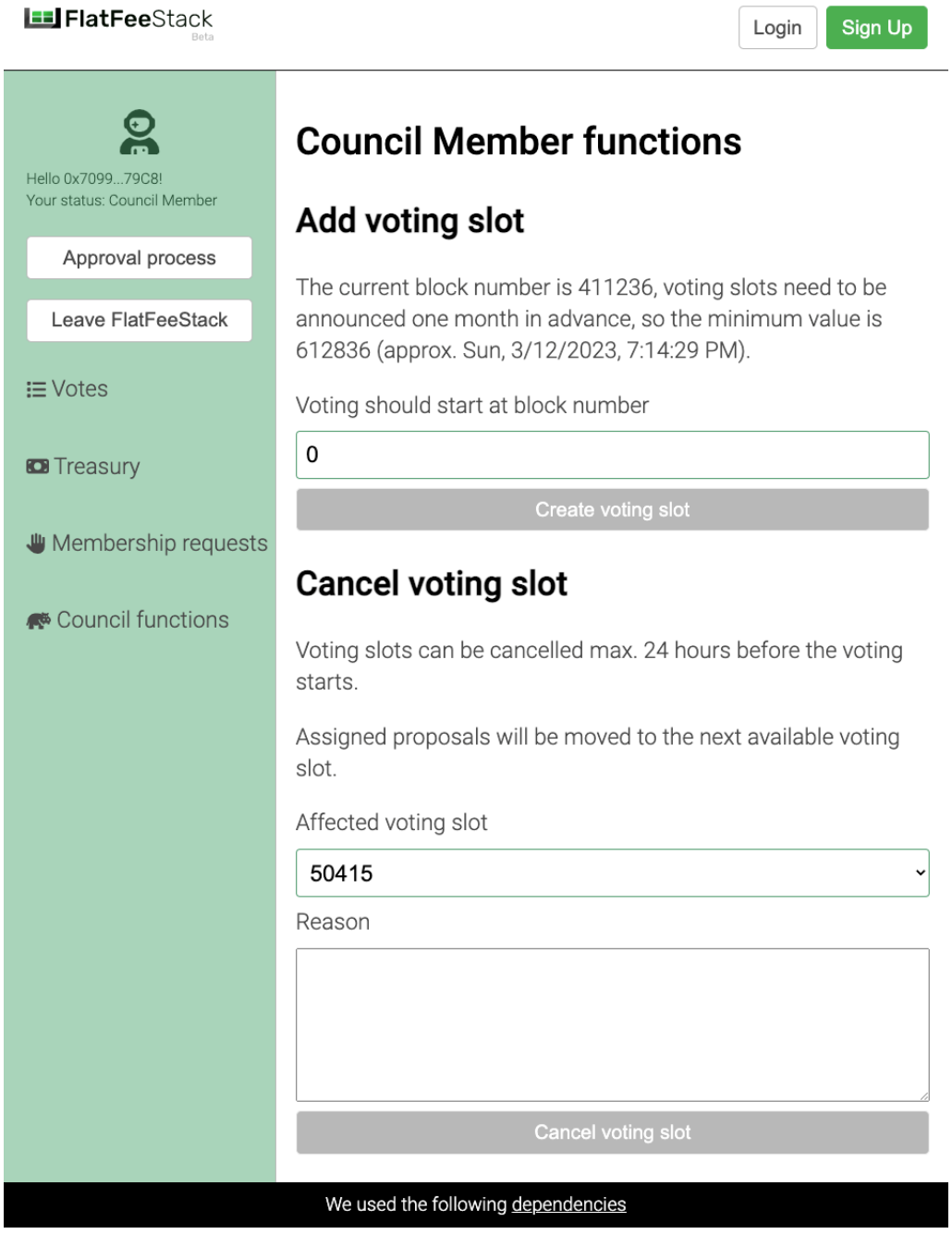

Figure 4.16: Council member view

- The council member can cancel a voting slot and needs to provide a reason for it. The view obtains the available voting slots from the voting slots store. The view also verifies that the cancellation happens not later than one day before the slot.

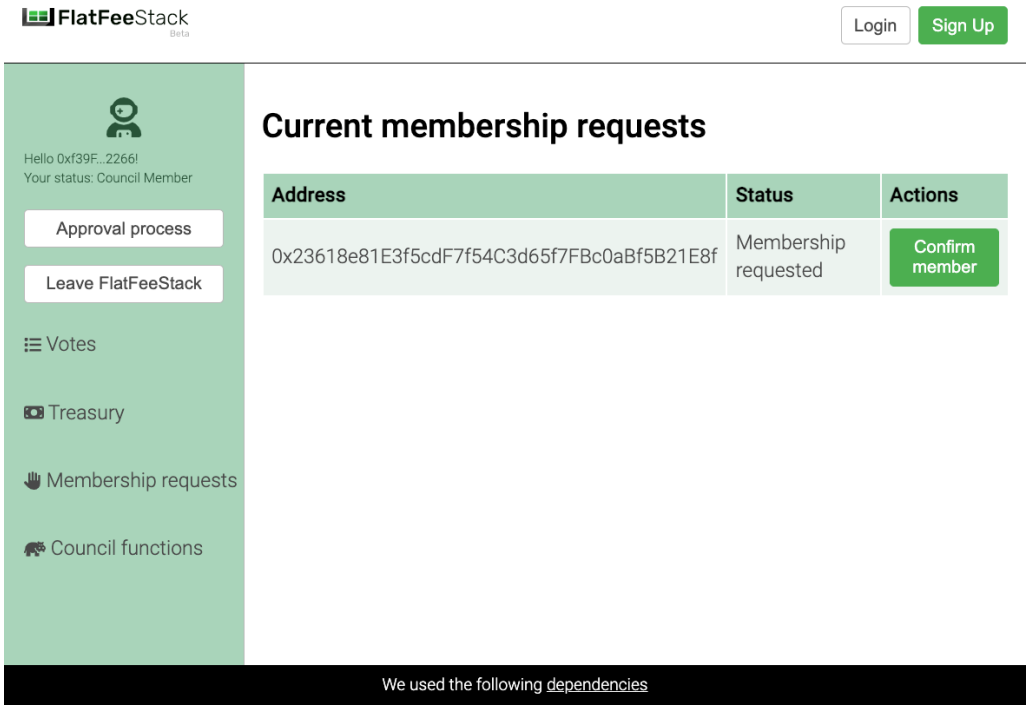

Figure 4.17: Membership requests view

Membership requests view

As outlined in section 4.3.4, membership to the DAO needs to be approved by two individual council members. The membership requests view shown in figure 4.17 lists the current membership requests.

The logic for obtaining membership requests is extensive. Members that need to be approved are either in a member state `requested` or `approvedByOne`. However, those changes between states are not tracked directly on the membership contract, only in the `ChangeInMembershipStatus` events. Therefore, the view collects the these events and applies a JavaScript filter to sort out already confirmed members.

For members in the state `approvedByOne`, the view checks if the current signer is the signer who did the first approval. If the current signer did the first approval, the *approve membership* button in the view is disabled.

Membership status view

From the DAO frontend navigation, members can inspect their current status. This view shown in figure 4.18 is available for everybody but is primarily interesting for people that applied for membership and are waiting for their confirmation.

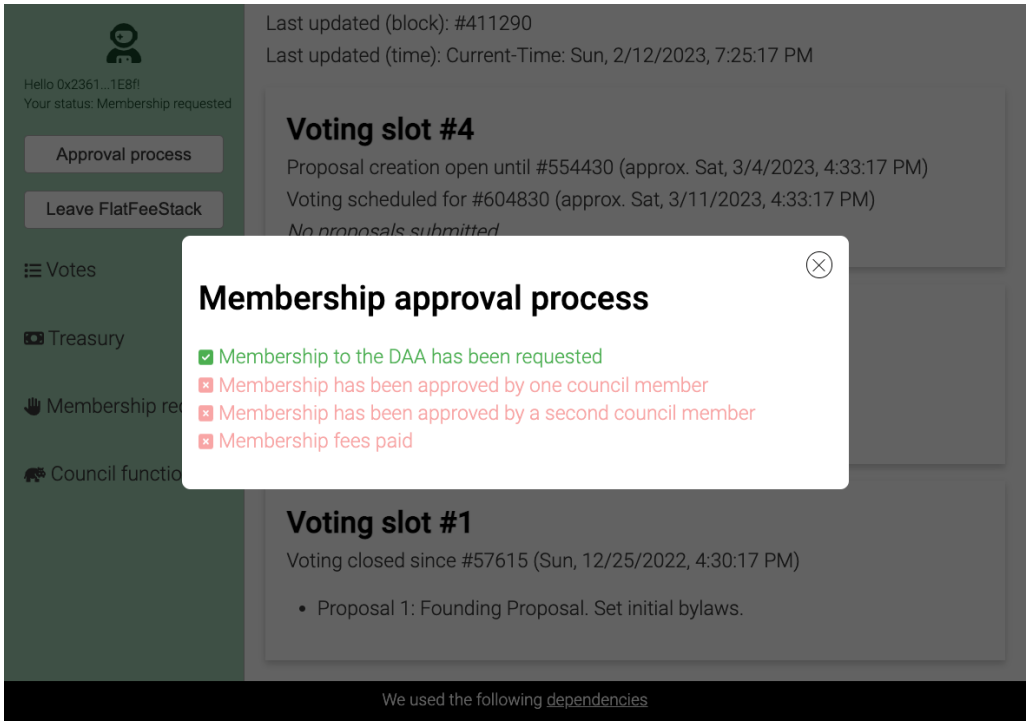

Figure 4.18: Membership status modal

The view accesses the membership status store. If the current signer is already a member, the view fetches the next date to pay the membership fee. The next payment date is stored locally in the component.

The user has two available actions on this view:

- If they did not request membership to the FlatFeeStack DAO, they have a button available to request it.
- If they still need to pay their membership fee, they have a button to pay it.

4.5 Non-functional requirements

This section verifies the non-function requirements listed in 2.2 against the final product.

Functionality: Each story is a functional component and does not break the functionality of others.
Acceptance criteria: Unit Tests run for all subsystems in the main branch.
Result: Partially fulfilled.

The smart contracts were written with unit tests alongside from the first line of code. There weren't any merges to main branch unless the continuous integration passed successfully. In the frontend, no unit tests were written and verification was done manually.

This sometimes resulted in code getting merged that broke functionality in other parts of the application.

Portability: The application must be portable and able to run on multiple platforms or devices without requiring significant modification.
Acceptance criteria: Manual testing on different devices and browsers.
Result: Fulfilled.

This requirement does not apply to the smart contracts. The frontend was frequenty used in different browsers and on different devices. There wasn't any instance where the frontend was broken because of a certain device type or browser.

Extensibility: The application must be easy to maintain and update over time, with clear documentation and support processes.
Acceptance criteria: Time to fix bugs or to install updates.
Result: Fulfilled.

This requirement is fulfilled for now, for both the smart contracts and the frontend. Documentation was added to setup both the smart contracts development environment and the frontend. The real test will follow once other people will continue the work on this project.

Robustness: The application must be robust and able to handle unexpected or invalid input without crashing or behaving unexpectedly.
Acceptance criteria: Manual testing with invalid or malicious input.
Result: Partially fulfilled.

These manual tests were conducted towards the end of the project.

The smarts contract are written to handle a lot of invalid cases. Additionally, the type system on Solidity prevents cases where e.g. a number is expected, but a string is passed.

The frontend handles invalid numbers for the *Create Proposal* form. Any other exception or invalid input neiter recorded nor properly captured. Given Svelte's nature, the frontend won't be working anymore and a hard refresh is needed.

Code quality: The application must be written in clean, well-organized code that is easy to understand and maintain.
Acceptance criteria: Static analysis run with each commit.
Result: Partially fulfilled.

The static analysis was added to the smart contracts and ran on each run of the continous integration.

The frontend does not contain any static analysis tool and the moment. No tool was added during this project.

4.6 Bylaws

The following bylaws were created based on the elaborated solution design from the previous chapter. They are based on the example bylaws from MME ([4]).

4.6.1 Articles

NAME, DOMICILE AND PURPOSE

Art. 1 Name
Under the name of "FlatFeeStack DAO" hereinafter referred to as "DAA" an association within the meaning of Article 60 et seqq. of the Swiss Civil Code exists for an indefinite term.

Art. 2 Domicile
The DAA has its domicile in Zürich.

Art. 3 Purpose
The purpose of the DAA is to directly or indirectly contribute to the success of the FlatFeeStack-Platform with the goal to support and promote open source software and its projects. The DAA can cooperate with or join other organisations that represent the same or similar interests. The DAA can provide services for the benefit of its members and member organisations or third parties and do anything that directly or indirectly promotes the interests of the members.

STRUCTURE AND ORGANISATION

Art. 4 Bodies
DAA's bodies are the:

- DAA Council Member ("Vorstand");
- DAA Ballot Vote ("Urabstimmung");

Art. 5 General Concept of Competences and Duties
The objective of DAA is to establish a decentralised and democratic association with flat hierarchies. Therefore, the DAA Council member ("Vorstand") will have only those competences, which actually require the action of an individual, natural person, such as representation duties or the duty to keep the books. The DAA Ballot

Vote ("Urabstimmung") has those elementary competences, which are mandatory stipulated for an association assembly by Swiss law (such as the change of statutes, the liquidation of the association, and others). The DAA Ballot Vote will be held digital, allowing the proper democratic decision-making required by Swiss law. Additionally, the DAA Ballot Vote shall be the body, which can decide about the "daily business" such as the proposal and support of projects and the funding allocation to projects. The DAA Ballot Vote provides a blockchain-based technical infrastructure to efficiently, democratically and transparently propose and vote.

Art. 6 Underlying Technology

The DAA is technically built with Ethereum Smart Contracts. All voting will take place on this technical infrastructure. The relevant technical functions are hereinafter written in italic. To be able to vote, holding a certain amount of Ether is necessary for every member (transaction fees, gas fees).

Art. 7 DAA Council member (Vorstand)

The DAA has at minimum two (2) council members with the following competencies and duties:

(a) Representing the DAA to the outside world;

(b) Keeping the books and creating the necessary financial statements of the DAA;

(c) Preparing and calling the next DAA Ballot Vote (*setVotingSlot*);

(d) Canceling a DAA Ballot Vote, if they have a good reason for it (*cancelVotingSlot*);

(e) Reviewing member applicants on their eligibility to join the DAA. If the member meets the requirements of Art. 11, the DAA Council member will add their public keys to the member registry (*approveMembership*);

(f) Remove Members who haven't paid their membership fees (*removeMembersThatDidntPay*);

(g) Keeping the member registry (name, address, e-mail);

The term of office for the DAA Council member starts with the foundation of this DAA and ends when he's been replaced via a proposal. If a DAA Council member becomes unable to act or loses his private key, he must be replaced with a new council member in the next DAA Ballot Vote. If all the DAA Council members are unable to act, an extraordinary DAA Ballot Vote will be called and two new DAA Council members must be elected. The personal liability of a DAA Council member is limited to cases of gross negligence.

Art. 8 DAA Ballot Vote ("Urabstimmung")

1. **Competences**
   The DAA Ballot Vote shall be the highest governing body of the DAA. It is chaired by one of the DAA Council members. The objective of the DAA Ballot Vote is to allow every member to propose new DAA projects and to vote on the funding allocation to those projects. The DAA Ballot Vote has the duty of collecting all the votes on the proposals on that vote.

2. **Voting Process**
   The whole voting process is purely digital and blockchain based. The members have one day to vote for every proposal in the Ballot Vote.

3. **Voting Majorities**
   Resolutions shall be adopted by a simple majority of the members participating at the individual DAA Ballot Vote. At least 1/5 of the DAA Members must vote on a proposal and a simple majority to become successful.

4. **Election Process for DAA Council member**
   Any member can propose themselves for candidacy as a council member (*propose & addCouncilMember*). If the proposal is successful the new DAA Council member term starts at the execution of the proposal. Any member can also propose to remove a council member (*propose & removeCouncilMember*)

5. **Convocation**
   The DAA Ballot Vote shall be held in regular intervals. A DAA Council member can set the date at least 1 month before the DAA Ballot Vote (*setVotingSlot*). The DAA Council member will inform the members electronically on the date. A DAA Ballot Vote lasts one full day.

   Every member can propose an extraordinary DAA Ballot Vote (*propose*). If 20% of all DAA members support the extraordinary DAA Ballot Vote, it will take place on the specified date.

6. **Proposals & Agenda Items**
   Every member can submit a proposal to pay Ether to a destination address (the amount can be zero). The funding allocation via the DAA Ballot Vote is binding and technically non-reversible – not even by the DAA Council member.

Proposals from members must be submitted at least 7 days prior to the DAA Ballot Vote using the specific DAO proposal function (*propose*).

MEMBERSHIP

Art. 9 Members & Membership Requirements

Natural persons, legal entities and organizations under public law can request membership of the DAA. Legal entities and organizations under public law shall appoint a representative who exercises membership rights at the DAA Ballot Vote. Every member is responsible to gain the technological know-how to be able to participate on votes on the DAA Assembly. In addition, every member has to assure to have a sufficient amount of Ether for the necessary transaction fees.

Art. 10 Becoming a Member

Everyone who is eligible for a membership can make a request(*requestMembership*). Every member has to do a KYC check. A new member has to be whitelisted by at least two DAA Council members before joining(*whitelistMember*). After the whitelisting and the payment of the membership fee (*payMembershipFee*), the applicant becomes a DAA member and gains voting power for the next ballot vote.

Art. 11 Awareness of Technological & Conceptual Risks

Blockchain is a new technology. The technical or conceptual structure of this DAA and the DAA voting process may have weaknesses, as it is the case with every blockchain project. Moreover, the DAA is dependent on the underlying Ethereum protocol. Therefore, it may be possible that the DAA loses part or the whole of its funds or become incapable of acting. Every member explicitly declares to be aware of and to agree to those risks.

TERMINATION OF MEMBERSHIP

Art. 12 Resignation

DAA members can leave the DAA using *removeMember*. The resignation will be of immediate effect. There is no entitlement to any refund of paid membership fees. The membership fee remains owed in full for the current fiscal year.

Art. 13 Expiration

Membership in the DAA ends automatically:

- upon liquidation of the DAA;
- by the death of the specific member.

The membership fee remains owed in full for the current fiscal year.

Art. 14 Exclusion

Every member of the DAA can be expelled by the DAA Member Community. The exclusion of a member can be proposed by every member (*propose & removeMember*). Members who did not pay their membership fees are excluded automatically without voting process (*removeMembersThatDidntPay*).

The excluded member has no right to an explanation. The membership fee remains owed in full for the current fiscal year.

FINANCES

Art. 15 Membership Contributions and Other Fundraising

The DAA is primarily financed by the contributions of its members (*payMembershipFee*). The Membership fee is 30'000 Gwei. The member contributions will be set initially or can be changed via a proposal (*propose & setMembershipFee*).
In addition, the DAA is financed by a fee from the contributions from the FlatFeeStack-Platform. The membership fees are due in intervals of 365 days after the individual date of accession.

Art. 16 Fiscal Year

The fiscal year is identical to the calendar year.

Art. 17 Liability

The assets of the DAA shall be solely liable for the obligations of the DAA. Personal liability of the members beyond the regularly adopted contributions is excluded.

ASSETS

Art. 18 Ownership Smart Contracts

The DAA has the ownership of following Ethereum Smart Contracts on the Mainnet:

- Name: DAA.sol Address: 0x
- Name: Membership.sol Address: 0x
- Name: Wallet.sol Address: 0x

Art. 19 Ownership Software Code

The DAA has the ownership of all the code who is written and stored on Github `https://github.com/flatfeestack`. All code must be licensed under an open source license.

UPDATE & DISSOLUTION

Art. 20 Update of the Underlying Code

An update of the underlying smart contract code of the DAA can be adopted via the DAA Ballot Vote.

Art. 21 Dissolution & Liquidation

The dissolution of the DAA can be adopted with a proposal (*propose*). The funds will be transferred to one of the DAA Council Members (specified in the proposal), who must carry out the liquidation process. If all council members aren't available anymore, the funds can also be transferred to a normal member.

FINAL PROVISIONS

Art. 22 Entry into Force

These Articles of Association are based on those of the founding meeting of DD.MM.YYYY and were adopted at the occasion of the inital DAA Ballot Vote of DD.MM.YYYY. They enter into force immediately.

4.6.2 Hash and store

Store space is expensive in Ethereum Smart Contracts; therefore, only a hash of the bylaws is stored in the smart contract. The bylaws are stored on Github ([19]). Every time there is a change in this repository, a CI pipeline will run and calculate a SHA256 hash. The calculated hash can be used to make a proposal in the DAA-Contract to change the bylaws.

Chapter 5

Conclusion

This chapter reflects on the goals and outcomes of the project to create an association on the Ethereum blockchain for the FlatFeeStack platform. It begins by summarizing the process of gathering knowledge on blockchain technologies and legal requirements, and designing and implementing the association's structure, smart contracts, and frontend. Finally, the chapter explores potential further developments for the FlatFeeStack association, including a discussion forum, verification of proposal content, and integration of external services.

5.1 Conclusion

This assignment designed and implemented an association compliant with Swiss Law based on blockchain technologies to maintain the FlatFeeStack platform. A set of smart contracts on the Ethereum blockchain abstracts the different activities needed in an association, namely managing members, discussing proposals, and voting. A frontend was implemented to make the smart contract more accessible for people without in-depth knowledge of the blockchain. A set of bylaws were written to connect the legal requirements with the developed application.

At the start of the assignment, the main focus was to gather knowledge on current blockchain developments and the legal requirements for an association under Swiss law. Based on those gatherings, first drafts of the association's structure were created, ultimately leading to the bylaws written as part of the assignment.

The existing Governor framework by OpenZeppelin proved to be a good starting point to comply with the already thriving governance ecosystem on Ethereum. Based on this implementation, additional functionalities were introduced to fulfill the requirements given

by the law. Due to unique requirements regarding votes per member and the approval process, a smart contract still compatible with the Governor's framework was written.

The frontend was integrated into the existing FlatFeeStack application. The framework with Svelte.js and the library to interact with the Ethereum blockchain, ether.js, were already given. The frontend contributed during this assignment resides in its own space within the application and is largely decoupled from the main application. One of the challenges in implementing the frontend was displaying information retrieved from the blockchain in a user-friendly manner while maintaining good performance. A solution was found by implementing caching of information used in different views, adding loading indicators, and creating a user interface draft in advance.

Digitalizing processes given by law in the form of an association proved to be a challenge. However, the work produces in this assignment is a good starting point for launching the FlatFeeStack association.

5.2 Future Work

This section aims to identify potential areas for future work in the FlatFeeStack association. These ideas have been identified based on the current limitations and challenges.

5.2.1 Discussion before creating a proposal

For most matters, a discussion is needed even before creating a proposal. Offering this functionality can serve multiple purposes: Members of the association could flesh out an idea together. A single member could post a draft of their proposal to see if they will get the needed majority on voting day. Questions could be clarified before creating the proposal to make sure the members have a general understanding of what should be achieved with a proposal.

A democratic institution should offer its members all the possibilities to allow them to form an opinion independently. With the current implementation, this is very limited.

A possibility could be integrating a forum into the FlatFeeStack site, where members connect with their MetaMask accounts. However, the data for the forum should be saved outside the blockchain to save costs. The vital information, which is the content of a proposal, will remain on the blockchain.

5.2.2 Verify proposal content

The content of a proposal, especially targets and functions that will be called, is intentionally free-text to allow the DAO to perform any blockchain-based actions. However, this is also a security risk: A member could describe that the proposal only sends 1 Ether to an unknown sender, but in reality, it is 100 Ether.

A copy of the used parameters and the generated proposal data could be sent to an off-chain service when creating a proposal through the frontend. This off-chain service could store this information. The frontend could talk with this service and mark the proposal as *safe* if all parameters were verified.

Proposals that are sent directly to the chain could be analyzed in hindsight as follows:

- The off-chain service needs to know the contracts related to the DAO (the DAO itself, Membership, Wallet, and Timelock) and their ABIs.
- The mentioned off-chain service could listen to the *Proposal created* event. Once a new proposal is created, it checks the targets to see if it matches any known contracts.
- If it matches any known contract, it can decode the call data. The decoding works by calculating the hash of every function using keccak256 and reducing it to four bytes. It is a known function if those four bytes match the start of any call data. If it is a known function, the parameters can also be derived. This information can be saved to the same database and the proposal can be marked as *safe*.

5.2.3 Member register

Associations keep a register of their members. While the FlatFeeStack association has a list of members in the form of Ethereum addresses, it does not have information about their names or address.

If somebody wants to become a council member, those *real-world* pieces of information would be needed, as they represent the association to the outside world However, the question remains if there should be a register of all members, consisting of their name, postal address, and Ethereum wallet address. How would one save this information in a decentralized fashion without accidentally exposing these personal details? Should the verification be done automatically or *old-fashioned*, e.g., using a Zoom call?

5.2.4 Funding from the platform to the association

Section 4.1 mentions that 1% of all donations on the FlatFeeStack platform will be redirected to the FlatFeeStack associations. This mechanism has not been implemented yet.

Glossary

bylaws Bylaws ("Statuten") containing the main rules and regulations governing an association and its activities, including its goals, structure, membership, decision-making processes, and the responsibilities of its organs. iii, 5, 8–10, 22, 25, 41, 46, 47

DAA Decentralized Autonomous Association (DAA) follows the concepts of a DAO but with the requirement to be compliant with Swiss law 2–5, 7–10, 14, 17, 52

DAO A Decentralized Autonomous Organization (DAO) is a type of organization that is run using a set of rules encoded as smart contracts on a blockchain. These rules allow the organization to operate in a transparent and democratic manner, without the need for a central authority or traditional management structure. i, iv, 1, 2, 7, 10, 11, 14, 15, 17, 18, 21, 25, 28–31, 33, 36, 38, 39, 41, 49, 84, 85

DOM When a web page is loaded, the browser creates a Document Object Model of the page. JavaScript can interact with the DOM to manipulate it. 29

ECTS ECTS stands for the European Credit Transfer and Accumulation System. It is a standardized system used by higher education institutions in the European Union and other countries to evaluate, transfer, and recognize academic credits. 2

EVM Deterministic runtime environment and computing engine in which the smart contracts provided on Ethereum are executed 18

KYC Know your customer (KYC) is a process to verify the identity of a person. 4, 11

List of Figures

List of Listings

Bibliography

[1] "On-chain governance open zeppelin," https://docs.openzeppelin.com/contracts/4.x/governance, Open Zeppelin, accessed: 2022-09-30.

[2] Schweizerisches zivilgesetzbuch. https://fedlex.data.admin.ch/eli/cc/24/233_245_233. Bundesversammlung der Schweizerischen Eidgenossenschaft. Accessed: 2022-09-27.

[3] Decentralized autonomous organization. https://en.wikipedia.org/wiki/Decentralized_autonomous_organization. Wikipedia. Accessed: 2022-10-05.

[4] "Model articles of association of a decentralized autonomous association (daa)," https://fs.hubspotusercontent00.net/hubfs/4431201/200511_MME_DAA.pdf, MME Compliance AG, accessed: 2022-09-30.

[5] Q. Yu, "Ethcc 2019," in *Beyond the technology, how about building a legally-compliant DAO?*, July 2019.

[6] Decentralised autonomous association switzerland – daas. https://github.com/validitylabs/daa. Validity Labs. Accessed: 2022-09-26.

[7] Pascal Knecht, Andy Pfister, "Meeting with mme, october 11, 2022, meeting minutes," October 2022.

[8] On-chain governance. https://www.investopedia.com/terms/o/onchain-governance.asp. Investopedia. Accessed: 2022-09-28.

[9] A. Lesi and M. Endres, "Kryptowährungen als zahlungsmittel bei flatfeestack," April 2022.

[10] Proxy patterns. https://blog.openzeppelin.com/proxy-patterns/. OpenZeppelin. Accessed: 2022-12-06.

[11] Upgrading smart contracts. https://ethereum.org/en/developers/docs/smart-contracts/upgrading/. Ethereum. Accessed: 2022-12-06.

[12] Hardhat - getting started. https://hardhat.org/hardhat-runner/docs/getting-started#overview. Hardhat. Accessed: 2022-12-19.

[13] Hardhat - testing contracts. https://hardhat.org/hardhat-runner/docs/guides/test-contracts. Hardhat. Accessed: 2022-12-19.

[14] ether.js meta classes. https://docs.ethers.io/v5/api/contract/contract/. ether.js. Accessed: 2022-12-06.

[15] hardhat-deploy. https://github.com/wighawag/hardhat-deploy. wighawag. Accessed: 2022-12-19.

[16] How to avoid making time-based decisions in contract business logic? https://ethereum.stackexchange.com/questions/117813/how-to-avoid-making-time-based-decisions-in-contract-business-logic/117874#117874. Ethereum StackExchange. Accessed: 2022-12-19.

[17] The merge ethereum: an unforeseen effect on block time. https://gettotext.com/the-merge-ethereum-an-unforeseen-effect-on-block-time/. get to text. Accessed: 2022-12-19.

[18] “Metamask under the hood—not just a crypto wallet,” https://chainstack.com/metamask-behind-the-scenes-not-only-a-crypto-wallet/, Chainstack, accessed: 2022-12-21.

[19] Flatfeestack bylaws. https://github.com/flatfeestack/bylaws. FlatFeeStack. Accessed: 2022-12-03.