FlatFeeStack Go-Live

Bachelor's Thesis

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

Spring Term 2023

Authors

Pascal Knecht & Andy Pfister & David Kalchofner

Advisor

Dr. Thomas Bocek

External Co-Examiner

Dr. Guilherme Sperb Machado

Internal Co-Examiner

Prof. Stefan F. Keller

June 14, 2023

Abstract

The FlatFeeStack platform offers a convenient and transparent way for individuals and companies to show their support for open-source projects. It operates on a simple model where users pay a fixed fee of $125 annually. Through FlatFeeStack, contributors to open-source projects receive funds based on their contribution level, measured by analyzing Git commits. Payments can be made using credit cards, and the platform supports payouts using cryptocurrencies.

The platform has been in development for several years now. The platform is primarily feature-complete, so it should be opened to the public. A platform-as-a-provider was evaluated as part of the thesis, and a continuous deployment strategy leveraging GitHub Actions was implemented. The platform uses Prometheus and Grafana for monitoring, which allows to observe both infrastructure and application-relevant metrics. The existing decentralized autonomous organization (DAO) is now accompanied by a forum where members can discuss proposals before they are written onto the blockchain. A complete overview of all the functionality was created, and bugs were fixed that were discovered during the creation of this overview.

On June 2nd, the platform was silently launched, and initial contributions to open-source projects were distributed on the same day.

Keywords: Blockchain, Fintech, Open-Source, Sponsoring.

Executive Summary

Initial Situation

FlatFeeStack is a platform that allows companies and individuals to sponsor open-source projects. Compared to other sponsoring platforms, FlatFeeStack has a fixed annual subscription of $125 instead of paying for each project individually. FlatFeeStack will analyze the contribution of each project member using Git metrics and distribute the funds accordingly. Payouts are done in cryptocurrencies to keep the payout process simple and transparent.

The platform has been in development for several years but is primarily feature-complete. After adding these missing features, the platform should be made available to the public, using a cloud provider to keep operation efforts low.

Procedure & Technologies

The project started by conducting functionalities already built into the platform, identifying missing or incomplete features. A list of tasks to complete was compiled in discussions with the advisor and external co-examiner, and a go-live date was set.

After an evaluation phase, DigitalOcean was chosen as a platform provider for the production instance. The different microservices which compose the FlatFeeStack platform were deployed as services under a common domain. For a simple deployment process, Docker images are built from GitHub Actions and pushed to the DigitalOcean registry, where the services pull new images if requested. Metrics are collected using Prometheus and displayed using Grafana, completing a modern, cloud-native deployment.

The existing decentralized autonomous organization (DAO) should receive a new component to discuss proposals. The requirements were unique, so a new microservice using Go was implemented. The forum allows users to create posts and comments. The forum observes the ETH blockchain, taking action when certain events happen in the DAO, like when a new proposal gets created.

As different papers have been written about FlatFeeStack, as well as the advisor and external co-examiner working on the platform in their free time, a complete overview of the platform's features needed to be composed. These are documented now in the form of use case diagrams. A test plan was created to ensure those use cases worked, as well as bugs fixed that occurred during testing.

Final Thoughts

The silent launch of the platform, accomplished on June 2nd, was successful. The platform is available at flatfeestack.io. The first contributions to open-source projects were even distributed the very same day.

Given the time constraint, some work had to be excluded. For example, some major library updates still need to be done. This was either due to them being released late in the project (hardhat-toolbox v3) or other dependencies not yet ready for the new version like neow3j not having support for Gradle v8. Cryptocurrency pay-ins still need to be activated, as the integration with the respective payment provider was not extensively tested. More future work is documented in a later chapter (5.2).

The platform is now in good shape for potential users and developers to continue improving it. Users benefit from the changes and improvements made in the frontend. The developers benefit from a good code base with up-to-date libraries and the designed continuous integration and continuous deployment. The chosen deployment with GitHub Actions, DigitalOcean, containers, and the monitoring infrastructure with Grafana and Prometheus is considered state-of-the-art in the industry. The forum component will help in the daily business of the DAO, allowing members to communicate with each other easily. The use case overview will help future developers to orient themselves in the system.

Acknowledgment

We want to thank the following people for helping with this Bachelor's thesis:

- Dr. Thomas Bocek and Dr. Guilherme Sperb Machado for their guidance and supervision this semester.
- Michael Bucher for explaining the payout signature algorithm to us and updating the development configuration for the NEO payout contract.

Contents

Chapter 1

Introduction

The FlatFeeStack website makes donating to open-source projects more accessible and transparent. Open-source refers to software, code, or any intellectual property that is freely available, accessible, and adaptable to anyone Developers can support any open-source project for an annual fee of $125, with the donation being equally split among them. The donation is divided proportionately if a developer chooses to support multiple projects. This flat fee per developer makes budgeting easier for companies, with a budget of $1250 for ten developers. There is no need for organizational overhead in selecting which projects to support, as developers can choose the libraries and frameworks that make their work more efficient. Additionally, the supported projects of the developers can serve as an indicator for the company of which pieces their IT landscape is built. The funds are distributed based on the contribution of each developer, with the FlatFeeStack application calculating a contribution score based on lines of code and other metrics. Another advantage is that donation payouts are made with cryptocurrencies. Compared to a classic bank transfer, crypto currencies ensure transparency in the payout process and allow everybody to retrieve their funds promptly.

1.1 Assignment

The primary assignment of this thesis is to launch the FlatFeeStack platform. This objective includes bringing it to a polished state and adding missing functionalities.

1.2 Basic Conditions

This work was done as part of a Bachelor's thesis (Bachelorarbeit). A time budget of 1080 hours is reserved for the work on this assignment and will be rewarded with twelve 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

This section contains all functional requirements for the FlatFeeStack platform. Section 2.1.1 documents the different personas that interact with the FlatFeeStack platform. Section 2.1.2 documents the existing features. Section 2.1.3 describes the functional requirements that yield from the primary assignment: launching the FlatFeeStack platform.

2.1.1 Persona

- **Open-Source Contributor:** Wants to claim their earnings for their contributions to open-source projects.
- **Company:** Wants to sponsor open-source projects they use in their products.
- **Individual:** Wants to sponsor open-source projects they use.
- **Developer:** Wants to participate in the further development of the FlatFeeStack platform.
- **Admin:** Wants to do administrative tasks for the FlatFeeStack platform, for example, monitoring.

For the DAO, different personas are used to describe the use-cases. Any of the personas listed above can be one of the following personas in the DAO [1]:

- **User:** A user is a person who is not already part of the DAO, but has the intention of joining it.
- **Member:** A member is part of the DAO. 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

The use cases help display FlatFeeStack's functionality and what different components do in more detail. Furthermore, the use cases are well suited as an input for a test plan. A similar approach to the C4 Model[1] notation is used to show a broad overview and go further into detail about every component. A similar approach like with the C4 Model notation is used, to show a broad overview and go further into detail of every component.

[1]https://c4model.com/

Top Level View

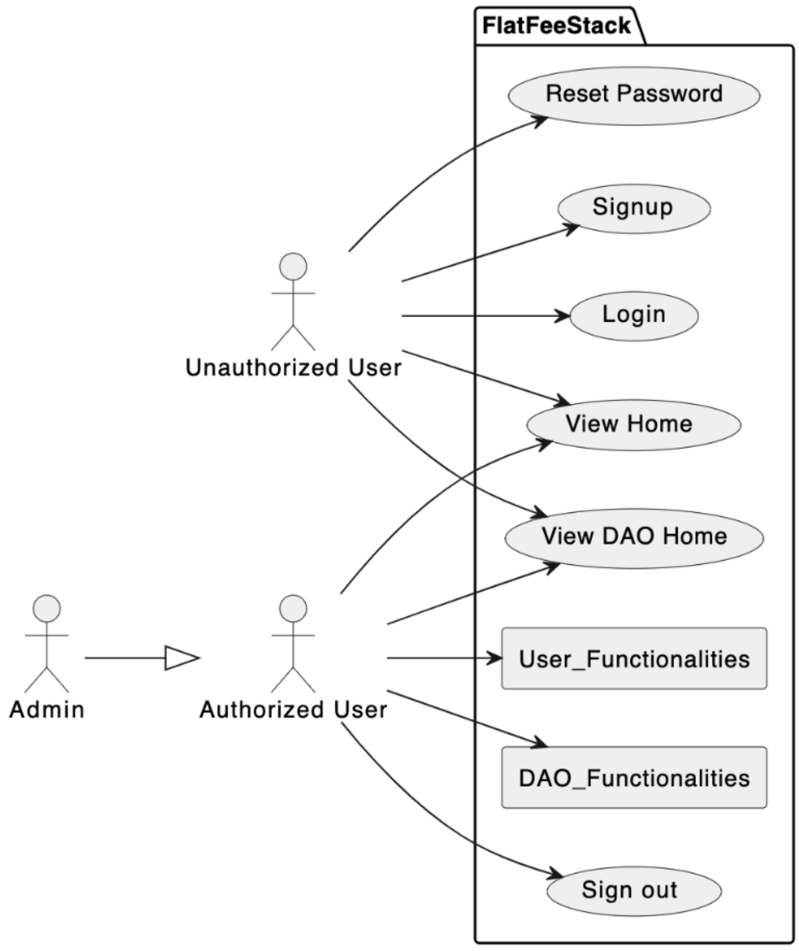


Figure 2.1: Use case diagram top level view.

Nr.	Name	Description
1	Reset Password	User can reset their password by entering their email and setting a new password.
2	Sign Up	User can signup for the application by entering their email and setting a password.
3	Login	User can login if an account already exists.
4	View home	Any user can visit the FlatFeeStack home site.
5	View DAO home	Any user can visit the DAO home site.
6	User functionalities	More details in section: User Functionality
7	DAO functionalities	More details in section: DAO Functionalities [1]
8	Sign Out	User can sign out again if logged in.

Table 2.1: Description for the top level view use cases.

Figure 2.1 displays a zoomed out overview of the FlatFeestack use cases with Table 2.1 providing the accompaning use cases.

User Functionality

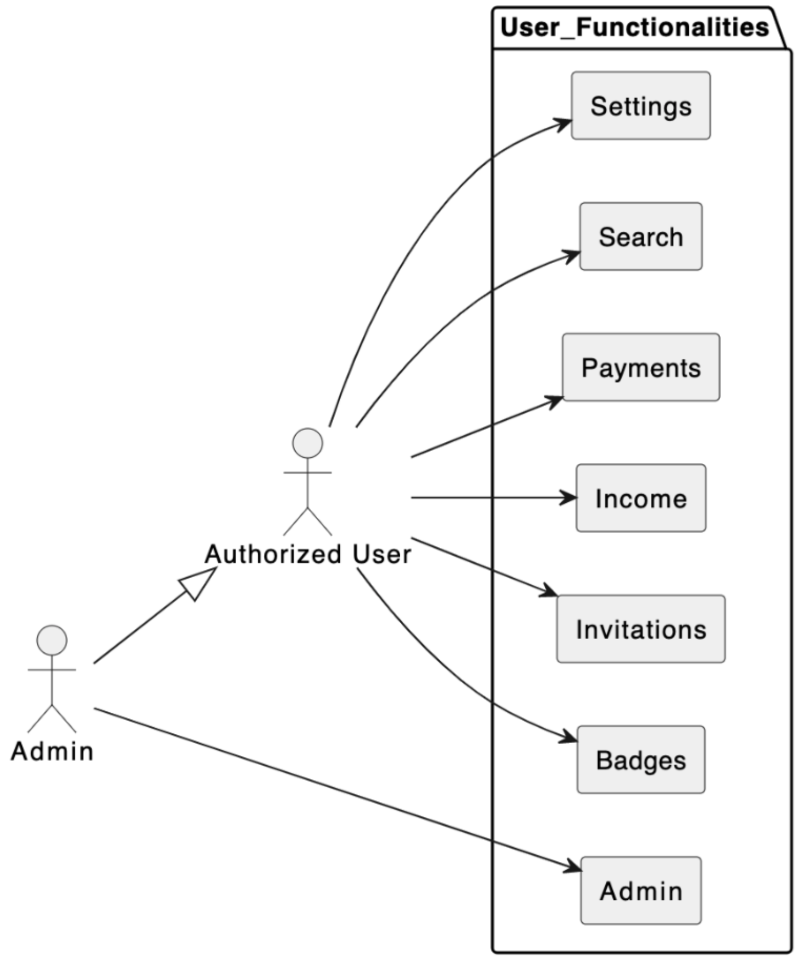


Figure 2.2: Use case diagram with user functionality overview.

Nr.	Name	Description
1	Settings	User can access the settings page and its functionalities. More under User Functionality - Settings
2	Search	User can access the search page and its functionalities. More under User Functionality - Search
3	Payments	User can access the payments page and its functionalities. More under User Functionality - Payments
4	Income	User can access the income page and its functionalities. More under User Functionality - Income
5	Invitations	User can access the invitations page and its functionalities. More under User Functionality - Invitations
6	Badges	User can access the badges page and its functionalities. More under User Functionality - Badges
7	Admin	Admin can access the settings page and its functionalities. More under User Functionality - Admin

Table 2.2: Description for the user functionalities use cases.

Figure 2.2 shows the overview of the user functionality use cases with Table 2.2 providing the accompaning use cases.

User Functionality - Settings

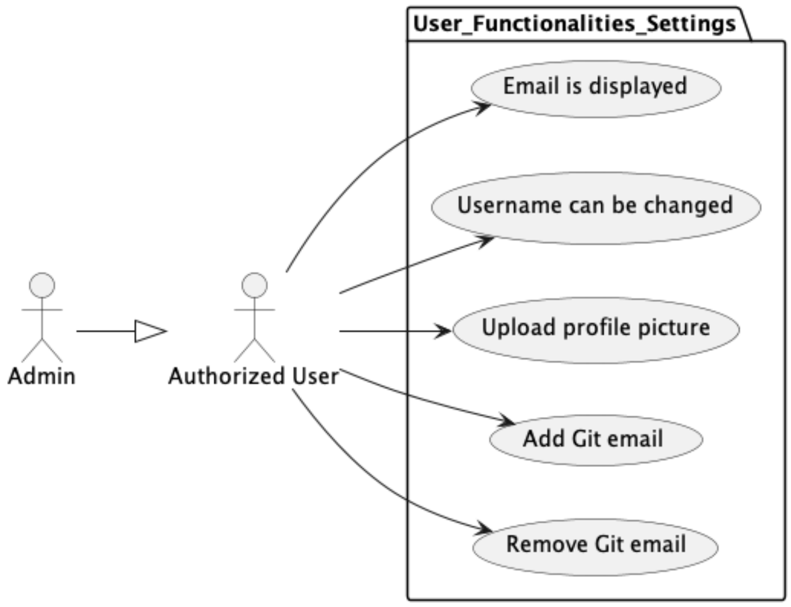


Figure 2.3: Use case diagram user functionality search.

Nr.	Name	Description
1	Email is displayed	User can view their email address in the settings.
2	Username can be changed	User can change their username.
3	Upload profile picture	User can upload and change their profile picture.
4	Add Git email	User can add an email address that will be used to find Git contributions.
5	Remove Git email	User can remove their added Git email addresses.

Table 2.3: Description for the settings use cases.

The use case diagram for the user settings page are provided in Figure 2.3 and the according description is in Table 2.3.

User Functionality - Search

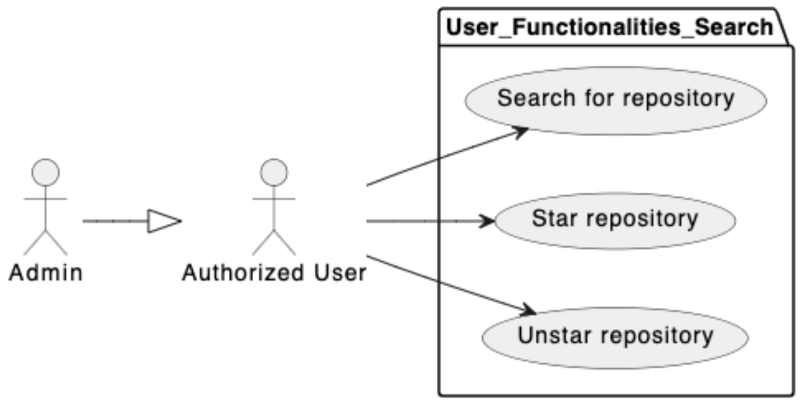


Figure 2.4: Use case diagram user functionality search.

Nr.	Name	Description
1	Search for repository	User can search for any repository on GitHub that they potentially want to support.
2	Star repository	User can star (support) any amount of repositories.
3	Unstar repository	User can unstar (stop supporting) a repository at any time.

Table 2.4: Description for the search use cases.

Figure 2.4 provides a use case diagram for the search page and Table 2.4 describes the use cases in further detail.

User Functionality - Payments

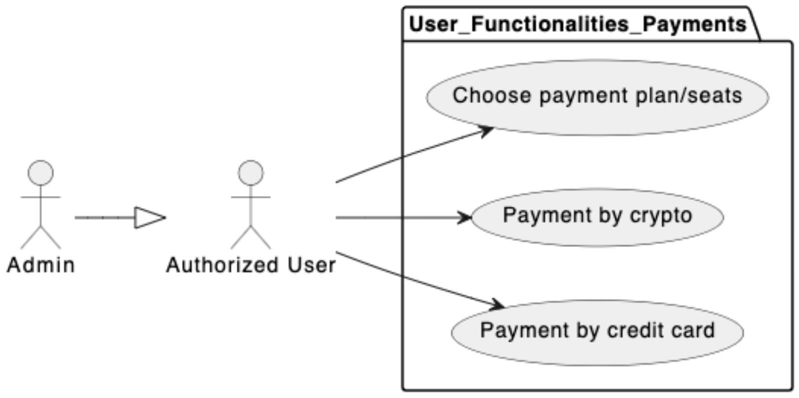


Figure 2.5: Use case diagram user functionality payments.

Nr.	Name	Description
1	Choose payment plan/seats	User can choose between a one and a five year payment and decide how many seats (voting power) they want to purchase.
2	Payment by crypto	User can do the payment in crypto (ETH or NEO).
3	Payment by credit card	User can do the payment via credit card (via Stripe[1]).

Table 2.5: Description for the payments use cases.

The payment page use case diagram is displayed in Figure 2.5 with the detailed use case information in Table 2.5.

User Functionality - Income

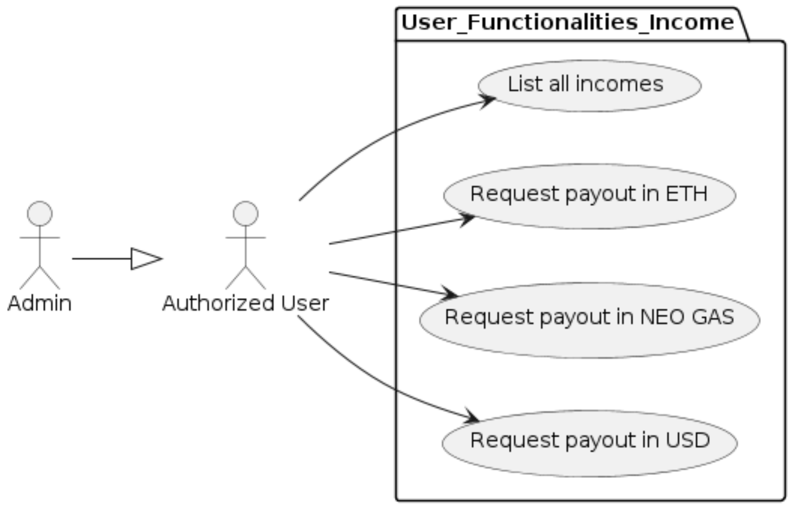


Figure 2.6: Use case diagram user functionality income.

Nr.	Name	Description
1	List all income	User can list all income they received through contributions.
2	Request payout in ETH	User can request a signature to withdraw their earnings in ETH from the respective smart contract.
3	Request payout in NEO GAS	User can request a signature to withdraw their earnings in NEO GAS from the respective smart contract.
4	Request payout in USD	User can request a signature to withdraw their earnings in USD from the respective smart contract.

Table 2.6: Description for the income use cases.

Figure 2.6 shows the use case diagram for the income page and detailed information is provided in Table 2.6.

User Functionality - Invitations

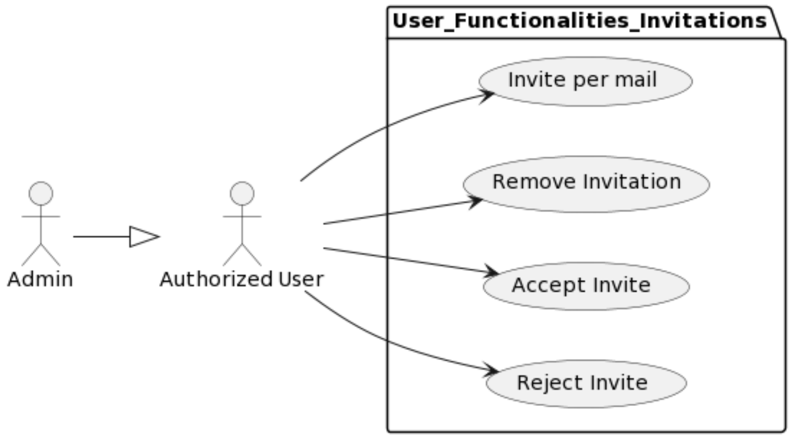


Figure 2.7: Use case diagram user functionality invitations.

Nr.	Name	Description
1	Invite per email	User can invite another user via email.
2	Remove invitation	User can remove an invited user again.
3	Accept invite	User can accept an invitation.
4	Reject invite	User can reject (remove) an invitation.

Table 2.7: Description for the invitations use cases.

The use case diagram for the invitations page is visible in Figure 2.7 and accompanying Table 2.7 provides a description for each use case.

User Functionality - Badges

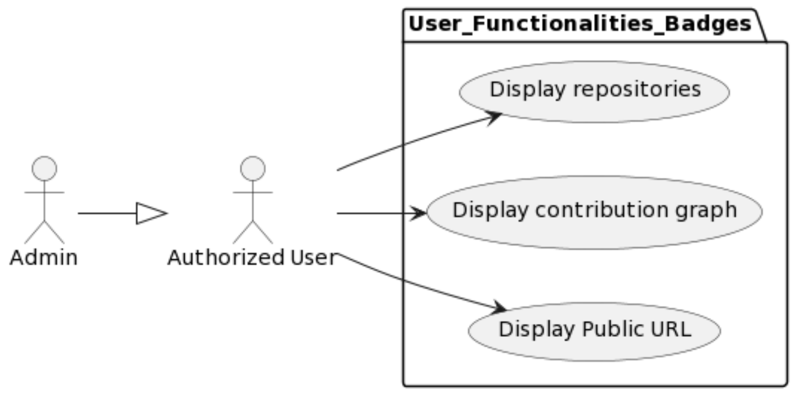


Figure 2.8: Use case diagram user functionality badges.

Nr.	Name	Description
1	Display repositories	User can see their starred repositories in a list.
2	Display contribution graph	User can see the contribution details of a repository by clicking on a button in the list.
3	Display contributions	User can list all contributions they have made.
4	Display Public URL	User can access and share their public badge URL with information about them and their contributions.

Table 2.8: Description for the badges use cases.

Figure 2.8 displays a zoomed out overview of the FlatFeeStack use cases with Table 2.8 providing the accompaning use cases.

User Functionality - Admin

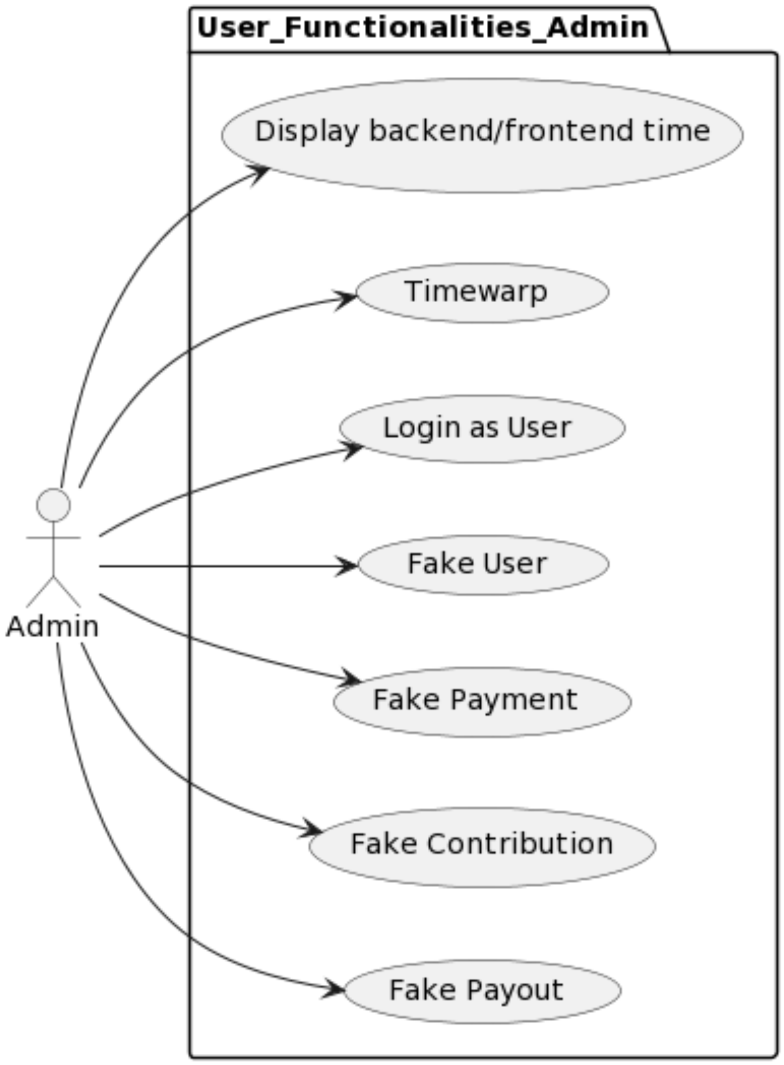


Figure 2.9: Use case diagram user functionality admin.

Nr.	Name	Description
1	Display backend / frontend time	Admin can view the frontend and backend time as it can differ if a timewarp took place.
2	Timewarp	Admin can timewarp into the future for testing purposes.
3	Login as User	Admin can login as any user on the site.
4	Fake User	Admin can create a new user.
5	Fake Payment	Admin can create a fake payment for testing purposes.
6	Fake Contribution	Admin can fake a contribution for testing purposes.
7	Fake payout	Admin can fake a payout of contributions for testing purposes.

Table 2.9: Description for the admin use cases.

Figure 2.9 presents the admin page use case diagram. More information on the individual use cases is in Table 2.9.

DAO Functionalities [1]

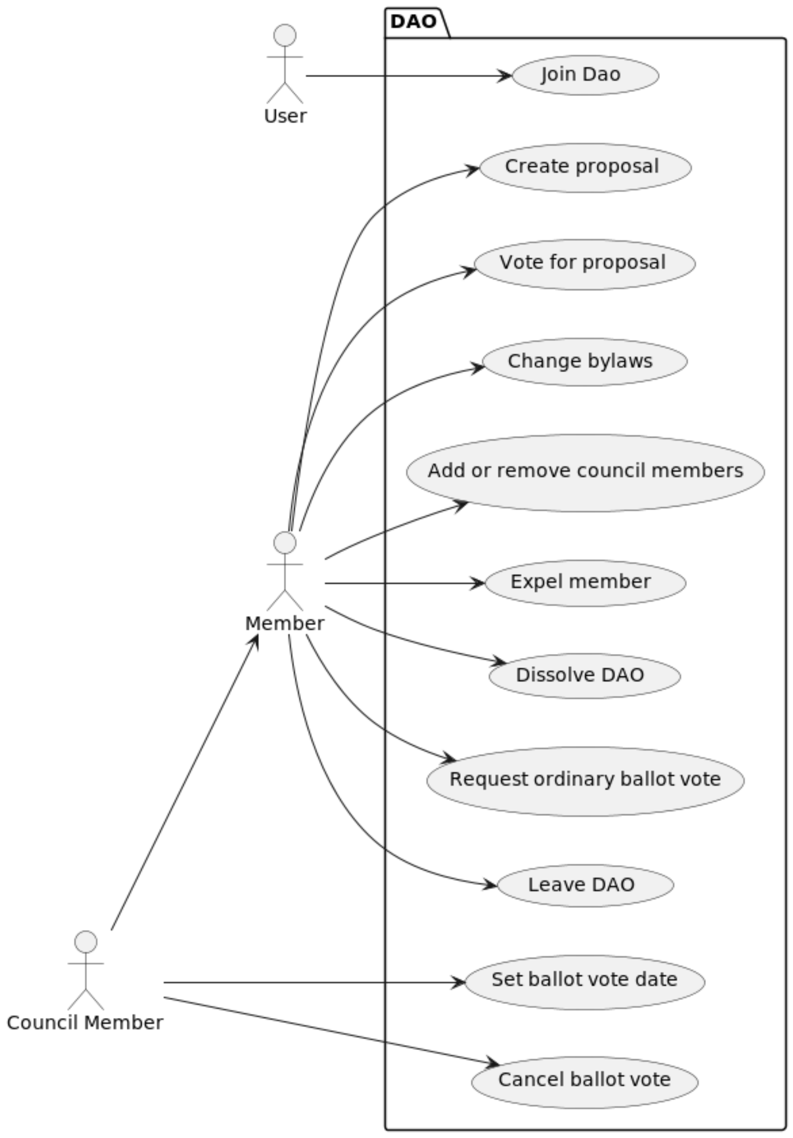


Figure 2.10: Use case diagram DAO.

Nr.	Name	Description
1	User can join DAO	A User can request to be a member of the DAO. It is necessary to do a KYC validation before a user is allowed to join.
2	Member can leave DAO	A member can leave the association with immediate effect.
3	Member can create a proposal	A member can create a proposal.
4	Member can vote for a proposal	A member can vote for a proposal. A member can only vote once per proposal.
5	Member can propose and vote for new bylaws	If a member wants to change the bylaws, they can create a proposal.
6	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.
7	Member can propose and vote to dissolve the DAO	A member can create this specific proposal to dissolve the DAO. The specific steps to dissolve are written in the bylaws. It requires a 20% quorum and a simple majority
8	Member can propose and vote to expel a member	A member can create a proposal to expel a member.
9	Council member can 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.
10	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.
11	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.

Table 2.10: Description for the DAO use cases.

Figure 2.10 exhibits the use case diagram for the DAO and the descriptions are listed in Table 2.10.

2.1.3 Requirements for this Thesis

As mentioned in section 1.1, the primary objective of this thesis is to launch the FlatFeeStack platform. This objective includes the following stories:

1. Merge the existing Git repositories of the different services into one mono-repository.

2. Evaluate a PaaS provider to deploy the production instances of the different services.
3. Deploy the FlatFeeStack platform to the chosen PaaS provider.
4. Evaluate and set up a solution to monitor the health of the FlatFeeStack platform.
5. Evaluate a discussion platform for the FlatFeeStack DAO.
6. Set up or implement the chosen discussion platform.
7. Find and squash bugs by verifying the existing functionalities on the platform.

2.2 Non-Functional Requirements

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

1. **Functionality**: Each story represents a functional component and does not disrupt the functionality of others.
   Acceptance criteria:
   - Unit Tests are executed for all subsystems in the main branch.
   - The application passes all acceptance tests before release.
   - User feedback and bug reports are regularly reviewed and addressed to improve functionality.
   - Changes to functionality are carefully communicated and documented to avoid confusion or unexpected behavior.
2. **Portability**: The application must be portable and capable of running on multiple platforms or devices without significant modification.
   Acceptance criteria:
   - Manual testing conducted on different devices and browsers.
   - The application does not use components or services that are only available on a single platform.
   - The application's user interface adapts seamlessly to different screen sizes and resolutions.
3. **Extensibility**: The application must be easily maintainable and updatable over time, with clear documentation and support processes.
   Acceptance criteria:
   - Time required to fix bugs or install updates is within the specified limits.
   - The application's architecture supports modular design and component reusability.
   - Regular code reviews are conducted to ensure adherence to extensibility best practices.
4. **Robustness**: The application must be robust and able to handle unexpected or invalid input without crashing or exhibiting unexpected behavior.
   Acceptance criteria:
   - Manual testing performed with invalid or malicious input.

- Automated testing performed with invalid or malicious input.
- Robust exception handling and error recovery mechanisms are implemented.

5. **Code quality**: The application must be developed using clean, well-organized code that is easy to understand and maintain.
   Acceptance criteria:
   - Static analysis executed with each commit.
   - Code follows established coding conventions and style guidelines.
   - Code complexity is regularly reviewed and minimized where possible.
6. **Maintainability**: The application should have a continuous integration and deployment process in place to ensure smooth and efficient development, testing, and deployment cycles.
   Acceptance criteria:
   - Deployment pipelines are set up to automatically deploy the application to staging or production environments.
   - Every commit is built and tested automatically.
   - Environment configurations and dependencies are version controlled and easily reproducible.
7. **Operability**: The application should be designed and developed with a focus on operability, ensuring ease of management, monitoring, and troubleshooting in production environments.
   Acceptance criteria:
   - The application provides robust logging and monitoring capabilities, allowing administrators to track and analyze system behavior and performance.
   - Proper error handling and informative error messages are implemented to facilitate troubleshooting and debugging processes.
   - Regular health checks are performed on the application to ensure its operational status.
   - System metrics are collected and analyzed to detect performance bottlenecks or resource constraints.

2.3 Initial Application Architecture

This chapter provides an overview of the architecture of the existing system FlatFeeStack, using the C4 Model Notation[1]. This architecture was present when the project started. The goal is to provide a starting point for the project, which can be used to identify the changes made to the architecture during the project.

2.3.1 System Context Diagram

Figure 2.11 shows a system context diagram, where the system is displayed in its environment and its interactions with external systems and users. The diagram includes the system boundary, actors, and external systems and shows how they interact with the system.

[1] https://c4model.com/

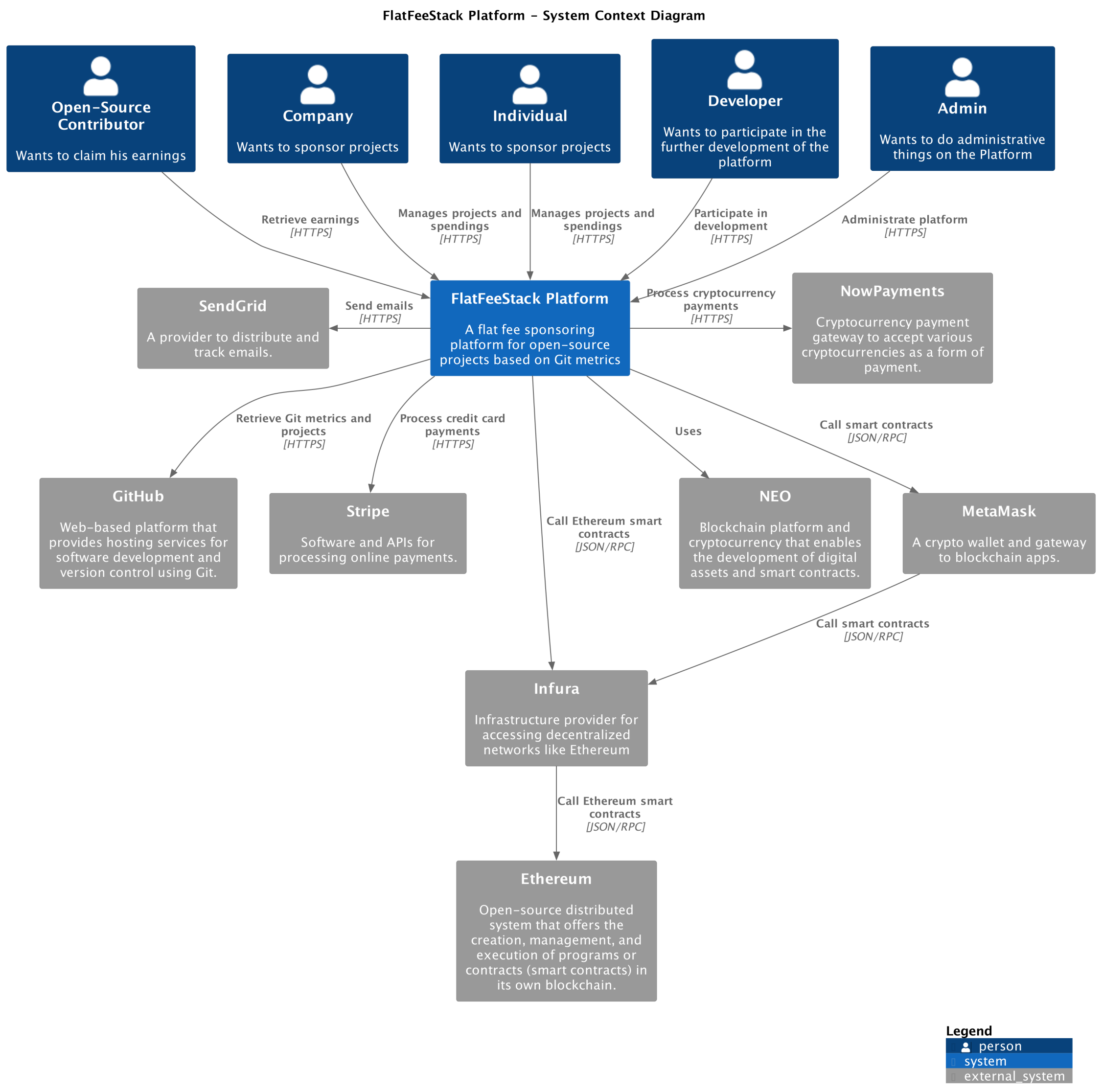

FlatFeeStack Platform - System Context Diagram
Open-Source Contributor
Wants to claim his earnings
Company
Wants to sponsor projects
Individual
Wants to sponsor projects
Developer
Wants to participate in the further development of the platform
Admin
Wants to do administrative things on the Platform
Retrieve earnings [HTTPS]
Manages projects and spendings [HTTPS]
Manages projects and spendings [HTTPS]
Participate in development [HTTPS]
Administrate platform [HTTPS]
SendGrid
A provider to distribute and track emails.
Send emails [HTTPS]
FlatFeeStack Platform
A flat fee sponsoring platform for open-source projects based on Git metrics
Process cryptocurrency payments [HTTPS]
NowPayments
Cryptocurrency payment gateway to accept various cryptocurrencies as a form of payment.
Retrieve Git metrics and projects [HTTPS]
Process credit card payments [HTTPS]
Uses
Call smart contracts [JSON/RPC]
GitHub
Web-based platform that provides hosting services for software development and version control using Git.
Stripe
Software and APIs for processing online payments.
Call Ethereum smart contracts [JSON/RPC]
NEO
Blockchain platform and cryptocurrency that enables the development of digital assets and smart contracts.
MetaMask
A crypto wallet and gateway to blockchain apps.
Call smart contracts [JSON/RPC]
Infura
Infrastructure provider for accessing decentralized networks like Ethereum
Call Ethereum smart contracts [JSON/RPC]
Ethereum
Open-source distributed system that offers the creation, management, and execution of programs or contracts (smart contracts) in its own blockchain.
Legend
person
system
external_system

Figure 2.11: System context.

2.3.2 Container

Figure 2.12 lists a container diagram using the C4 Model Notation, which shows the high-level components of the system and their relationships.

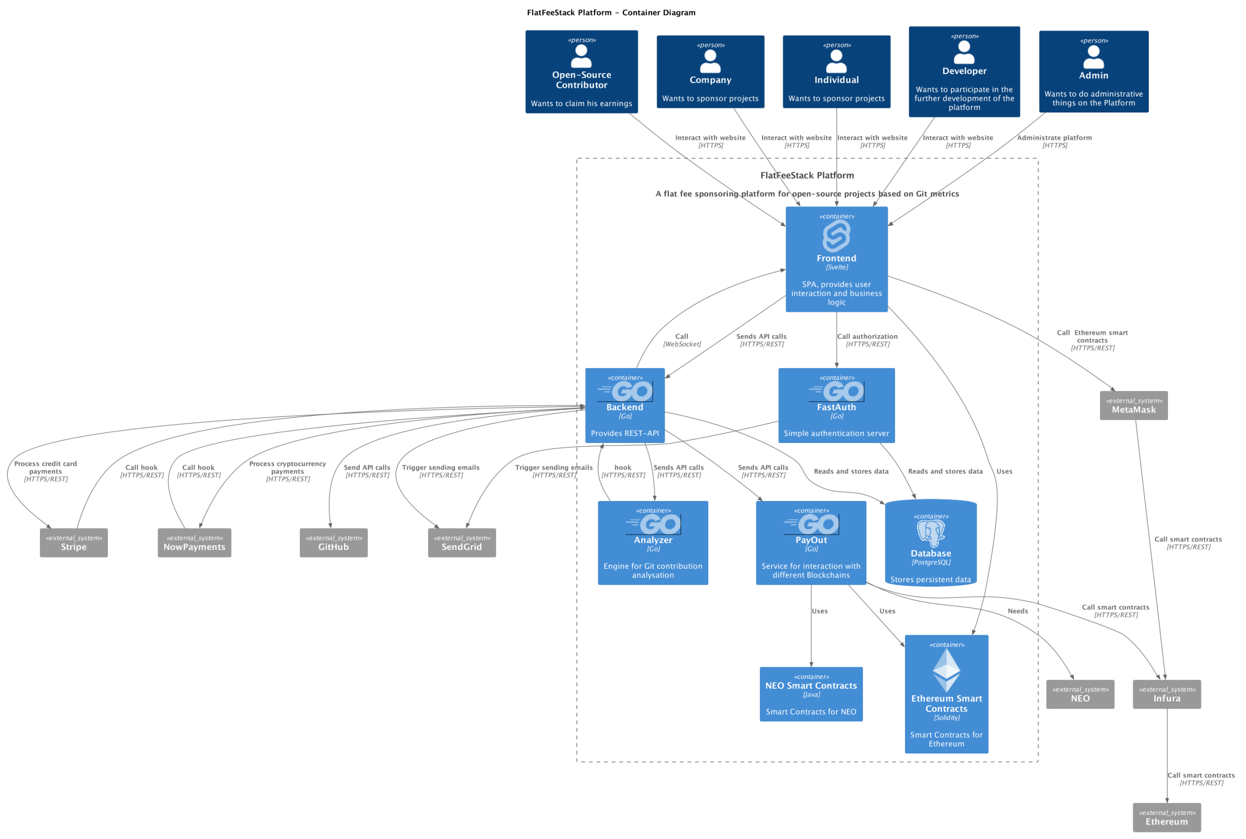

FlatFeeStack Platform – Container Diagram
«person» Open–Source Contributor
Wants to claim his earnings
«person» Company
Wants to sponsor projects
«person» Individual
Wants to sponsor projects
«person» Developer
Wants to participate in the further development of the platform
«person» Admin
Wants to do administrative things on the Platform
Interact with website [HTTPS]
Interact with website [HTTPS]
Interact with website [HTTPS]
Interact with website [HTTPS]
Administrate platform [HTTPS]
FlatFeeStack Platform
A flat fee sponsoring platform for open–source projects based on Git metrics
«container» Frontend [Svelte]
SPA, provides user interaction and business logic
Call [WebSocket]
Sends API calls [HTTPS/REST]
Call authorization [HTTPS/REST]
Call Ethereum smart contracts [HTTPS/REST]
«container» Backend [Go]
Provides REST–API
«container» FastAuth [Go]
Simple authentication server
«external_system» MetaMask
Process credit card payments [HTTPS/REST]
Call hook [HTTPS/REST]
Call hook [HTTPS/REST]
Process cryptocurrency payments [HTTPS/REST]
Send API calls [HTTPS/REST]
Trigger sending emails [HTTPS/REST]
Trigger sending emails [HTTPS/REST]
hook [HTTPS/REST]
Sends API calls [HTTPS/REST]
Sends API calls [HTTPS/REST]
Reads and stores data
Reads and stores data
Uses
«external_system» Stripe
«external_system» NowPayments
«external_system» GitHub
«external_system» SendGrid
«container» Analyzer [Go]
Engine for Git contribution analysation
«container» PayOut [Go]
Service for interaction with different Blockchains
«container» Database [PostgreSQL]
Stores persistent data
Call smart contracts [HTTPS/REST]
Uses
Uses
Needs
Call smart contracts [HTTPS/REST]
«container» NEO Smart Contracts [Java]
Smart Contracts for NEO
«container» Ethereum Smart Contracts [Solidity]
Smart Contracts for Ethereum
«external_system» NEO
«external_system» Infura
Call smart contracts [HTTPS/REST]
«external_system» Ethereum

Figure 2.12: Container diagram.

2.3.3 Component Diagrams

The chapter provides detailed component diagrams for each of the system's components. For clarity, the diagrams are split into parts. Each part shows the component's internal structure and the relationships between internal and external components. The diagram showing all the components is in the appendix (D.1).

Frontend

The component diagram for the frontend can be seen in Figure 2.13. In order to interact with the ETH blockchain, the frontend requires the MetaMask[1] browser plugin. The frontend is built as a single page application (SPA) using the Svelte[2] framework. The routes component defines each page, which utilizes Svelte stores and other reusable Svelte components. REST-API calls are abstracted in the services component.

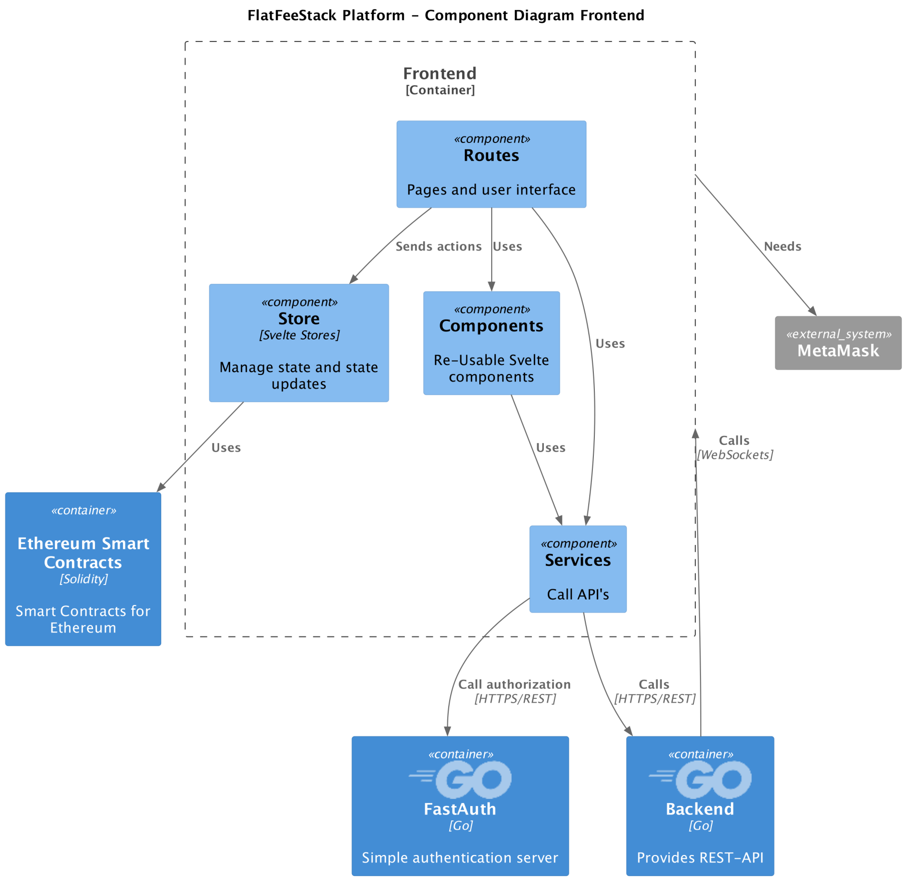


Figure 2.13: Component diagram frontend.

[1]https://metamask.io/
[2]https://svelte.dev/

Backend

Figure 2.14 displays the backend component diagram, which is the largest microservice in the system. It offers a REST-API for the frontend and employs WebSocket communication to handle credit card payments. The main component constructs the server, manages REST-API routes, and initiates Cron jobs. The Cron jobs distribute money to users and initiate new Git repository analysis. The `PaymentNow` and `PaymentStripe` components communicate with external payment providers NOWPayments[1] and Stripe[2]. The database component abstracts the database connection and provides SQL query execution functions. The email component communicates with the external SMTP server SendGrid[3]. The API component abstracts communication with the payout (2.3.3) and analyzer (2.3.3) components. It also abstracts calls to the GitHub API when users search for repositories to sponsor.

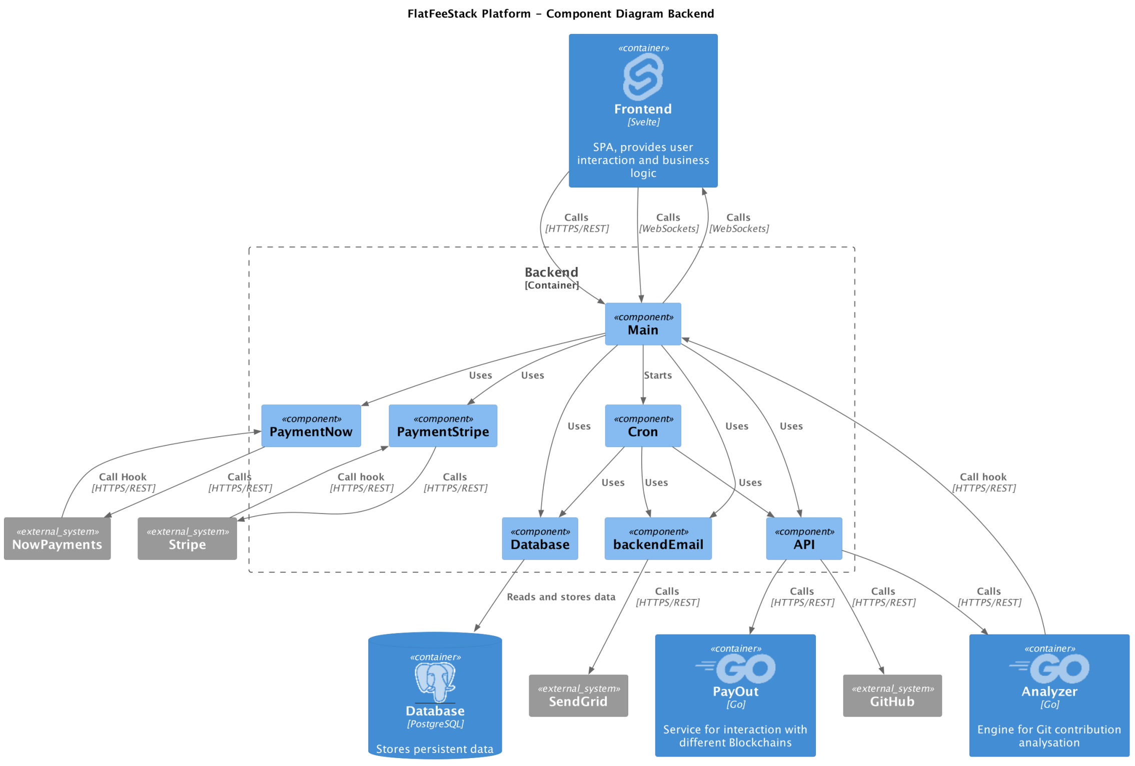


Figure 2.14: Component diagram backend.

Analyzer

Figure 2.15 displays the diagram for the analyzer component, which examines Git repositories. It evaluates the commits in the repository and assigns a weight to each user based

[1] https://nowpayments.io/
[2] https://stripe.com/
[3] https://sendgrid.com/

on their contributions. The more a user contributes, the higher their weight. The backend components assign tasks to the analyzer, and upon completion, it sends the results back to the backend.

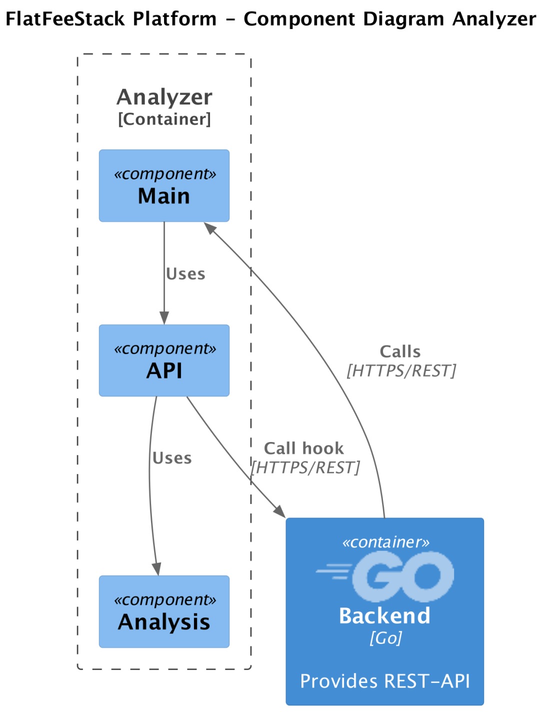


Figure 2.15: Component diagram analyzer.

FastAuth

The fastauth component, displayed in Figure 2.16, is in charge of authenticating users. To achieve this, it offers a REST-API to the frontend. During the sign-up process, it utilizes the email component to send a verification email to the user. The main component routes the calls to the API component, which implements the business logic. Additionally, the database components abstract the connection and provide functions to execute SQL queries.

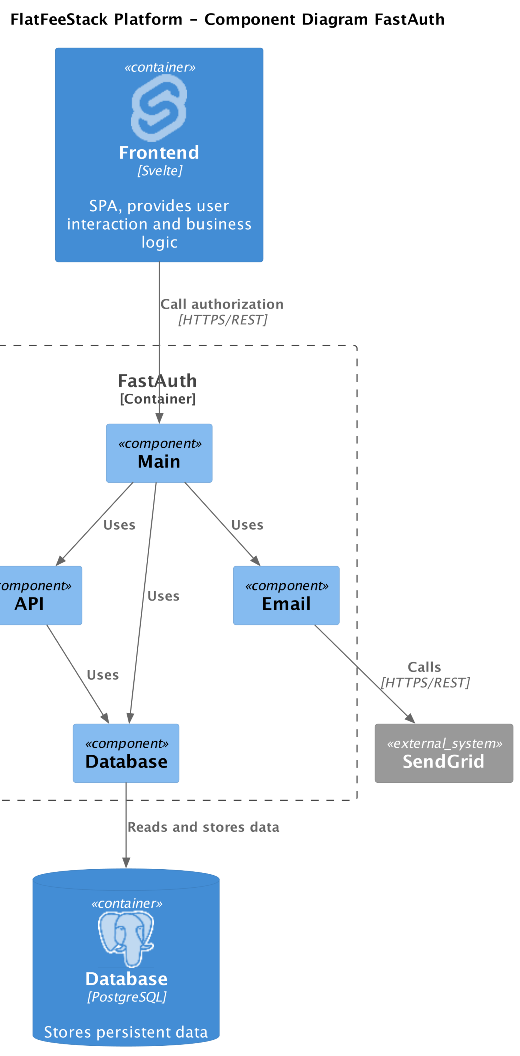


Figure 2.16: Component diagram fastauth.

Payout

The payout component, seen in Figure 2.17, distributes money to the users. To initiate a payout, the backend component (2.3.3) contacts the payout component through a REST-API. Depending on whether the payout is in ETH or NEO, the component calculates a signature that allows users to withdraw their money by calling the appropriate smart contracts.

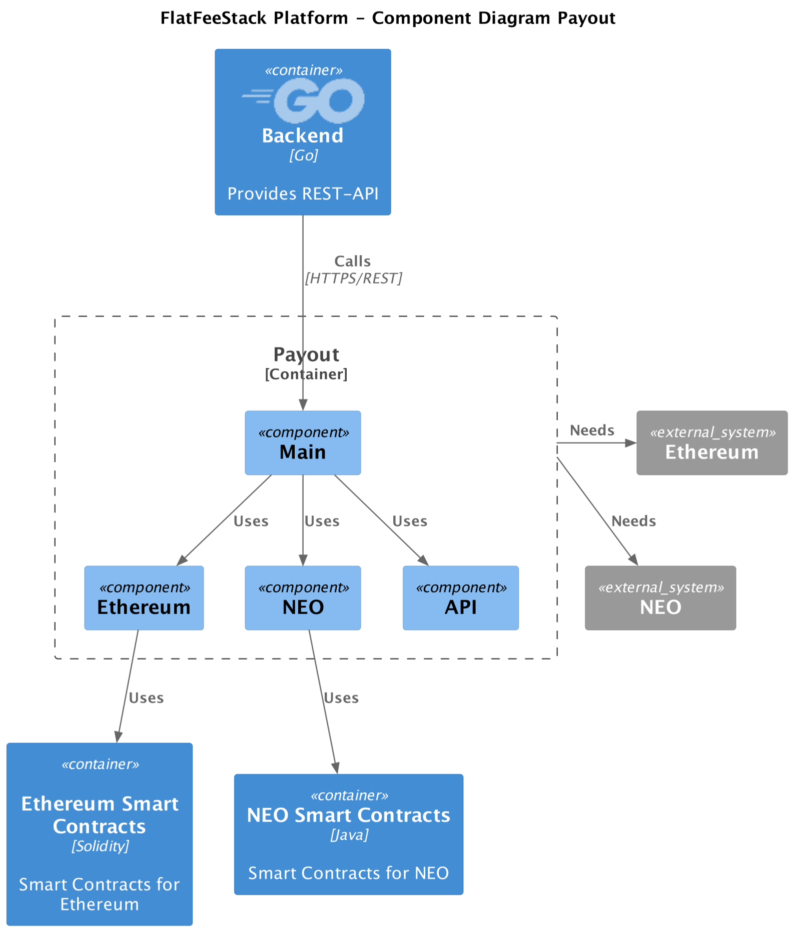


Figure 2.17: Component diagram payout.

Smart Contracts

The smart contracts portion in Figure 2.18, includes the smart contracts deployed on the ETH and NEO blockchains. More information about the DAO smart contracts is provided in [1]. The payout contracts are utilized by the payout component, which is discussed further in section 2.3.3.

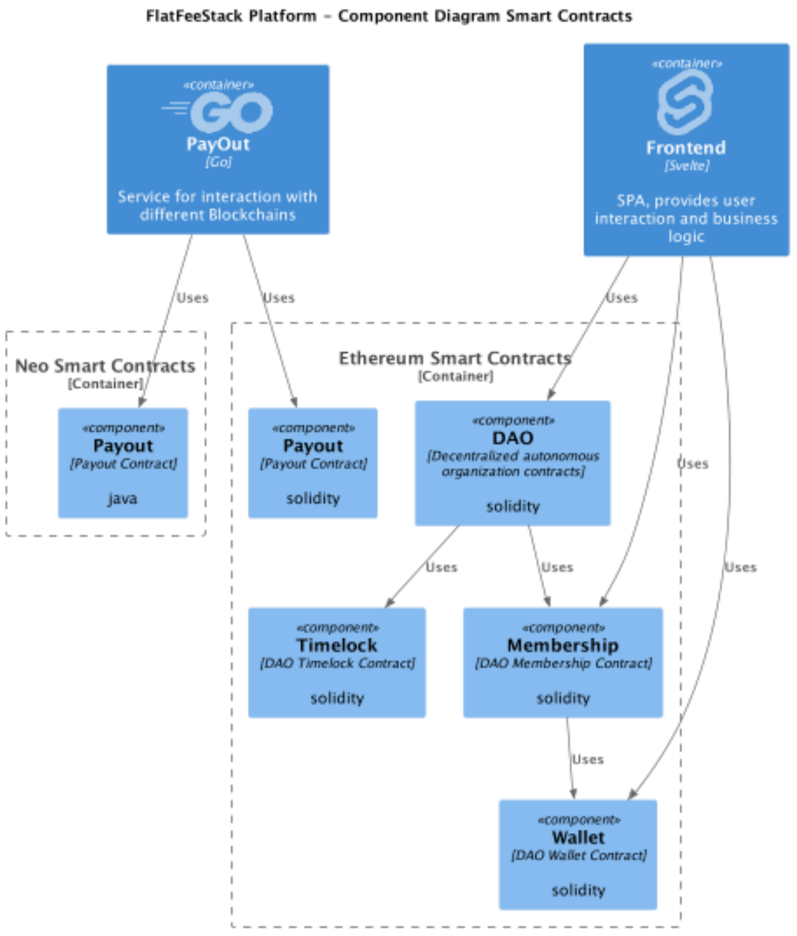


Figure 2.18: Component diagram smart contracts.

2.3.4 Potential for Improvement

For the creation of these diagrams, the existing code was deeply analyzed. The architecture itself is clean and well-structured. The code is also well-structured inside the components, but there is room for improvement.

Database

The backend and the fastauth component use the database. Both components have code for connecting to the database and code for executing SQL from an external file. This code should be refactored into a separate package that both components can use.

Also, in a microservice architecture, each service should have its database, which differs from the existing system However, a good thing is that the services only access certain database tables. So the fastauth component only accesses the table auth, which is also created by it. Accordingly, the backend also only accesses the which it created.

JWT

Each go microservice has code to validate JWT tokens. This code should be refactored into a separate package that all microservices can use.

Email

The backend and the fastauth component email users using SendGrid[1] as an SMTP server. Both components have code for sending emails and reading templates for these emails. Creating a separate package that both components can use will improve the code.

[1]`https://sendgrid.com/`

Chapter 3

Research

For various tasks that should be completed until the Go-Live, research is necessary to get an overview of existing functionalities and products.

Section 3.1 describes a set of evaluation criteria for the PaaS provider where FlatFeeStack should be hosted. Afterward, three providers are compared, and a platform is chosen for implementation.

Section 3.2 summarises the existing work on the payout functionality of the platform of different theses. Questions about the design of the solution are examined, and an implementation proposal is presented.

Section 3.3 evaluates a platform for discussions for the DAO. The previous work on the FlatFeeStack DAO concluded that a platform was needed to discuss topics about the DAO [1]. A list of criteria is conducted before different solutions are compared against it, choosing a solution for the discussions.

3.1 Evaluation PaaS Provider

3.1.1 Why a PaaS Provider?

During the first meeting, the project team and advisor discussed how to host the production environment for FlatFeeStack while discussing the tasks for the go-live. The current staging environment runs on a dedicated server in the office of Axelra[1] with containers. One idea was to host the production environment on an additional server in the Axelra office.

[1]https://www.axelra.com/

Below is the outcome of the discussion on why not to self-host the production environment:

- One static IP address is assigned to the Axelra office and must be shared between the staging and production environments. This necessitates routing traffic through a combined reverse proxy, which creates an unhealthy dependency between the two domains.
- Self-hosting means somebody needs to invest time to install software updates and to ensure the service runs.

Therefore, as part of this thesis, a suitable, cloud-based PaaS provider should be evaluated where the FlatFeeStack platform can be deployed.

3.1.2 Evaluation Criteria

1. **Container support** (20% weight): The provider should provide container deployment and management support, and they should reuse the existing Dockerfiles and built containers.
2. **Ease of deployment** (15% weight): The provider should offer an easy way to deploy and manage containers without requiring you to write more extensive configuration files (like the YAML definition for a Kubernetes[1] deployment). Configuring environment variables and other settings should be simple and intuitive.
3. **Scalability** (10% weight): The provider should offer easy scaling of your application horizontally and vertically. Autoscaling based on metrics such as CPU and memory usage should be available.
4. **Metrics and monitoring** (10% weight): The provider should offer a dashboard to review metrics such as CPU and memory usage, with the ability to set automated alerts.
5. **Database as a service** (10% weight): The provider should offer a managed database service, ideally PostgreSQL[2], with easy provisioning, scaling, and backups.
6. **Self-service portal** (10% weight): The provider should offer a self-service portal to manage services, including provisioning and scaling of resources and managing user access and permissions.
7. **Pricing** (10% weight): The provider should offer a fair pricing model for their services.

[1] https://kubernetes.io/
[2] https://www.postgresql.org/

For each criterion, a number between 0 (not fulfilled) and 3 (expectations exceeded) will be assigned based on the offerings of a cloud provider.

The points for the price will be examined based on costs per month 24 hours uptime. The highest price will receive 0 points, while the cheapest will get 3. Points for prices in-between will be calculated as follows:

$$\frac{\text{Highest Price} - \text{Price}}{\text{Highest Price} - \text{Lowest Price}} \cdot 3 \tag{3.1}$$

3.1.3 Pre-Selection of Providers

Based on internet research and discussions with the advisor, a list of providers has been conducted for the evaluation:

- DigitalOcean App Platform[1]
- Google App Engine[2]
- Flow Swiss App Engine[3]

3.1.4 Evaluation

DigitalOcean App Platform

DigitalOcean App Platform allows publishing applications using various languages, but also Docker. Two points for Container support.

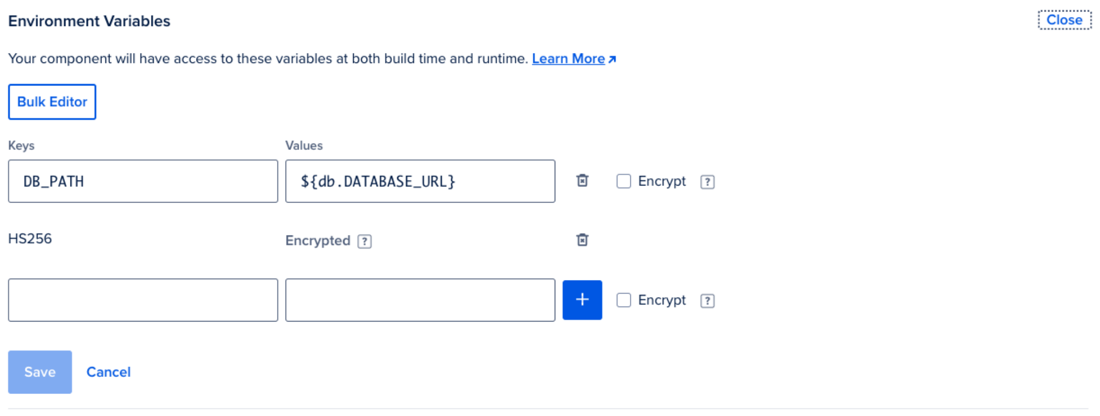
For testing, only the backend was deployed to DigitalOcean. The setup wizard connects to existing repositories on GitHub and GitLab. Although DigitalOcean claims to support mono repositories with their approach [2], adding the FlatFeeStack mono repository and pointing the source folder for the code to the backend did not work. The build process claimed not to find a Dockerfile. Selecting the version of the backend before the mono repository merge worked fine. If DigitalOcean were chosen, a copy of the Docker container would have to be pushed to their registry.

DigitalOcean can watch changes on a branch and trigger a redeployment. This functionality can help implement continuous deployment for the project later.

Environment variables can be configured using the web interface. A great benefit is that variables can be encrypted and are not readable once saved, as shown in Figure 3.1. This

[1]https://www.digitalocean.com/products/app-platform
[2]https://cloud.google.com/appengine
[3]https://flow.swiss/app-engine

feature is good for the HS256 secret encrypting JWT headers in the backend. Overall 3 points for ease of deployment.


Environment Variables

Your component will have access to these variables at both build time and runtime. Learn More

Keys	Values	
DB_PATH	${db.DATABASE_URL}	Encrypt
HS256	Encrypted	
		Encrypt

Figure 3.1: Management of environment variables on DigitalOcean.

DigitalOcean allows scaling horizontally and vertically. Existing instances of a service can be allocated more resources. If another instance is needed, a service can be copied. However, scaling is not automated. Two points for scalability.

DigitalOcean offers a simple dashboard for resource consumption, mainly CPU and memory usage, as shown in Figure 3.2. Additionally, one can define mail notifications if certain thresholds are passed — 3 points for metrics and monitoring.

DigitalOcean composes multiple services as one app. For apps, one or multiple PostgreSQL databases can be attached. Afterwards, a variable is available for each service in the app with the database URL to make a connection. This comes in convenient for FlatFeeStack, where it is meant that two of the services (backend and fastauth) connect to the same database. The cheapest database comes without automated failover, which is something to consider when evaluating the price. Two points for the database as a service.

The self-service portal is designed in an easy way to find all essential functions quickly. All services can be inspected as well as edited. Created apps usually belong to a team and not an individual account. This would prevent access issues compared to only one person owning an application—three points for the self-service portal.

Given the pricing information, hosting FlatFeeStack on DigitalOcean would create the following monthly cost:

- $15 for a database with failover, one vCPU, 1GB of RAM, and 10 GB of storage.
- $25 for hosting all four services (backend, frontend, payout, analyzer, fastauth), with the smallest instance size available (1 vCPU, 512 MB RAM).

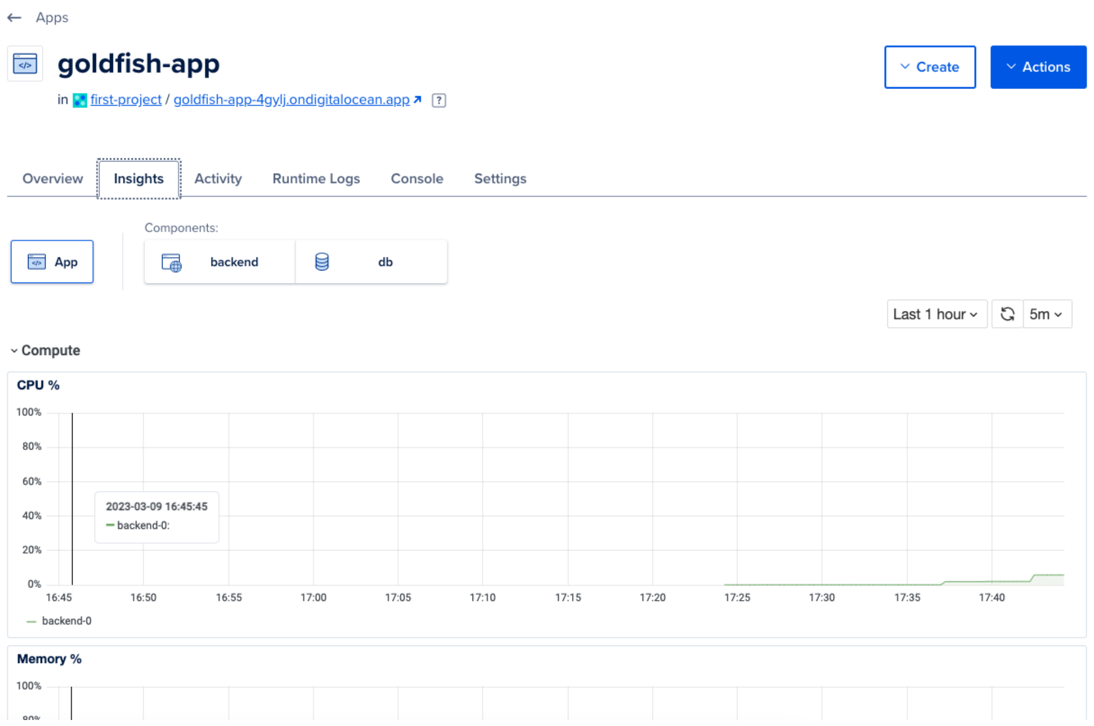


Figure 3.2: Metrics for a service on DigitalOcean.

- $5 for hosting the container images.

Given a currency conversion rate from the US dollar to Swiss francs of 0.95, the monthly price for DigitalOcean would be 42.75 CHF. This is the lowest price of all three providers, giving three points for the pricing.

Google Cloud App Engine

Google App Engine allows the deployment of applications using Dockerfiles—two points for Container support.

Google Cloud divides cloud resources by projects. One project in Google Cloud then has many resources. In the case of App Engine, a user can deploy multiple applications to the same project. Each application can be assigned traffic rules. New deployments create new versions, where mechanisms like A/B testing or Canary deployment can be applied using the dashboard.

To test the App Engine, the backend of FlatFeeStack was deployed. Two files must be defined in the source repository: an `app.yaml`, where the runtime environment is specified, and an additional file with environment variables, which should not be committed to the Git repository. For production deployment, an encrypted version of this file could be added to the FlatFeeStack repository, and the secret shared over another channel with all the developers.

A developer can deploy the application using the `gcloud` CLI with the mentioned file. There are also ways to connect the application with the GitHub repository, so automated deployments can be triggered when pushing to Git.

Overall, two points for ease of deployment, as the amount of required configuration to deploy an application is low.

Google offers configuration options for auto-scaling [3]. They continuously measure resources used by the instances, and based on set criteria, more services are created, or existing ones are destroyed when no longer needed—two points for scalability.

App Engine offers the most extensive metrics dashboard of all the evaluated tools. A selection of the available metrics is shown in Figure 3.3. Additionally, the metrics dashboard looks good—three points for metrics and monitoring.

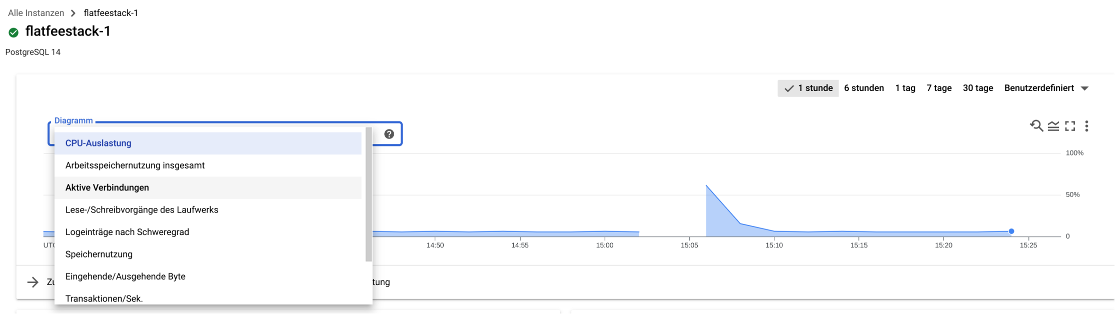


Figure 3.3: Metrics for the database on Google Cloud.

Databases for App Engine are in a separate service called Cloud SQL. Google offers PostgreSQL as a service, including many configuration options for the number of resources for the server, high availability, and target deployment zones. Afterwards, there are various ways to connect from an App Engine instance to the database: The recommended way is to create a service account for it and give it permissions. Then a UNIX socket will be mounted into the container [4].

Security-wise, this is a suitable mechanism, as the authentication happens outside the main container. However, getting all the configuration up and running is more effort than other services—two points for the database as a service.

The user interface is generally easy to navigate. The main pain point is to know how to find matching services or settings on this platform. Google makes this up by providing numerous tutorials to do various things, but for example, the dashboard never showed that the initial deployment failed for one of the test instances. It was only visible from the local developer console. Also, connecting a database from App Engine are two separate tutorials. Google Cloud is a vast product, and this shines through in the user interface—one point for the self-service portal.

The application's hosting prices have been calculated with the Google Cloud Pricing Calculator, with the price list from March 15th, 2023, and using us-central as the base zone.

- Running five App Engine instances costs $322.42. There isn't any specification of how many resources are guaranteed for the services.
- Running one PostgreSQL with 4 GB of RAM and 1 CPU, including high availability, is $102.82 a month.

Given a currency conversion rate from the US dollar to Swiss francs of 0.95, the monthly price for Google Cloud would be 403.98 CHF. This is the highest price of all three providers, giving zero points for the pricing.

Flow Swiss App Platform

Flow's platform is based on a software named Virtuozzo. Therefore, references to documentation in this section often point to their website.

Flow supports deploying from a container registry, two points for Container support.

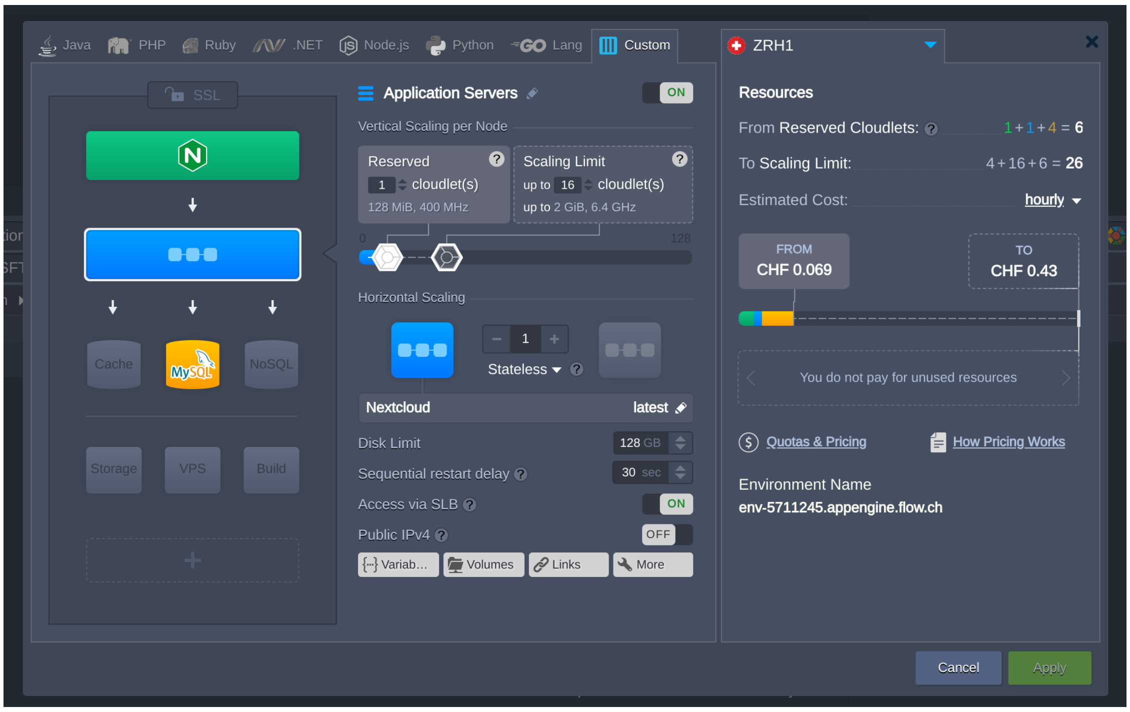
For testing, an instance of Nextcloud[1] was deployed. Flow can usually connect to a container registry directly. However, something did not work with the authentication against the GitHub container registry, where the Docker images for FlatFeeStack are saved. Nextcloud started as a platform to store files but is extensible with add-ons like video chat. The application requires a permanent data store and a database, and some configuration using environment variables, similar to the use case of a service in FlatFeeStack.

Flow organizes deployment in the so-called environments. An interface allows users to select a load balancer, one or more applications, and database services. The interface exposes multiple customization options and is kept close to the underlying technology. For instance, a menu point shown in Figure 3.4 named variables, which takes one to configure the environment variables, which looks similar to a dotenv file that can be passed to a Docker container.

Once an environment is configured, a developer can trigger a redeployment from the user interface or via CLI [5].

Overall two points for ease of deployment.

[1] https://nextcloud.com/



Application Servers ON

Vertical Scaling per Node

Reserved	Scaling Limit
1 cloudlet(s)	up to 16 cloudlet(s)
128 MiB, 400 MHz	up to 2 GiB, 6.4 GHz

Horizontal Scaling: 1, Stateless

Nextcloud latest

Disk Limit 128 GB

Sequential restart delay 30 sec

Access via SLB ON

Public IPv4 OFF

Variab... Volumes Links More

Resources

From Reserved Cloudlets: 1+1+4 = 6

To Scaling Limit: 4+16+6 = 26

Estimated Cost: hourly

FROM CHF 0.069 TO CHF 0.43

You do not pay for unused resources

Quotas & Pricing How Pricing Works

Environment Name

env-5711245.appengine.flow.ch

Figure 3.4: Interface to configure a new environment on Flow.

Flow can scale the application automatically. When creating application services, a user has to select a range of how many so-called cloudlets should run for an application, as seen in Figure 3.4. This mechanism works for load balancers and databases as well. Triggers can be defined to define criteria when scaling should happen, but automated scaling is not possible based on current resource consumption [6] — two points for scalability.

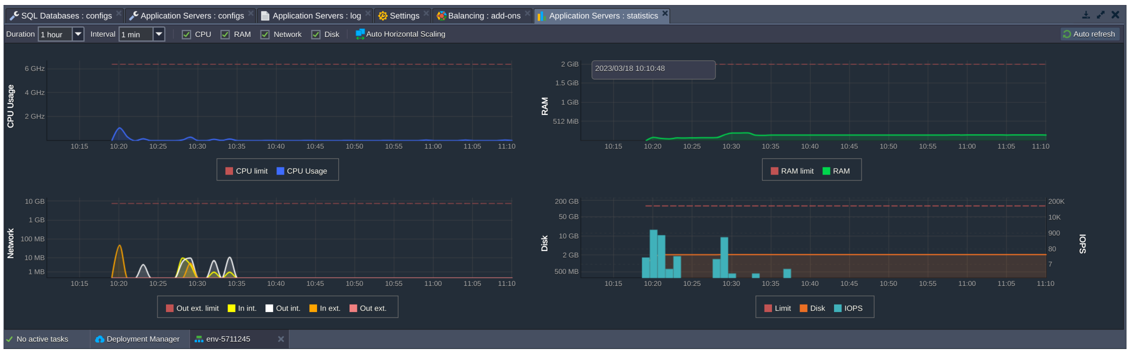
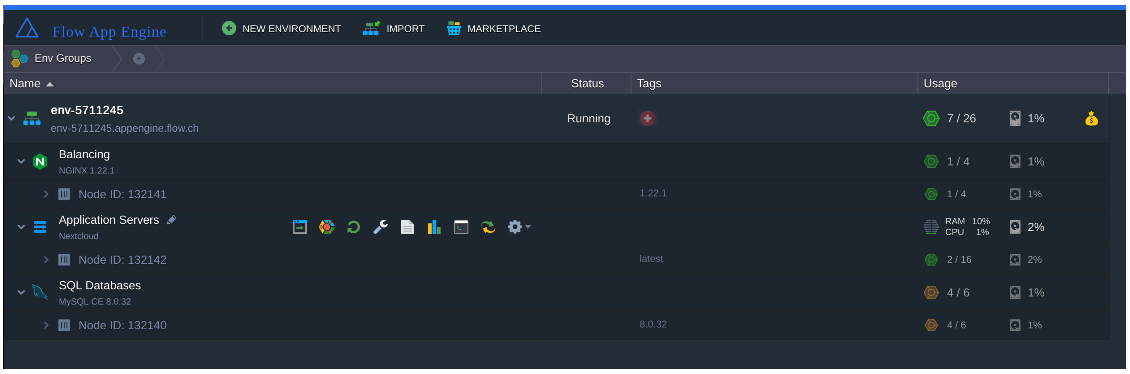


Figure 3.5: Resource consumption interface on flow.swiss.

Flow offers good insights into current resource consumption, as shown in Figure 3.5. Two points for metrics and monitoring.

As mentioned in ease of deployment, services are composed together as environments. The database is part of this environment and is configured and deployed in the same way as an application service. To enable the database to be available to a system, the containers must be linked, similar to Docker networks — 3 points for the database as a service.



Name	Status	Tags	Usage	
env-5711245 env-5711245.appengine.flow.ch	Running		7 / 26	1%
Balancing NGINX 1.22.1			1 / 4	1%
Node ID: 132141		1.22.1	1 / 4	1%
Application Servers Nextcloud			RAM 10% CPU 1%	2%
Node ID: 132142		latest	2 / 16	2%
SQL Databases MySQL CE 8.0.32			4 / 6	1%
Node ID: 132140		8.0.32	4 / 6	1%

Figure 3.6: Environment overview in Flow.

As mentioned in ease of deployment, the user interface exposes many configuration options to manage an environment. Nevertheless, at the same time, it is often cluttered, and menu points could be more intuitive. Many menus are hidden behind symbols that need to be clarified. For instance, to add SSL to one's load balancer, an add-on menu is available that is abstracted using a hexagon symbol, as seen in Figure 3.6. One point for the self-service portal.

Pricing for Flow could look as follows:

- Four cloudlets per service (512 MB and 1.6 GHz) are 0.039 CHF per hour. FlatFeeStack has five services (backend, frontend, payout, analyzer, fastauth), and as monthly costs should be compared, this gives a total of $0.039 \text{ CHF per hour} \cdot 24 \text{ hours a day} \cdot 30 \text{ days a month} \cdot 5 \text{ services} = 140.40 \text{ CHF}$.
- A database with three cloudlets (384 MB and 1.2 GHz) is 0.029 CHF per hour. Monthly costs, therefore, are $0.029 \text{ CHF per hour} \cdot 24 \text{ hours a day} \cdot 30 \text{ days a month} = 20.88 \text{ CHF}$.

This calculation results in a total cost of 161.28 CHF per month. Using the formula to evaluate the points, Flow receives 2.02 points for the pricing.

Summary

Table 3.1 summarizes the evaluation of the different hosting providers.

Criteria	Weight	DigitalOcean		Google Cloud		flow.swiss	
Container support	20%	2	0.4	2	0.4	2	0.4
Ease of deployment	15%	3	0.45	2	0.3	2	0.3
Scalability	10%	2	0.2	2	0.2	2	0.2
Metrics and monitoring	10%	3	0.3	3	0.3	2	0.2
Database as a service	10%	2	0.2	2	0.2	3	0.3
Self-service portal	10%	3	0.3	1	0.1	1	0.1
Pricing	10%	3	0.3	0	0	2.02	0.202
		2.15		**1.5**		**1.802**	

Table 3.1: Comparison of the hosting providers.

Therefore, the FlatFeeStack production environment will be hosted on DigitalOcean, as they scored the most points in the evaluation.

3.2 Complete Payout Functionality

While testing the existing functionalities, it was discovered that the payout functionality no longer works. Before this thesis, two other theses have been written about payouts within FlatFeeStack. This chapter will summarise the state of this functionality prior to this thesis. Additionally, section 3.2.1 describes the changes between the two theses and the start of this Bachelor's thesis. Afterward, the goals of this task are discussed, including ideas for implementation.

3.2.1 Prior Work

Kryptowährungen als Zahlungsmittel bei FlatFeeStack, Lesi and Endres, SA HS21

Lesi and Endres added the pay-in functionality using cryptocurrencies in FlatFeeStack. Because the pay-in and payout processes in FlatFeeStack are tightly coupled, their thesis also changed how the payout works.

Prior to their thesis, pay-in was only possible using a credit card with Stripe. This process was extended by integrating FlatFeeStack with NOWPayments[1], which allows receiving payments in cryptocurrencies. FlatFeeStack runs a daily calculation to distribute the pay-ins of a user to a repository. The daily deduction was previously a fixed amount of 33 cents of the user's current credit on the platform. They changed the system to divide the payed-in amount by the number of days for which the pay-in is valid (either 90 or 365 days) and also to allow the distribution of the newly integrated cryptocurrencies. This system also allows one to accept new currencies or add other durations for pay-in periods without changing much code.

Before their thesis, the payout service sent a daily status update to a smart contract on the ETH blockchain about how many contributions a user has received, using their wallet address as uniquely identifying information. This mechanism has been changed to a batch payout mechanism: Each month, the payout service submits the contributions for a user directly to their wallet, bypassing the need for a smart contract on the respective blockchain. Pay-ins in US dollars get converted into ETH by calling the CoinGecko[2] API.

Additionally, as their thesis focused on allowing pay-ins and payouts using the Tezos cryptocurrency, they implemented a new payout microservice in NodeJS, as no library was available in Go to interact with Tezos [7].

[1] https://nowpayments.io/
[2] https://www.coingecko.com/

Design and Implementation of a Fee Optimization Mechanism in Blockchain-based Payments for an Open Source Donation Platform, Michael Bucher, HS21

The content of the thesis from Michael Bucher was to optimize the transaction costs for sending the contributions from FlatFeeStack to the respective recipients. This thesis was written parallel to the one of Lesi and Endres, described in section 3.2.1.

Michael Bucher references a work by Jonas Brunner [8], where they initially decided to use blockchains for a transparent payout process. However, the fee to place a transaction on the blockchain can be substantial. As mentioned in 3.2.1, as the FlatFeeStack platform currently handles the payout, it also covers the transaction fees on the respective blockchain. However, the question remains if the transaction fee should be deducted from the total payout or if FlatFeeStack covers the fees themselves.

The thesis explores implementing a signature mechanism, where the payout service signs a message containing the payout amount for a user who received contributions. The private key with which the message is signed is identical to the owner of the smart contract on the respective blockchain.

The signature is returned to the user, who is instructed to send it to the smart contract on the respective blockchain. Within the smart contract, the message is recreated from the provided parameters. The signer of the message and the signature parameter is then recovered and compared to the smart contract owner. In case the recovered signer and the smart contract owner match, the parameters are valid.

Since the developer needs to request the withdrawal at the smart contract instead of FlatFeeStack sending the earned amount to them, the developers pay the transaction fees, circumventing the question of how FlatFeeStack should pay the transaction fees.

Another important detail is that the message contains the developer's total earned amount, sometimes referenced as TEA. The TEA has the significant benefit of the payout service not communicating with the smart contracts on the chain to tell how much the developer is eligible to withdraw. Instead, the smart contracts also track the total earned amount for each developer. Once a new withdrawal is requested and the provided signature is valid, they update the total earned amount and pay the difference between the old and new TEA.

As a result of their thesis, Michael Bucher implemented two smart contracts, one for the NEO blockchain and one for ETH, which implement the signature mechanism described above [9].

Changes in 2022 / Early 2023

Thomas Bocek worked by himself on FlatFeeStack during 2022 and early 2023. The most notable changes to the payout functionality were:

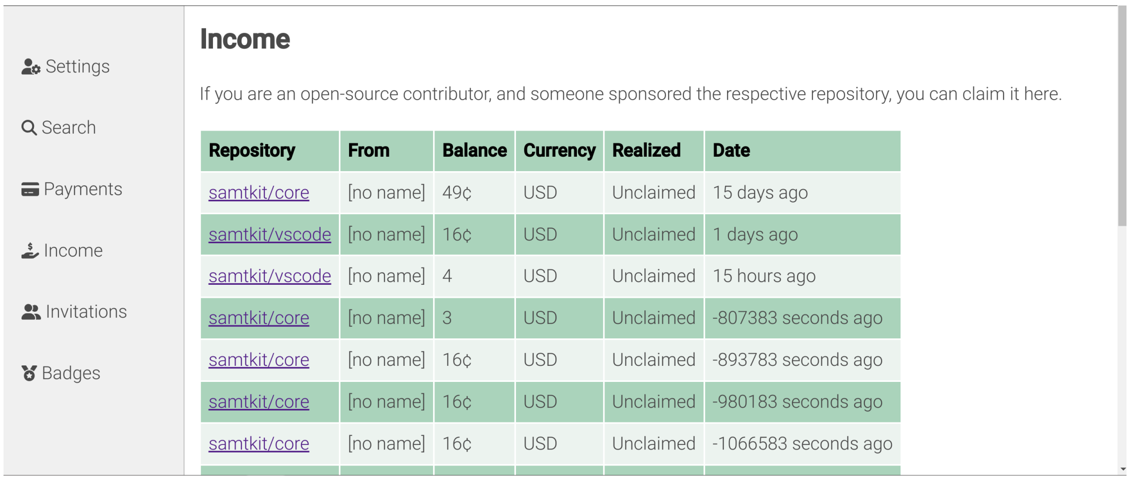
1. He finished the integration of the work done by Lesi and Endres.
2. He changed the mapping in the ETH smart contracts where the TEA is tracked from ETH addresses to user IDs used in the FlatFeeStack backend, eliminating the need in FlatFeeStack to track the wallet addresses of the developer. Michael Bucher implemented the necessary changes in the NEO smart contract in early January 2023 [10].
3. Disabled the integration in the payout service for the NEO blockchain.

3.2.2 Goal / Initial Position

This task aims to wire the previous work together, allowing the developers to withdraw their contributions with the signature mechanism as proposed by Michael Bucher. This results in the following tasks:

1. Add a new API endpoint in the backend, which calculates the total earned amount for a user and calls the payout service to generate a valid signature to retrieve the contributions.
2. Extend the existing Income page in the frontend to display the option to generate a signature for a currency and display the generated signature.
3. For the payout service, re-activate the integration with the NEO blockchain and implement a sign endpoint.

Figure 3.7 shows the existing income page in the frontend.



Income

If you are an open-source contributor, and someone sponsored the respective repository, you can claim it here.

Repository	From	Balance	Currency	Realized	Date
samtkit/core	[no name]	49¢	USD	Unclaimed	15 days ago
samtkit/vscode	[no name]	16¢	USD	Unclaimed	1 days ago
samtkit/vscode	[no name]	4	USD	Unclaimed	15 hours ago
samtkit/core	[no name]	3	USD	Unclaimed	-807383 seconds ago
samtkit/core	[no name]	16¢	USD	Unclaimed	-893783 seconds ago
samtkit/core	[no name]	16¢	USD	Unclaimed	-980183 seconds ago
samtkit/core	[no name]	16¢	USD	Unclaimed	-1066583 seconds ago

Figure 3.7: Existing income page in the frontend.

3.2.3 Design Questions

Tezos Integration

As mentioned in section 3.2.1, Lesi and Endres implemented a new payout service in NodeJS to handle the payout in Tezos. After discussions with the advisor, it was decided to launch the project without Tezos' support.

Payout of US Dollars

As described in section 3.2.1, before the thesis of Endres and Lesi, pay-ins were only available in US dollars, and the contributions were converted to ETH for the payout.

After their thesis, they kept converting US dollars to ETH but combined the payout of both ETH and US dollars. The payout was converted to a manual job, started by an administrator of FlatFeeStack, where they needed to pass the USD conversion rate to ETH. Additionally, users need to withdraw the contributions in their original currency. For example, if a user receives contributions in NEO and Tezos, they must have a Tezos and NEO wallet and make two withdrawal requests.

Each time a payout is triggered, the backend updates the paid-out amount for each user in the respective database table. A user can later see in the frontend how big their next payout will be. This number is calculated by getting the sum of all contributions and subtracting the paid-out amount from the database table mentioned before.

As the work of Lesi and Endres was based upon FlatFeeStack doing the payout, a spike was implemented to see if there was a way to allow users to receive payouts in one currency using the signature mechanism of Michael Bucher. The design included the following steps performed by the backend:

- Convert the contributions received in different currencies to one target currency using conversion rates retrieved from CoinGecko.
- Generate a signature in the target currency.
- Add a database entry to a new table named *payout_requests* with the amount in the target currency.
- Link the individual contributions to the created entry in *payout_requests* to mark them as claimed.

This system would have allowed recalculating the total earned amount needed for the signature based on the information from the *payout_requests* table. However, in discussions with the advisor, a major issue was discovered regarding the volatility of cryptocurrencies.

1. Assume that today, \$1 yields 0.05 ETH. A user generates a signature using this conversion rate.
2. They are not required to use the signature on the blockchain immediately, so they wait for another four weeks.
3. After four weeks, \$1 yields only 0.01 ETH, which means the ETH price increased by five times.
4. Now the user retrieves the payout from the smart contract with the signature generated four weeks ago. As the signature was created four weeks ago and FlatFeeStack likely also transferred the necessary amount four weeks ago, the price increase is at the expense of FlatFeeStack.

Therefore, it was decided with the advisor and external co-examiner to also payout the US dollars in something close to the original fiat currency. This requires an additional payout contract that can interact with the USDC stablecoin on ETH. The stablecoin mimics the dollar price and therefore is not subject to fluctuating exchange rates.

3.3 Evaluation Form of Discussion on Proposals

The previous work on the FlatFeeStack DAO [1] concluded that a thorough discussion regarding the proposal is necessary. This chapter will delve into a detailed analysis of different potential solutions.

3.3.1 Introduction

The FlatFeeStack DAO operates on a decentralized decision-making model, where every member can propose changes to the organization. While this method gives members more control over how the DAO develops, it occasionally results in ambiguous suggestions or calls for additional discussion. In situations like this, it is critical to have a reliable mechanism to evaluate and rank the suggested solutions. This guarantees that the DAO chooses the best course of action for attaining its goals and makes informed judgments. This thesis explores various solutions for discussing proposals within the FlatFeeStack DAO, aiming to provide recommendations for the decision-making process.

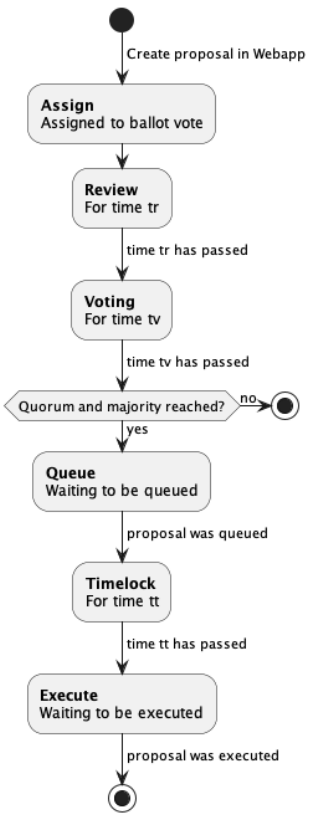


Figure 3.8: Current proposal lifecycle.

The current proposal lifecycle, depicted in Figure 3.8, relies on the FlatFeeStack DAO's current implementation. Under this lifecycle, a DAO member can generate a proposal, which is promptly written to the blockchain and linked to a ballot vote in the smart contract. Notably, there is no provision for discussion within the current proposal lifecycle, which means that DAO members cannot deliberate on the proposal before casting their votes unless they resort to external communication platforms. Additionally, the member who initiates the proposal has already spent time writing it and money for the transaction to create the proposal on the ETH blockchain.

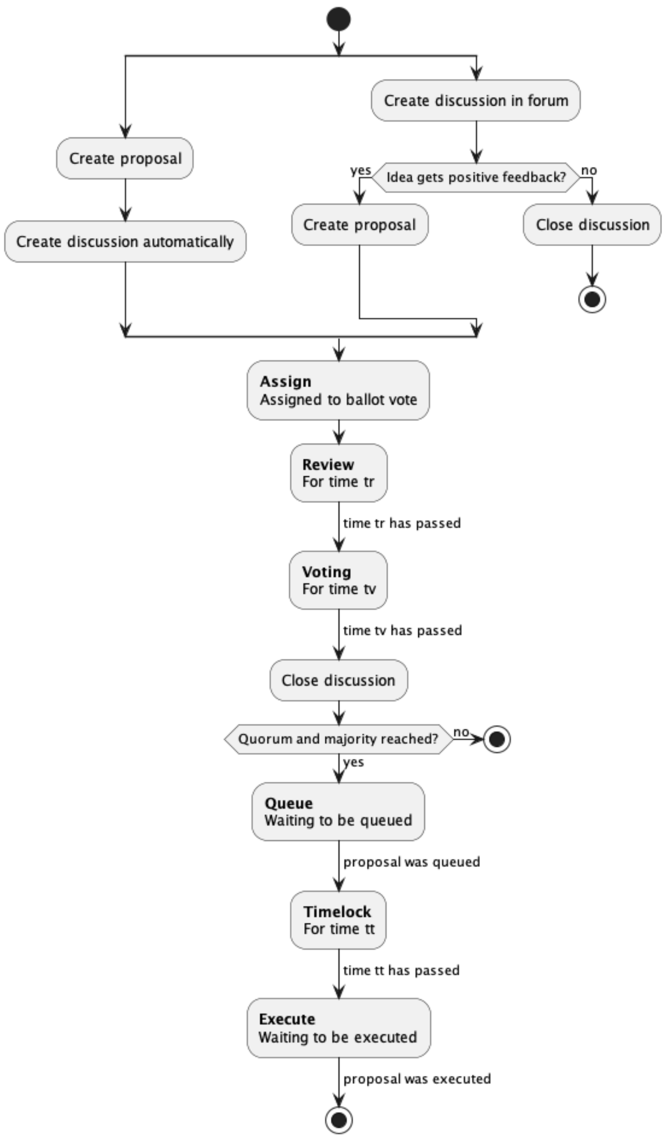


Figure 3.9: Goal proposal lifecycle.

Figure 3.9 shows the goal proposal lifecycle. The main difference is the DAO member can choose to create a proposal directly on the blockchain or to create a proposal off-chain. When the member chooses to create a proposal directly on the chain, the system will automatically start a discussion thread and write the proposal to the blockchain. Therefore, the members also can discuss the proposal before voting.

If the member chooses to create a discussion thread first, they will create it and can subsequently discuss the proposal. The member who initiates the proposal can wait for feedback from the community. If the idea gets enough support, the member can create the proposal on the chain. Alternatively, the member can close the discussion thread, preventing the proposal from being created on the chain. This procedure can save time and money for the member.

The evaluated solution must support the lifecycle, shown in Figure 3.9.

3.3.2 Evaluation Criteria

The solution must meet the following identified criteria.

1. **Cost:** The solution must be cost-effective.
2. **Ease of use:** The solution must be easy to use and intuitive for the DAO members.
3. **Integration:** Integrating the solution within the current FlatFeeStack frontend is necessary.
4. **Restrictions:** Only users with a login to the FlatFeeStack platform should be able to participate in the discussions.

3.3.3 Pre-Selection of Possible Solutions

The evaluation criteria, internet research, and discussions with the external co-examiner and advisor led to the identification of the following solutions as potential candidates for solution:

- **Discourse**[1] - An open-source discussion platform which is widely used.
- **GitHub Issues**[2] - GitHub Issues is used as a discussion platform by the GrantShares-DAO. This DAO is maintained by AxLabs, the company of the external co-examiner.
- **Microservice** - A new microservice with a database for the discussion.

[1] `https://www.discourse.org/`
[2] `https://github.com/features/issues`

3.3.4 Results

Discourse

Discourse is an open-source discussion forum software that offers a modern and user-friendly interface with mobile optimization and flexibility for customization. It also prioritizes trust and safety with built-in features like user moderation tools and community guidelines. Discourse provides community-building tools, including badges, user profiles, and private messaging, to encourage participation and create a community among members. In addition, the platform offers detailed analytics on user engagement, topics, and discussions, which enables community managers to monitor the performance of their community and make improvements where necessary.

Cost
Discourse is 100% open-source and free to use, except for hosting costs. Discourse also offers to host their product for interested customers. Pricing plans start at $50 per month [11], which is not feasible for a small community like FlatFeeStack.

Ease of use
The user experience of discourse is excellent. It is a widely used software with over 37k stars on GitHub.

Integration
There is no integration possible into the current FlatFeeStack frontend. The application would have a different URL, requiring the user to go to another page to engage in discussion. It is possible to have single sign-on, but one must configure it and choose the provider, such as GitHub or Google.

Restrictions
This restriction is not possible out of the box. As previously mentioned, verifying whether the user is a FlatFeeStack user must be done at another point, even with the option for single sign-on.

GitHub Issues

A feature of the well-known code hosting service GitHub called GitHub Issues allows users to discuss a particular project or repository, raise issues, and make improvement suggestions. It offers a cooperative and open channel for users and developers to discuss a project and cooperate on its improvement. Each issue receives a unique number to aid in management and prioritization and can be tagged, assigned to specific users, and categorized.

Cost
This solution would be free, but it would need some time to implement the GitHub REST API[1].

Ease of use
Most of the developers are familiar with GitHub and the GitHub Issues. It is easy to use and user-friendly.

Integration
With the GitHub REST API, it would also be possible to integrate the content of the GitHub Issues into the FlatFeeStack frontend. This would be a nice feature, but it would need some time to implement. Without this integration, the user must go to another page to discuss.

Restrictions
Restricting access to the project in GitHub with the issues is possible for certain individuals. However, someone must sync the GitHub users with the FlatFeeStack users.

Microservice

The FlatFeeStack platform will have a new microservice added to it that will provide a REST API with all relevant endpoints for creating, reading, updating, and deleting discussion threads. The PostgreSQL[2] database will store discussion threads, and the microservice development will use Go, similar to the other microservices.

Cost
Hosting the microservice and the database would add little to the costs, but implementing the microservice would take some time.

Ease of use
The user experience can be adjusted and simplified to the needs.

[1]https://docs.github.com/en/rest/issues?apiVersion=2022-11-28
[2]https://www.postgresql.org/

Integration
This solution allows discussion threads to be easily integrated into the FlatFeeStack frontend.

Restrictions
The restriction is easily possible with this solution. We can check the user authentication without the need to implement something different.

3.3.5 Conclusion

Discourse (3.3.4) can not be integrated into the current FlatFeeStack frontend, and the user must go to another page to discuss. Furthermore, another application without prior experience would need to be hosted.

GitHub Issues (3.3.4) can be integrated into the FlatFeeStack frontend, but the user management would be complex. Also, the users would have to authenticate themselves on GitHub in the FlatFeeStack platform to be able to participate in the discussions.

A new microservice (3.3.4) can be integrated into the FlatFeeStack frontend, and the user management will be simple to implement as there is already a microservice in place. Implementing the goal proposal lifecycle (3.9) would be simple, and the user would not have to go to another page to discuss.

Chapter 4

Solution

This chapter provides an overview of the outcomes and solutions achieved in this thesis. The results encompass a range of tasks that were addressed, and their corresponding solutions are presented in detail.

4.1 Application Architecture

This chapter presents the final architecture of the application, which has undergone a series of refactorings based on the analysis conducted in 2.3 and the outcomes of discussions with the advisors. The C4 Model[1] notation clearly and concisely represents the architecture's components and relationships.

The architecture is structured to facilitate modularity and separation of concerns, allowing independent development and deployment of different components. It emphasizes the use of well-defined interfaces for loose coupling and improved flexibility.

4.1.1 System Context

The system context diagram, representing the external systems and people interacting with the application, remains unchanged throughout the refactorings. No alterations or modifications were made to the external systems or the individuals involved in the system's operation.

Please refer to the diagram provided in 2.3.1 for a visual representation of the system context diagram.

[1]https://c4model.com/

4.1.2 Container

Figure 4.1 shows the container diagram of the application. The green color signals a new container, component, or relation.

The analysis results from 3.3 is the new forum container and its relations to the frontend, backend, and database. Detailed information about the implementation of the forum can be found in 4.5.

The WebSockets were removed entirely due to reducing complexity.

The use of a common go library is also new but not shown in the diagram. Usage of this shared go library will be displayed in the component diagrams in 4.1.3. More on this topic is described in 4.7.11.

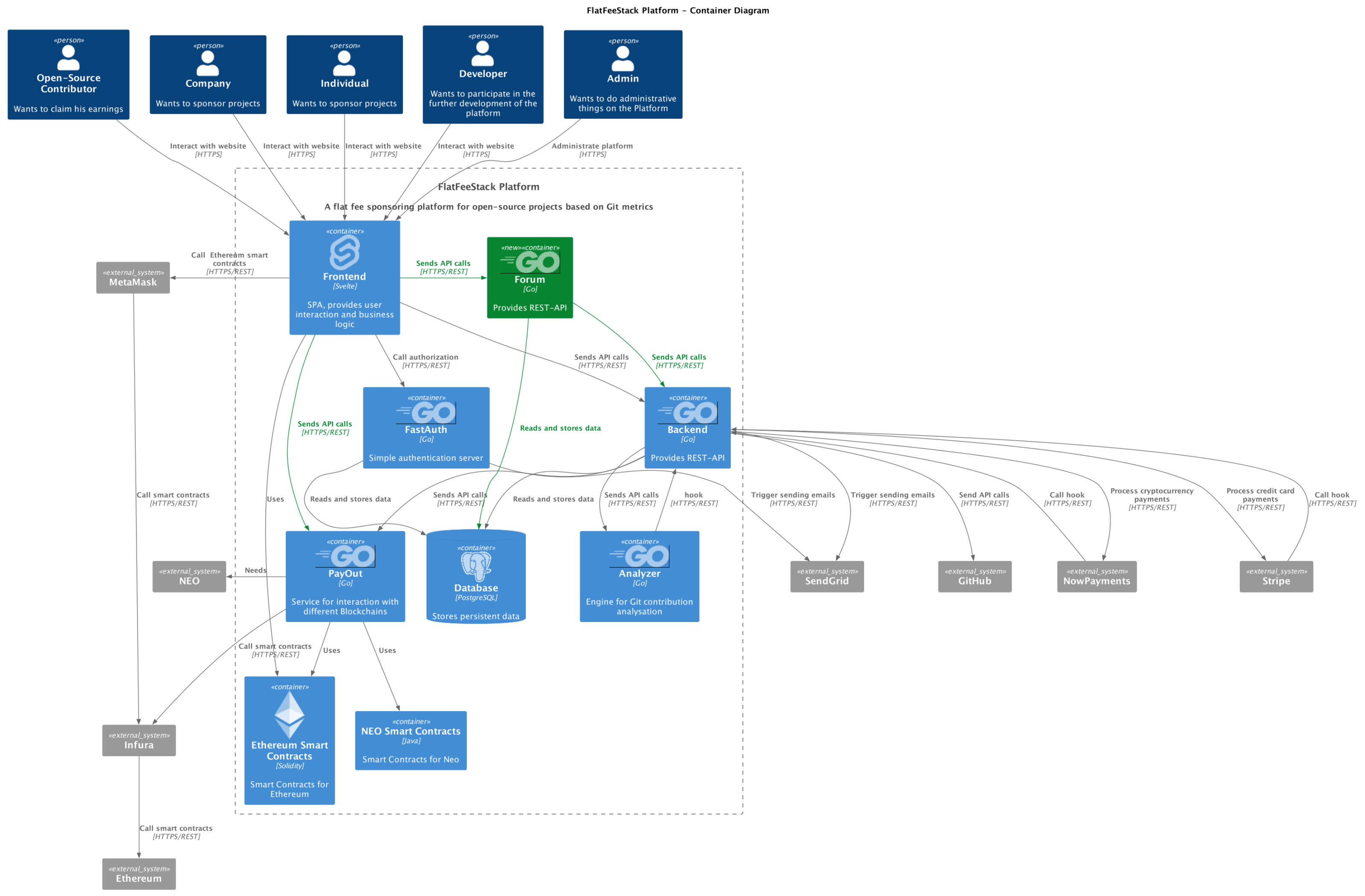

FlatFeeStack Platform - Container Diagram
«person» Open-Source Contributor Wants to claim his earnings
«person» Company Wants to sponsor projects
«person» Individual Wants to sponsor projects
«person» Developer Wants to participate in the further development of the platform
«person» Admin Wants to do administrative things on the Platform
Interact with website [HTTPS]
Interact with website [HTTPS]
Interact with website [HTTPS]
Interact with website [HTTPS]
Administrate platform [HTTPS]
FlatFeeStack Platform
A flat fee sponsoring platform for open-source projects based on Git metrics
«container» Frontend [Svelte] SPA, provides user interaction and business logic
«new»«container» Forum [Go] Provides REST-API
«external_system» MetaMask
Call Ethereum smart contracts [HTTPS/REST]
Sends API calls [HTTPS/REST]
Call authorization [HTTPS/REST]
Sends API calls [HTTPS/REST]
Sends API calls [HTTPS/REST]
«container» FastAuth [Go] Simple authentication server
Reads and stores data
«container» Backend [Go] Provides REST-API
Sends API calls [HTTPS/REST]
Call smart contracts [HTTPS/REST]
Uses
Reads and stores data
Sends API calls [HTTPS/REST]
Reads and stores data
Sends API calls [HTTPS/REST]
hook [HTTPS/REST]
Trigger sending emails [HTTPS/REST]
Trigger sending emails [HTTPS/REST]
Send API calls [HTTPS/REST]
Call hook [HTTPS/REST]
Process cryptocurrency payments [HTTPS/REST]
Process credit card payments [HTTPS/REST]
Call hook [HTTPS/REST]
«external_system» NEO
Needs
«container» PayOut [Go] Service for interaction with different Blockchains
«container» Database [PostgreSQL] Stores persistent data
«container» Analyzer [Go] Engine for Git contribution analysation
«external_system» SendGrid
«external_system» GitHub
«external_system» NowPayments
«external_system» Stripe
Call smart contracts [HTTPS/REST]
Uses
Uses
«container» Ethereum Smart Contracts [Solidity] Smart Contracts for Ethereum
«container» NEO Smart Contracts [Java] Smart Contracts for Neo
«external_system» Infura
Call smart contracts [HTTPS/REST]
«external_system» Ethereum

Figure 4.1: Container diagram.

4.1.3 Component

This section shows the component diagrams of the different containers of the application. The green color signals a new container, component, or relation. For a better overview, the diagrams are split by container and only show the direct relations. The entire component diagram can be found in Appendix D.2.

Frontend

The frontend now interacts with the forum component to retrieve, show, and edit data. This is illustrated in Figure 4.2. The payout service is called to generate to allow developers retrieve their retrieved funds and to get the contract addresses for the DAO.

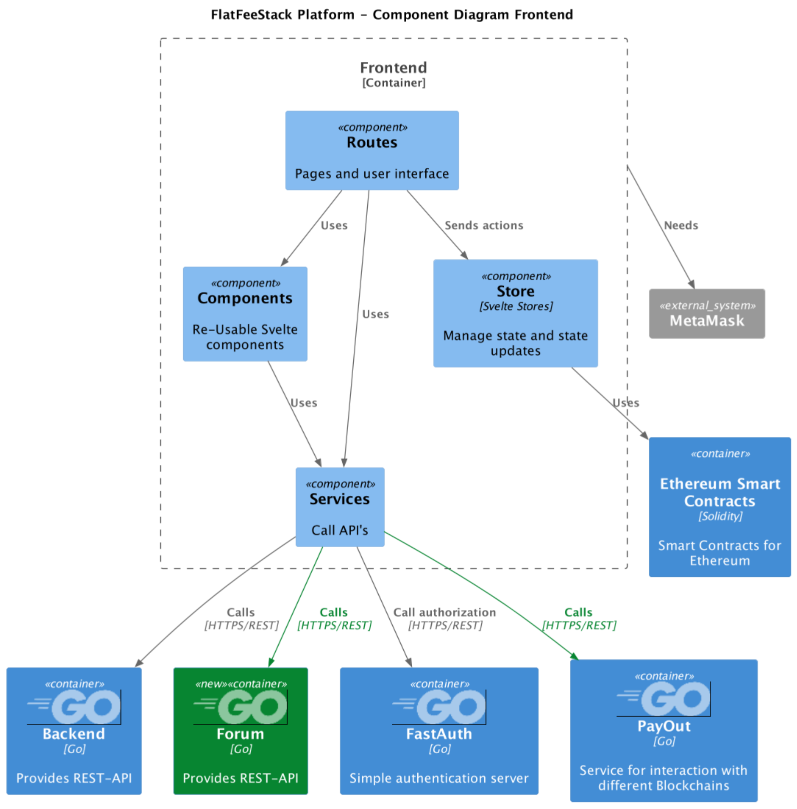

FlatFeeStack Platform – Component Diagram Frontend
Frontend [Container]
«component» Routes
Pages and user interface
Uses
Sends actions
Needs
«component» Components
Re-Usable Svelte components
Uses
«component» Store [Svelte Stores]
Manage state and state updates
«external_system» MetaMask
Uses
Uses
«container» Ethereum Smart Contracts [Solidity]
Smart Contracts for Ethereum
«component» Services
Call API's
Calls [HTTPS/REST]
Calls [HTTPS/REST]
Call authorization [HTTPS/REST]
Calls [HTTPS/REST]
«container» Backend [Go]
Provides REST-API
«new»«container» Forum [Go]
Provides REST-API
«container» FastAuth [Go]
Simple authentication server
«container» PayOut [Go]
Service for interaction with different Blockchains

Figure 4.2: Component diagram frontend.

Backend

Figure 4.3 shows the component diagram of the backend. Thomas Bocek did most of these refactorings. The forum component (4.1.3) calls the backend to retrieve user data. As mentioned, the WebSockets were removed and replaced by pulling the backend service in an interval. The API component is now responsible for the full implementation of the REST API. The original API component was renamed to clients and serves as an abstraction layer for used services. The sending of emails was moved to the new clients component. The calls to NOWPayments[1] and Stripe[2] have been relocated to the API component.

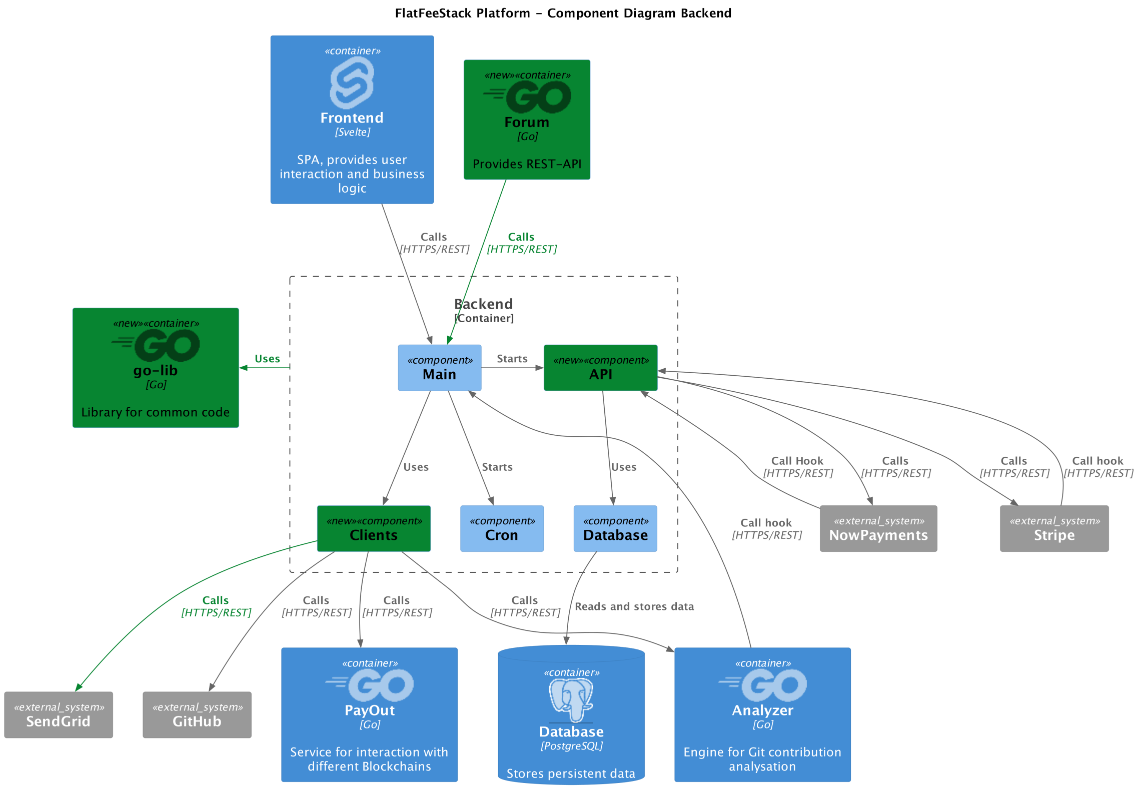

FlatFeeStack Platform - Component Diagram Backend
«container» Frontend [Svelte] SPA, provides user interaction and business logic
«new»«container» Forum [Go] Provides REST-API
Calls [HTTPS/REST]
Calls [HTTPS/REST]
Backend [Container]
«new»«container» go-lib [Go] Library for common code
Uses
«component» Main
Starts
«new»«component» API
Uses
Starts
Uses
«new»«component» Clients
«component» Cron
«component» Database
Call Hook [HTTPS/REST]
Calls [HTTPS/REST]
Calls [HTTPS/REST]
Call hook [HTTPS/REST]
Call hook [HTTPS/REST]
«external_system» NowPayments
«external_system» Stripe
Calls [HTTPS/REST]
Calls [HTTPS/REST]
Calls [HTTPS/REST]
Calls [HTTPS/REST]
Reads and stores data
«external_system» SendGrid
«external_system» GitHub
«container» PayOut [Go] Service for interaction with different Blockchains
«container» Database [PostgreSQL] Stores persistent data
«container» Analyzer [Go] Engine for Git contribution analysation

Figure 4.3: Component diagram backend.

Analyzer

The architecture of the analyzer component, shown in 4.4, has remained the same, except it makes use of the go-library (4.7.11).

[1]https://nowpayments.io/
[2]https://stripe.com

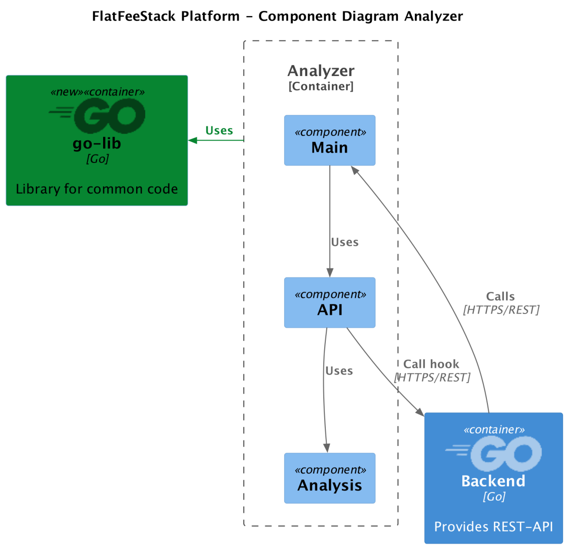


Figure 4.4: Component diagram analyzer.

FastAuth

The architecture of the fastauth component, shown in 4.5, has remained the same, except it makes use of the go-library (4.7.11). Additionally, over 1500 lines of unused code were removed.

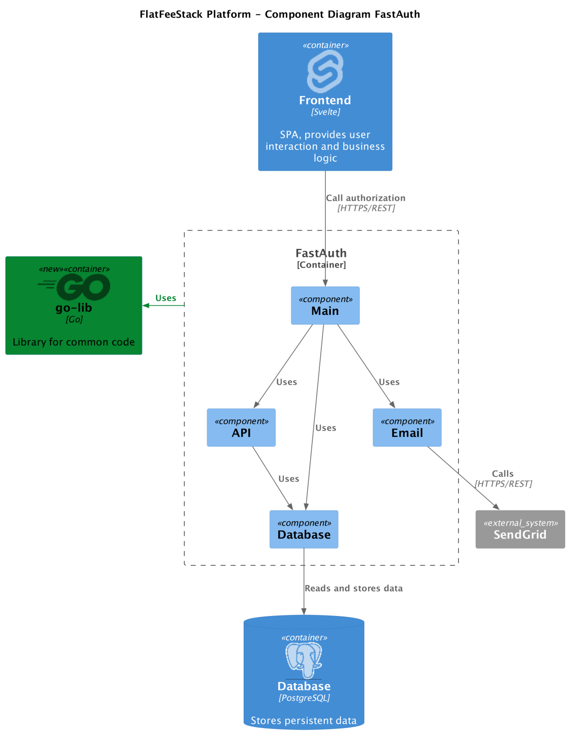


Figure 4.5: Component diagram fastauth.

Payout

The architecture of the payout component, shown in 4.6, has remained the same, except it makes use of the go-library (4.7.11).

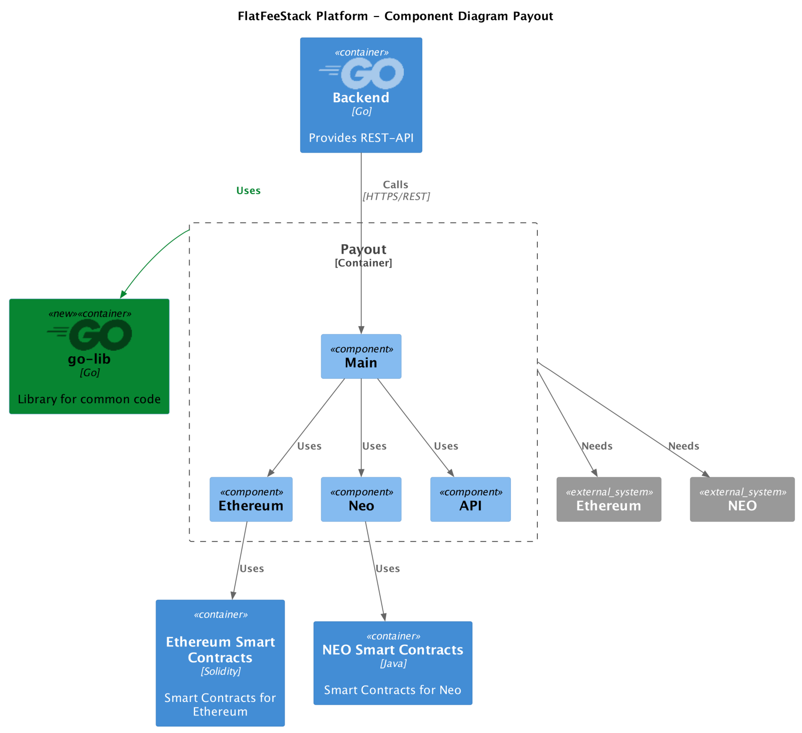


Figure 4.6: Component diagram payout.

Smart Contracts

The smart contract component shown in 4.7 has had a minor change in the payout contracts. The new payout USDC contract is an adapted version of the original payout contract. They share the same interface and signature algorithm, but since the methods to interact with USDC, which is an ERC20 token in its core, and native ETH are different, this refactoring was necessary. The payout USDC contract has been written to work with any ERC20 token. For example, if FlatFeeStack wants to support pay-ins with CHF at some point, the payout could be done with the CCHF stablecoin, which is also an ERC20 token.

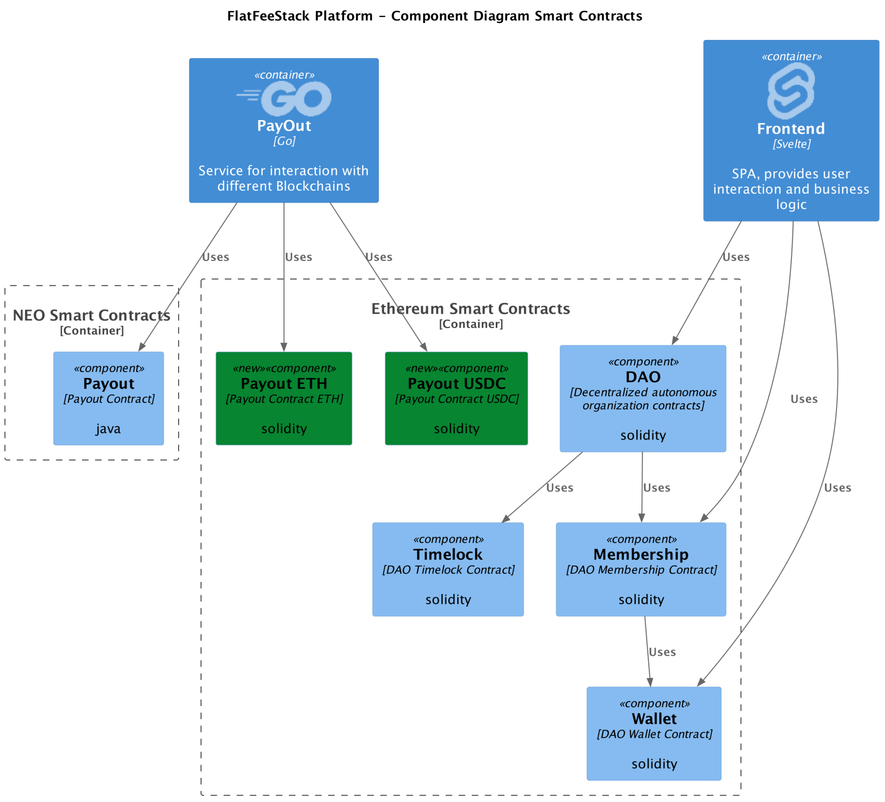


Figure 4.7: Component diagram smart contracts.

Forum

The component diagram of the forum is shown in section 4.5.2.

4.2 Mono Repository Merge

This task aimed to merge all services of FlatFeeStack into one repository. Additionally, continuous integration and automated dependency updates should be added.

A repository had to be chosen to start with the mono repository merge, where all the repositories would be added. As the central FlatFeeStack repository already had a script that downloaded all dependent services into subdirectories and a docker-compose file to start the whole platform, it was decided to use that repository. It would allow merging the repositories one by one without disrupting much of the existing development workflow.

The merge process for one of the services looked as follows [12]:

- Rewrite the Git history of the service locally to move everything into a subdirectory.
- From the FlatFeeStack repository, add the local service repository as a remote.
- Merge the state of the local repository using `--allow-unrelated-histories`, as the initial commit for the service and FlatFeeStack repository are different.
- Additionally, for each service, continuous integration has been added.
- Each service already had a Dockerfile to build a Docker image and usually unit tests. The continuous integration runs with GitHub Actions and only when files of said service have been changed, as shown in Listing 4.1. Sometimes tests in the source Git repository were broken, but they were fixed before the merge.

```yaml
name: Test and build analyzer

on:
  push:
    paths:
      - ".github/workflows/analyzer.yaml"
      - "analyzer/**"
  pull_request:
    paths:
      - ".github/workflows/analyzer.yaml"
      - "analyzer/**"
  workflow_dispatch:

jobs:
  build-test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up QEMU
        uses: docker/setup-qemu-action@v2

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2

      - name: Build test container
        uses: docker/build-push-action@v4
        with:
          context: analyzer
          load: true
          target: builder
          tags: ghcr.io/flatfeestack/flatfeestack/analyzer:test

      - name: Run tests
        run: docker run -v $(pwd)/analyzer:/app --rm ghcr.io/flatfeestack/ ↵
flatfeestack/analyzer:test go test -v ./...

      - name: Build container
        uses: docker/build-push-action@v4
        with:
          context: analyzer
          load: true
          tags: ghcr.io/flatfeestack/flatfeestack/analyzer:test

      - name: Run analyzer
        run: docker run --env-file analyzer/.example.env --rm -d -p  ↵
9083:9083 ghcr.io/flatfeestack/flatfeestack/analyzer:test

      - name: Check if analyzer is reachable
        uses: nick-fields/retry@v2
        with:
          timeout_seconds: 15
          max_attempts: 5
          command: curl -v localhost:9083
```

Listing 4.1: Continuous integration pipeline for the analyzer service.

For automating dependency updates, Renovate[1] was chosen and added via GitHub apps[2]. Renovate scans the repository for various dependency files, a significant advantage compared to GitHub's built-in Dependabot[3], where every dependency file must be configured manually. Renovate opens a pull request for each dependency update. As shown in List-

[1]https://github.com/renovatebot/renovate
[2]https://docs.github.com/en/apps
[3]https://github.com/dependabot

ing 4.2, the configuration was adapted to group path and minor dependency updates per service, reducing the number of pull requests created.

```
{
 "groupName": "analyzer - all non-major dependencies",
 "matchPaths": [
   "analyzer/"
 ],
 "matchUpdateTypes": [
   "minor",
   "patch"
 ]
}
```

Listing 4.2: Renovate configuration to group dependency updates for the analyzer service.

In total, six services have been merged into one repository:

- analyzer
- backend
- fastauth
- frontend
- smart-contracts-eth (renamed from daa)
- smart-contracts-neo (renamed from payout-neo)

4.2.1 Payout

The payout service is the notable exception to the process outlined in section 4.2. Before the merge, it contained a go server to handle payouts from the FlatFeeStack platform to the different blockchains, a smart contract for ETH for the payout, and a compiled version for the NEO payout contract. The payout service also deployed both the NEO and ETH contracts.

The following refactoring has been applied before the merge into the main repository to align the payout service to the general structure of the mono repository, where every subdirectory represents one service:

- The payout contract in ETH has been moved to the smart-contracts-eth directory and integrated into the existing development setup of the FlatFeeStack DAO.
- The process of deploying smart contracts has been eliminated. It is recommended that the deployment of smart contracts be managed independently by each respective technology.

- The payout repository still has a copy of the ETH contract ABI and a compiled version of the NEO payout contract. It needs them to interact with the smart contracts on their respective blockchains.

4.3 Production Deployment

This chapter documents the deployment of the FlatFeeStack platform to the DigitalOcean App platform. The final solution consists of two parts: The configuration, as described in section 4.3.1, and the deployment from GitHub Actions, as described in section 4.3.2.

4.3.1 DigitalOcean App Platform

As documented in section 3.1.4, different services can be grouped into one application. One application has one domain under which the services are reachable.

Section 3.1.4 lists issues with the deployment from the mono repository. The evaluation suggests that Docker images are built from the existing Dockerfile from GitHub Actions and pushed to the DigitalOcean Docker registry (section 4.3.2). Those images are then referenced as the image for each service that composes the architecture of FlatFeeStack. Each service always uses the tag `main`, and auto-deploy of the tag has been disabled, so deployments to production are manually controlled through GitHub Actions (see section 4.3.2).

The FlatFeeStack container supports two ways to pass variables for configuration: Either environment variables or a .env file. Environment variables are used to pass the required configuration to align the DigitalOcean deployment with the staging deployment. Secrets like the JWT auth secret or credentials for the different APIs are marked as encrypted, which ensures that the values cannot be retrieved nor read from the user interface for security reasons.

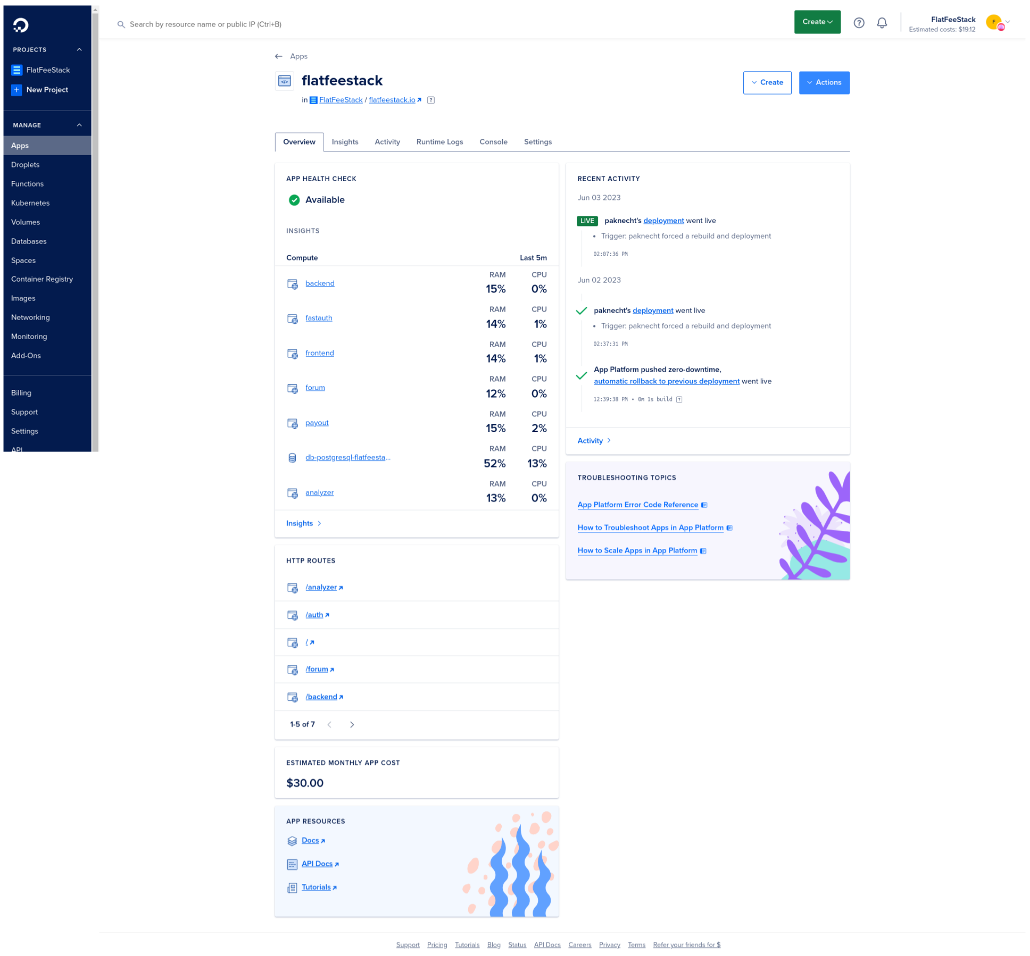
The staging environment uses a Caddy[1] reverse proxy to route the different subpaths to the respective services. DigitalOcean App platform allows the definition of the subpaths per service, which means the reverse proxy is built into the product. It allows later to quickly scale the services when necessary, as the reverse proxy of DigitalOcean knows how many instances are available for a particular service and can route the traffic correctly. The different subpaths are also shown in Figure 4.8.

The configuration on the App platform is extensive. DigitalOcean allows exporting this configuration in a YAML file called `app-spec`; a copy is available in the Git repository. In the worst case, the whole configuration can be rebuilt from this YAML file.

4.3.2 GitHub Actions

GitHub Actions play two critical parts in deployment architecture.

[1] `https://caddyserver.com/`



flatfeestack

in FlatFeeStack / flatfeestack.io

APP HEALTH CHECK

Available

INSIGHTS

Compute	RAM	CPU
backend	15%	0%
fastauth	14%	1%
frontend	14%	1%
forum	12%	0%
payout	15%	2%
db-postgresql-flatfeesta...	52%	13%
analyzer	13%	0%

Last 5m

Insights >

HTTP ROUTES

/analyzer
/auth
/
/forum
/backend

1-5 of 7

ESTIMATED MONTHLY APP COST

$30.00

APP RESOURCES

Docs
API Docs
Tutorials

RECENT ACTIVITY

Jun 03 2023

LIVE paknecht's deployment went live
- Trigger: paknecht forced a rebuild and deployment

02:07:36 PM

Jun 02 2023

paknecht's deployment went live
- Trigger: paknecht forced a rebuild and deployment

02:37:31 PM

App Platform pushed zero-downtime, automatic rollback to previous deployment went live

12:39:38 PM • 0m 1s build

Activity >

TROUBLESHOOTING TOPICS

App Platform Error Code Reference
How to Troubleshoot Apps in App Platform
How to Scale Apps in App Platform

Figure 4.8: Overview of the configured application on DigitalOcean.

The first jobs build and push the Docker images of the different services to the DigitalOcean registry. A configuration can look like the one shown in Listing 4.3.

```yaml
build-and-push-analyzer:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up QEMU
        uses: docker/setup-qemu-action@v2

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2

      - name: Install doctl
        uses: digitalocean/action-doctl@v2
        with:
          token: ${{ secrets.DIGITALOCEAN_ACCESS_TOKEN }}

      - name: Login to DigitalOcean container registry
        run: doctl registry login --expiry-seconds 10000

      - name: Build container
        uses: docker/build-push-action@v4
        with:
          context: analyzer
          load: true
          tags: "registry.digitalocean.com/flatfeestack/analyzer:${{ github.ref_name }}"
          cache-from: type=gha
          cache-to: type=gha,mode=max

      - name: Run analyzer
        run: docker run --env-file analyzer/.example.env --rm -d -p 9083:9083 "registry.digitalocean.com/flatfeestack/analyzer:${{ github.ref_name }}"

      - name: Check if analyzer is reachable
        uses: nick-fields/retry@v2
        with:
          timeout_seconds: 15
          max_attempts: 5
          command: curl -v localhost:9083

      - name: Push
        uses: docker/build-push-action@v4
        with:
          context: analyzer
          push: true
          tags: "registry.digitalocean.com/flatfeestack/analyzer:${{ github.ref_name }}"
```

Listing 4.3: Build and push job for the analyzer component.

A few essential details:

1. The setup is mainly based on the official example of Docker [13].
2. As a tag for the images, the current branch name, exposed as a GitHub Action variable named `ref_name` is used. Most Docker images usually use Git tags for versioning, but since the platform is continuously improved, having one tag is sufficient. One can avoid discussing how and when to add a new Git tag. It also allows building Docker tags for feature branches without introducing another tag/versioning scheme.

3. Before pushing the images to production, the containers are started using a set of example environment variables, and a smoke test is conducted to spot issues with the runtime.

Parts of the code listed in 4.3 are similar to the continuous integration workflow for each service listed in section 4.2. The main differences are that the `build and push` job always runs for all services on the main branch, ignores unit tests and pushes the Docker images.

This GitHub Action workflow is run automatically for all services on each push to the main branch and can be manually started for a feature branch.

The deployment to production is handled in a separate workflow which a developer can start manually.

```yaml
name: Deploy to production

on:
    workflow_dispatch:

jobs:
    deploy-to-production:
    runs-on: ubuntu-latest
    steps:
    - name: Install doctl
      uses: digitalocean/action-doctl@v2
      with:
        token: ${{ secrets.DIGITALOCEAN_ACCESS_TOKEN }}

    - uses: actions/setup-python@v4
      with:
        python-version: "3.11"

    - name: Trigger production deployment
      run: doctl apps create-deployment --wait $(doctl apps list --output ↵
json | python3 -c "import sys, json; print(json.load(sys.stdin)[0]['id'])")
```

Listing 4.4: Deployment to production workflow.

This script, listed in 4.4, will trigger a new deployment on DigitalOcean. Behind the scenes, DigitalOcean will pull the new images for each service, start them parallel to the existing ones and, once stable, delete the old ones.

Notably, this workflow can be started from any branch, but the tag that will be used is always main, as the configuration of the DigitalOcean App platform enforces it. If one wants to change the tag for a particular reason in production, one must change the used tag in the mentioned configuration.

4.3.3 Increased Price

Due to additions during the thesis, the monthly price of DigitalOcean is $71 instead of the projected $45 (see section 3.1.4). This increase has the following reasons:

1. One additional service, the forum (see section 4.5), costs $5.
2. The droplet for the monitoring (see section 4.6) costs $6.
3. With the forum component, the usable amount of Docker registry images of the evaluated pricing plan was exceeded. An upgrade to the next plan was needed, which costs an additional $15 per month.

4.4 Complete Payout Functionality

This chapter documents the additions to the different components of FlatFeeStack to finish the payout functionality.

4.4.1 Smart Contracts

As documented in section 3.2.3, the payout contract should be adapted to work with the USDC stablecoin. The USDC stablecoin uses the ERC20 interface [14], which differs from sending native Ethers.

Before modifying the existing contract, automated tests with the existing Hardhat environment were written to verify the functionalities. Then, standard methods were extracted to a new base payout contract before implementing a new variant that can handle ERC20 tokens.

Additionally, the signature was extended to must include a symbol, which is currently either *ETH* or *USDC*. This addition prevents users from withdrawing multiple assets at different smart contracts with the same signature.

4.4.2 Payout Service

In the payout service, the mentioned NEO endpoint was implemented according to the code mentioned in the thesis of Michael Bucher [9]. Another endpoint was added to generate the signature for payouts with the USDC stable tokens.

```go
func getEthSignature(data PayoutRequest, symbol string) (PayoutResponse, ↵
error) {
 var arguments abi.Arguments
 arguments = append(arguments, abi.Argument{
  Type: abi.Type{T: abi.FixedBytesTy, Size: 32},
 })
 arguments = append(arguments, abi.Argument{
  Type: abi.Type{T: abi.StringTy},
 })
 arguments = append(arguments, abi.Argument{
  Type: abi.Type{Size: 256, T: abi.UintTy},
 })
 arguments = append(arguments, abi.Argument{
  Type: abi.Type{T: abi.StringTy},
 })

 correspondingPrivateKey, err := privateKeyFromOpts(symbol)
 if err != nil {
  return PayoutResponse{}, err
 }

 privateKey, err := crypto.HexToECDSA(correspondingPrivateKey)
 if err != nil {
  return PayoutResponse{}, err
 }

 encodedUserId := [32]byte(crypto.Keccak256([]byte(data.UserId.String())))
 packed, err := arguments.Pack(encodedUserId, "#", data.Amount, symbol)
 hashRaw := crypto.Keccak256(packed)

 // Add Ethereum Signed Message prefix to hash
 prefix := []byte("\x19Ethereum Signed Message:\n32")
 prefixedHash := crypto.Keccak256(append(prefix, hashRaw[:]...))

 signature, err := crypto.Sign(prefixedHash[:], privateKey)
 if err != nil {
  return PayoutResponse{}, err
 }

 return PayoutResponse{
  Amount: data.Amount,
  Currency: symbol,
  EncodedUserId: hexutil.Encode(encodedUserId[:]),
  Signature: hexutil.Encode(signature),
 }, nil
}
```

Listing 4.5: The adapted signature function in the payout service.

The existing signature implementation for ETH had to be adapted, as the message now needed to contain a symbol (see section 4.4.1). This work led to another discovery that the UUIDs used as IDs for users in FlatFeeStack are too long for the `bytes32` type defined for the user IDs in the ETH contract. Therefore, an additional hashing step was introduced for the user IDs in both the signing endpoints and the smart contracts. The used `keccak256` hashing algorithm shortens the UUID to the required 32 bytes. The adapted code is shown in Listing 4.5.

4.4.3 Backend

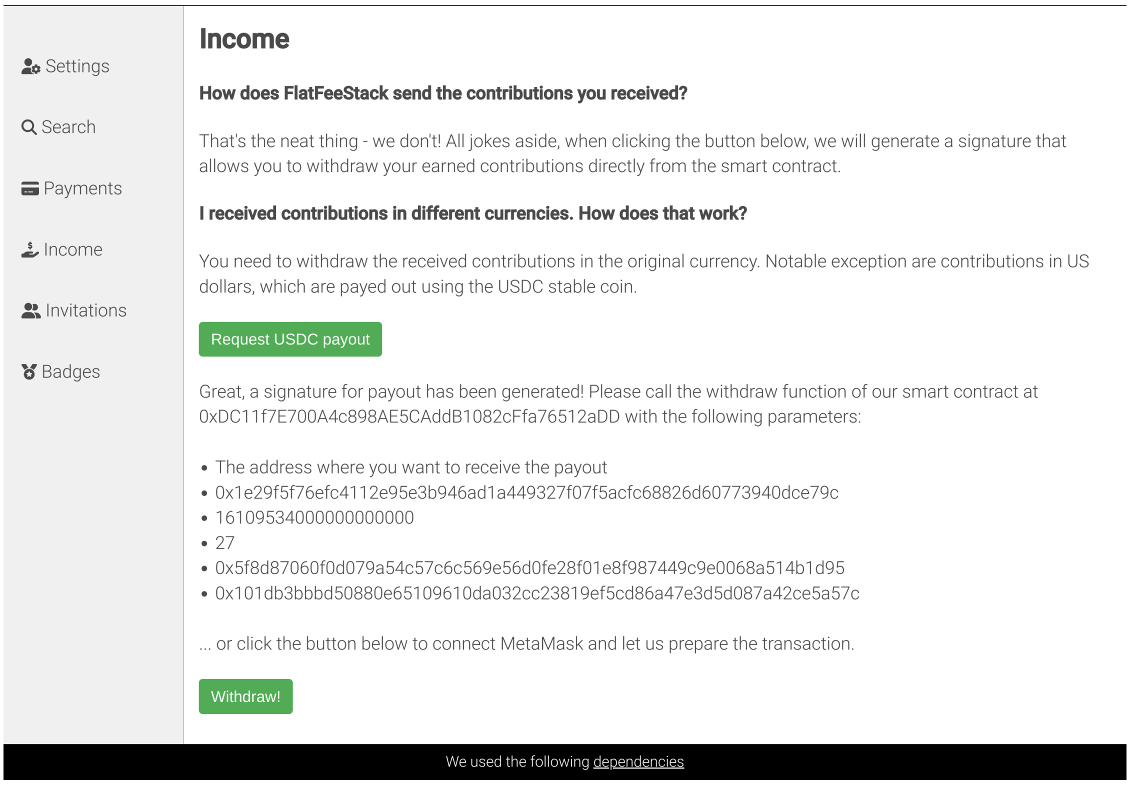
The needed backend endpoint was implemented according to the instruction listed in 3.2.2.

1. The user needs to be authenticated and provide a valid target currency.
2. All contributions received in the target currency for the current user are summed up.

3. Submit a request to the payout service to generate a signature in the target currency with the calculated total earned amount.
4. Mark the contributions as claimed in the database.
5. Return the generated signature, encoded user ID, and the amount.

4.4.4 Frontend

The existing Income component was extended with a short question-and-answer section about the payout process. Buttons were added to request a signature in each supported currency, as shown in 4.9.



Income

How does FlatFeeStack send the contributions you received?

That's the neat thing - we don't! All jokes aside, when clicking the button below, we will generate a signature that allows you to withdraw your earned contributions directly from the smart contract.

I received contributions in different currencies. How does that work?

You need to withdraw the received contributions in the original currency. Notable exception are contributions in US dollars, which are payed out using the USDC stable coin.

Great, a signature for payout has been generated! Please call the withdraw function of our smart contract at 0xDC11f7E700A4c898AE5CAddB1082cFfa76512aDD with the following parameters:

- The address where you want to receive the payout
- 0x1e29f5f76efc4112e95e3b946ad1a449327f07f5acfc68826d60773940dce79c
- 16109534000000000000
- 27
- 0x5f8d87060f0d079a54c57c6c569e56d0fe28f01e8f987449c9e0068a514b1d95
- 0x101db3bbbd50880e65109610da032cc23819ef5cd86a47e3d5d087a42ce5a57c

... or click the button below to connect MetaMask and let us prepare the transaction.

We used the following dependencies

Figure 4.9: Adapted income component in the frontend. The table with the received contributions have been cropped to save space in the documentation.

Additionally, when requesting a signature in ETH or USDC, a button was added to call MetaMask[1] to do the payout directly.

[1]https://metamask.io/

4.5 Discussion Implementation

As decided in 3.3, the discussion will be implemented as a microservice. The following chapter will describe the architecture and design of the discussion microservice.

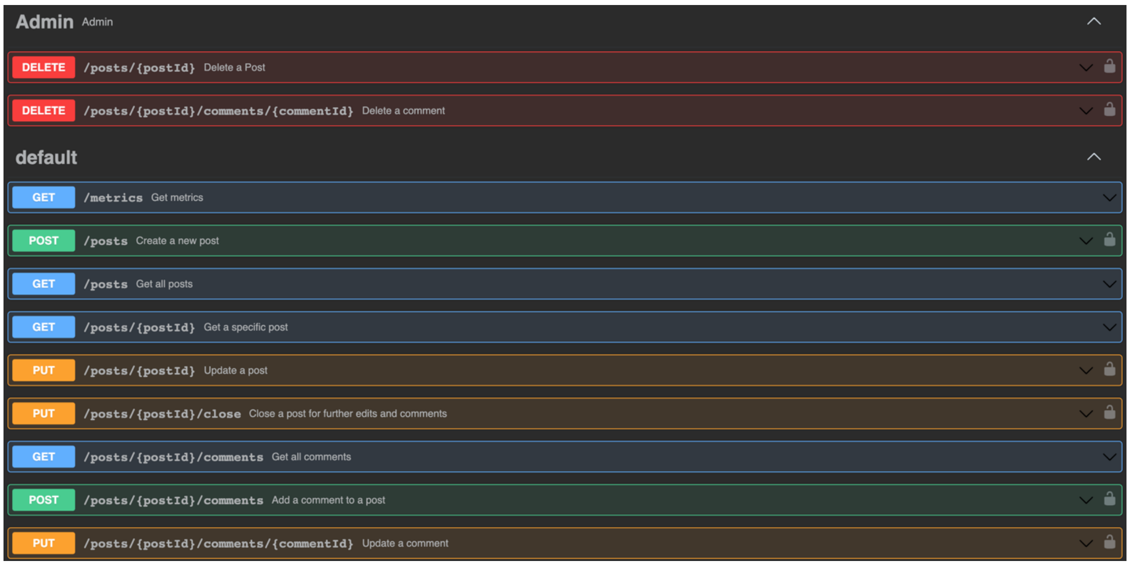
4.5.1 API Design

The discussion microservice should be as simple as possible within this project's scope. Therefore, a simple REST API is implemented with a CRUD interface. Libraries should be used to generate as much code as possible. The API design should be made with the API-First approach. This approach says that the API should be designed before the implementation starts.

Requirements

- Everyone must be able to read the forum.
- Only authenticated users can create and update posts and comments.
- Administrators can delete posts and comments.
- If the discussion is finished, the post can be closed.
- Closed posts can not be reopened or edited.

These requirements lead to the API endpoints shown in Figure 4.10.



Admin Admin

DELETE `/posts/{postId}` Delete a Post

DELETE `/posts/{postId}/comments/{commentId}` Delete a comment

default

GET `/metrics` Get metrics

POST `/posts` Create a new post

GET `/posts` Get all posts

GET `/posts/{postId}` Get a specific post

PUT `/posts/{postId}` Update a post

PUT `/posts/{postId}/close` Close a post for further edits and comments

GET `/posts/{postId}/comments` Get all comments

POST `/posts/{postId}/comments` Add a comment to a post

PUT `/posts/{postId}/comments/{commentId}` Update a comment

Figure 4.10: Forum endpoints.

4.5.2 Architecture

Figure 4.11 shows the architecture of the discussion microservice.

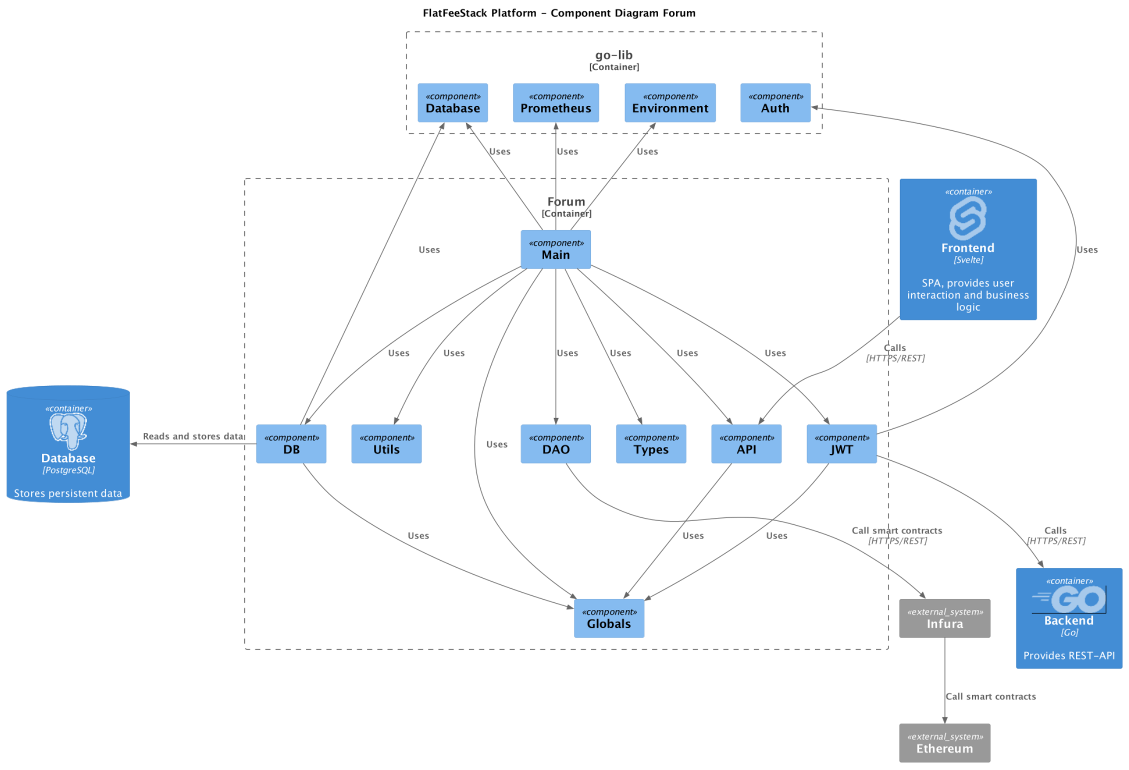


Figure 4.11: Component diagram forum.

Main

The main component is the entry point of the microservice. It constructs everything and starts up the server.

JWT

The JWT component is responsible for the user's authentication because some functions are only available for authenticated users (see 4.5.1). Depending on the endpoint, it checks whether the user needs to be authenticated or even admin to access it.

DB

The DB component is responsible for communication with the database. It establishes the connection to the database and executes the database creation on startup. Also, it provides functions to access the database.

Types

The types component defines the global type definitions that the microservice uses.

Utils

The utils component offers useful functions implemented in many components.

API
The API-Component is responsible for providing the code for the API endpoints.

Globals
The globals component declares and holds global variables used in other components.

DAO
The DAO component manages the calls to the FlatFeeStack DAO on the blockchain.

4.5.3 Implementation

The OpenAPI 3.0[1] standard is used to define the API. OpenAPI 3.0 is a highly prevalent and industry-accepted standard designed to streamline the process of designing and documenting RESTful APIs. It offers a structured and machine-readable framework that enables developers to define various aspects of an API, including its endpoints, requests and response payloads, and authentication mechanisms.

OpenAPI 3.0 has a great community and many tools to generate code from the API definition. The tool oapi-codegen[2] generates code from the OpenAPI 3.0 definition for the go environment.

The generated code was separated into model and server code to improve the code organization and for a more straightforward overview.

With the configuration shown in Listing 4.6, the model code can be generated:

```yaml
package: api
generate:
  models: true
output: ./api/forum-types.gen.go
```

Listing 4.6: Model generation configuration file.

This generates all models defined in the OpenAPI 3.0 definition, the request types, and the error models. A section of this file can be seen in Listing 4.7.

```go
// Package api provides primitives to interact with the openapi HTTP API.
//
// Code generated by github.com/deepmap/oapi-codegen version v1.12.4 DO NOT ↵
EDIT.
package api

import (
   "time"

```

[1]https://swagger.io/specification/
[2]https://github.com/deepmap/oapi-codegen

```go
	openapi_types "github.com/deepmap/oapi-codegen/pkg/types"
)

const (
	BearerAuthScopes = "bearerAuth.Scopes"
)

// Comment defines model for Comment.
type Comment struct {
	Author openapi_types.UUID `json:"author"`
	Content string `json:"content"`
	CreatedAt time.Time `json:"created_at"`
	Id openapi_types.UUID `json:"id"`
	UpdatedAt *time.Time `json:"updated_at,omitempty"`
}

...

// BadRequest defines model for BadRequest.
type BadRequest struct {
	Error string `json:"error"`
}

...

// PostPostsJSONRequestBody defines body for PostPosts for application/json ↵
ContentType.
type PostPostsJSONRequestBody = PostInput

...
```

Listing 4.7: Generated model from OpenAPI.

With the configuration shown in Listing 4.8, the server code can be generated:

```yaml
package: api
generate:
  gorilla-server: true
  strict-server: true
  embedded-spec: true
output: ./api/forum-server.gen.go
```

Listing 4.8: Server generation configuration file.

This oapi-codegen configuration generates a Go language server implementation using the Gorilla toolkit, enforces strict server code generation, and embeds the OpenAPI specification within the generated server code.

It generates an interface for the user to implement. Due to the strict-server option, users are compelled to utilize the generated types, which handle the parsing of request and response objects. A section of the generated server code can be seen in Listing 4.9.

```go
// Interface
type StrictServerInterface interface {
   // Get metrics
   // (GET /metrics)
   GetMetrics(ctx context.Context, request GetMetricsRequestObject) (GetMetricsResponseObject, error)
   // Get all posts
   // (GET /posts)
   GetPosts(ctx context.Context, request GetPostsRequestObject) (GetPostsResponseObject, error)
   // Create a new post
   // (POST /posts)
   ...
}
// Implementation
...
func (s *StrictServerImpl) GetMetrics(ctx context.Context, request GetMetricsRequestObject) (GetMetricsResponseObject, error) {
   return GetMetrics200Response{}, nil
}

func (s *StrictServerImpl) GetPosts(ctx context.Context, request GetPostsRequestObject) (GetPostsResponseObject, error) {
   posts, err := database.GetAllPosts()
   if err != nil {
     log.Error(err)
     return GetPosts500Response{}, nil
   }
   if posts == nil {
     return GetPosts204JSONResponse{}, nil
   }
   var response GetPosts200JSONResponse
   for _, dbPost := range posts {
     post := mapDbPostToPost(dbPost)
     response = append(response, post)
   }
   return response, nil
}

func (s *StrictServerImpl) PostPosts(ctx context.Context, request PostPostsRequestObject) (PostPostsResponseObject, error) {
   id := getCurrentUserId(ctx)
```

```go
	newPost, err := database.InsertPost(id, request.Body.Title, request.Body.Content)
	if err != nil {
		log.Error(err)
		return PostPosts500Response{}, nil
	}
	return PostPosts201JSONResponse{
		Author: newPost.Author,
		Content: newPost.Content,
		CreatedAt: newPost.CreatedAt,
		Id: newPost.Id,
		Open: newPost.Open,
		Title: newPost.Title,
		UpdatedAt: newPost.UpdatedAt,
	}, nil
}
...
```

Listing 4.9: Server interface and implementation code.

Listing 4.10 shows how the server can be initiated using the generated code.

```go
...
// Create an instance of the implementation of the server interface
server := api.NewStrictServerImpl()
...
//Create a new router
router := mux.NewRouter()
...
// Define options for the Gorilla Server
serverOptions := api.GorillaServerOptions{
	BaseURL: "",
	BaseRouter: router,
	Middlewares: []api.MiddlewareFunc{
		jwt.AuthMiddleware,
	},
	ErrorHandlerFunc: func(w http.ResponseWriter, r *http.Request, err error) {
		utils.WriteErrorf(w, http.StatusInternalServerError, "Internal Server Error: %v", err.Error())
		return
	},
}
...
// Create an http handler with an interface and options
handler := api.HandlerWithOptions(serverInterface, serverOptions)
...

```

```go
// Start the server
log.Fatal(http.ListenAndServe(":"+strconv.Itoa(globals.OPTS.Port), handler))
```

Listing 4.10: Http handler code.

A simple authentication middleware, shown in Listing 4.11, is used to know if the client is authenticated and authorized. The generated code provides a field in the request context where the information is stored if the user needs authentication and authorization. This can be generated because the security schemes are defined in the OpenAPI 3.0 definition.

```go
func AuthMiddleware(next http.HandlerFunc) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    // Get the context from the request
    ctx := r.Context()

    scopes := ctx.Value(api.BearerAuthScopes)

    // If no scopes are defined, the endpoint is not protected and is ↵
    publicly available
    if scopes == nil {
      next.ServeHTTP(w, r)
      return
    }

    ...
    // Slice the scopes
    scopesSlice, ok := scopes.([]string)
    ...
    // If the scope is admin, we have to check that the user is admin
    // If the scope is user, it is enough to know that the JWT validation ↵
    was successful
    switch scopesSlice[0] {
    case "Admin":
      if dbUser.Role == "Admin" {
        next.ServeHTTP(w, r.WithContext(ctx))
        return
      }
      utils.WriteErrorf(w, http.StatusUnauthorized, "You are not admin: %v ↵
      ", claims.Subject)
      return
    case "User":
      next.ServeHTTP(w, r.WithContext(ctx))
    default:
      utils.WriteErrorf(w, http.StatusInternalServerError, "Unknown scope")
      return
    }
```

Listing 4.11: Authentication middleware function.

4.5.4 Ethereum Event Listener

When certain events happen on the blockchain, the forum takes automated action according to the proposal lifecycle shown in figure 3.9.

First, the forum needs an active connection to the ETH blockchain to receive such events. This connection is handled with WebSockets. Both the local ganache development environment and Infura offer those endpoints. Once a proposal is created, the DAO emits an event containing all its metadata, like a description or the ID.

The event listener in the forum is set up only to receive the *proposal created* events. It will inspect its description and looks for a discussion URL, as shown in Listing 4.12. If a valid URL is found, the forum will try to save the proposal ID to the discussion in its database, so the frontend can later display a link. If no link is found or the UUID of the discussion is invalid, the forum creates a new post.

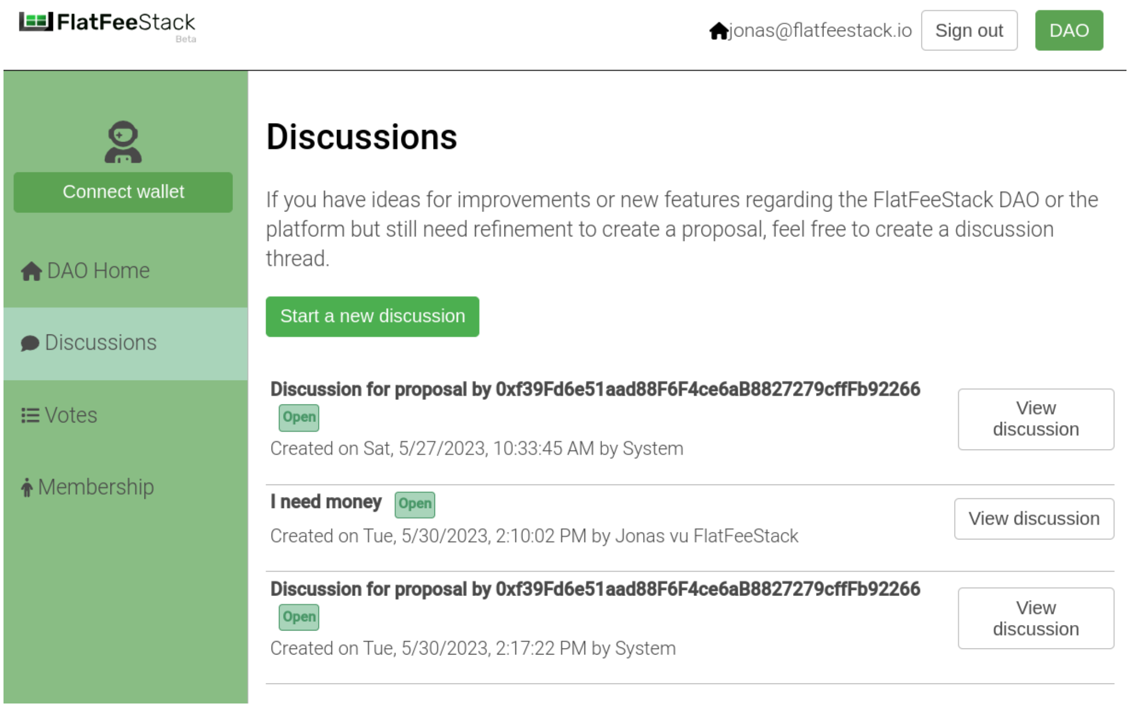
```go
func LinkOrCreateDiscussion(event ContractDAOProposalCreated) {
  // if the user selects a discussion in our "Create proposal" mask in the frontend
  // a line like "Original discussion: http://localhost:8080/dao/discussion/21a3c381-4bcf-4f4b-a341-a28365518af1" is added to the discussion
  linkPattern := regexp.MustCompile(`Original discussion\: [a-zA-Z\:\/\.\d]+\/dao\/discussion\/([0-9a-fA-F]{8}\b-[0-9a-fA-F]{4}\b-[0-9a-fA-F]{4}\b-[0-9a-fA-F]{4}\b-[0-9a-fA-F]{12})$`)
  matches := linkPattern.FindStringSubmatch(event.Description)

  if len(matches) == 0 {
    _, err := createPostForProposal(event)
    if err != nil {
      log.Errorf("Unable to create post for proposal: %s", err)
    }
  } else {
    // discussion is linked, check if reference is valid.
    updateExistingDiscussion(event, matches)
  }
}
```

Listing 4.12: Regular expression to detect a discussion link.

4.5.5 Frontend Implementation

The forum is added to the existing DAO navigation. An overview of all discussions is shown when opening it, as seen in Figure 4.12.

Figure 4.13 shows a detailed view of a discussion. At the top, links to the different proposals are provided, if any are linked. As the currently logged-in user is the creator of the



Discussions

If you have ideas for improvements or new features regarding the FlatFeeStack DAO or the platform but still need refinement to create a proposal, feel free to create a discussion thread.

Discussion for proposal by 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266 Open
Created on Sat, 5/27/2023, 10:33:45 AM by System

I need money Open
Created on Tue, 5/30/2023, 2:10:02 PM by Jonas vu FlatFeeStack

Discussion for proposal by 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266 Open
Created on Tue, 5/30/2023, 2:17:22 PM by System

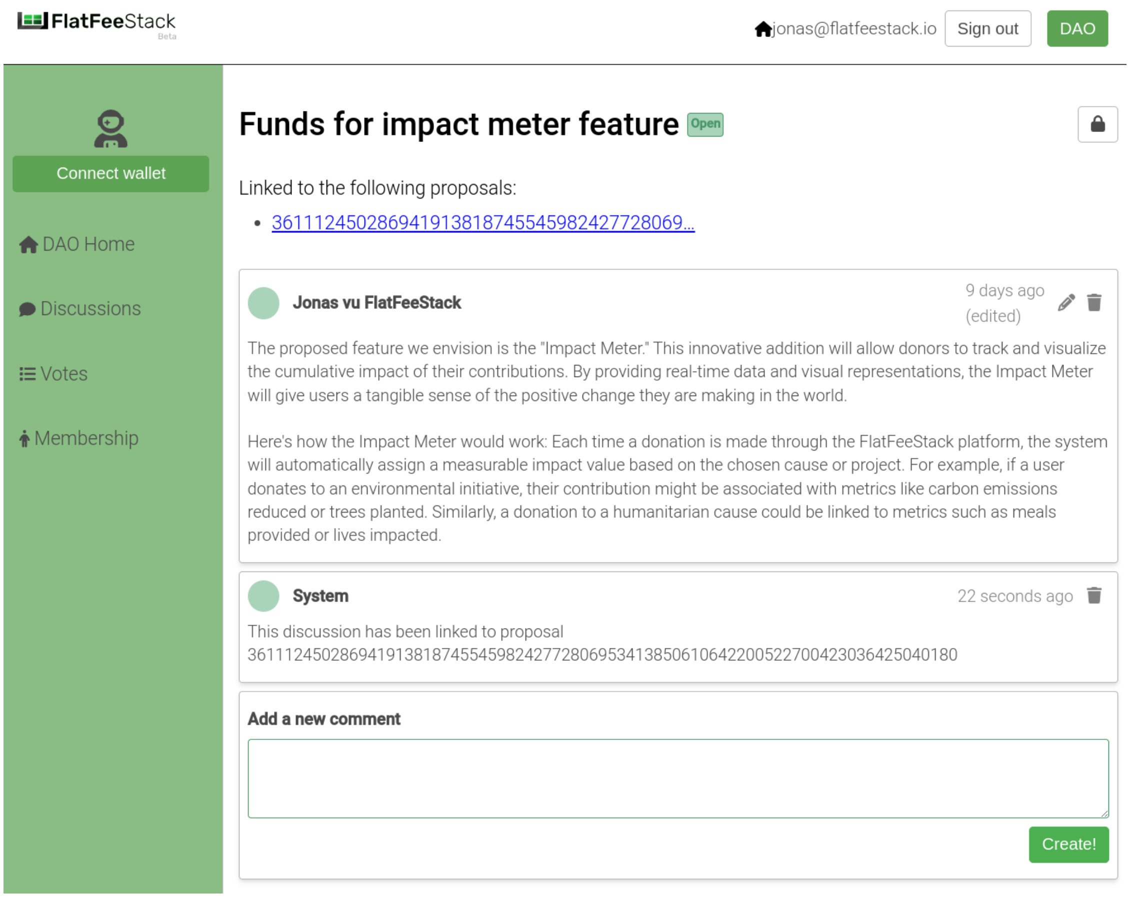
Figure 4.12: Discussion overview.

discussion, a lock icon is shown to allow closing the discussion. The pencil icon is displayed when the current user is the post creator or comment. Upon clicking it, a form is shown that allows the creator to edit the content. An *edited* remark on the post also indicates an edited entity.

As the logged-in user is also an administrator, the trash icon is displayed, allowing one to delete a post or comment.

Figure 4.14 shows the post creation form, which is identical to the form to update a post. It contains the two required fields, title and content.

The form uses client-side validations, which are aligned with the constraints implemented in the backend, so the user has a quicker feedback loop if something is not good with their provided text.



Funds for impact meter feature

Linked to the following proposals:

- 361112450286941913818745545982427728069...

Jonas vu FlatFeeStack 9 days ago (edited)

The proposed feature we envision is the "Impact Meter." This innovative addition will allow donors to track and visualize the cumulative impact of their contributions. By providing real-time data and visual representations, the Impact Meter will give users a tangible sense of the positive change they are making in the world.

Here's how the Impact Meter would work: Each time a donation is made through the FlatFeeStack platform, the system will automatically assign a measurable impact value based on the chosen cause or project. For example, if a user donates to an environmental initiative, their contribution might be associated with metrics like carbon emissions reduced or trees planted. Similarly, a donation to a humanitarian cause could be linked to metrics such as meals provided or lives impacted.

System 22 seconds ago

This discussion has been linked to proposal 36111245028694191381874554598242772806953413850610642200522700423036425040180

Add a new comment

Figure 4.13: Detail view for a discussion.

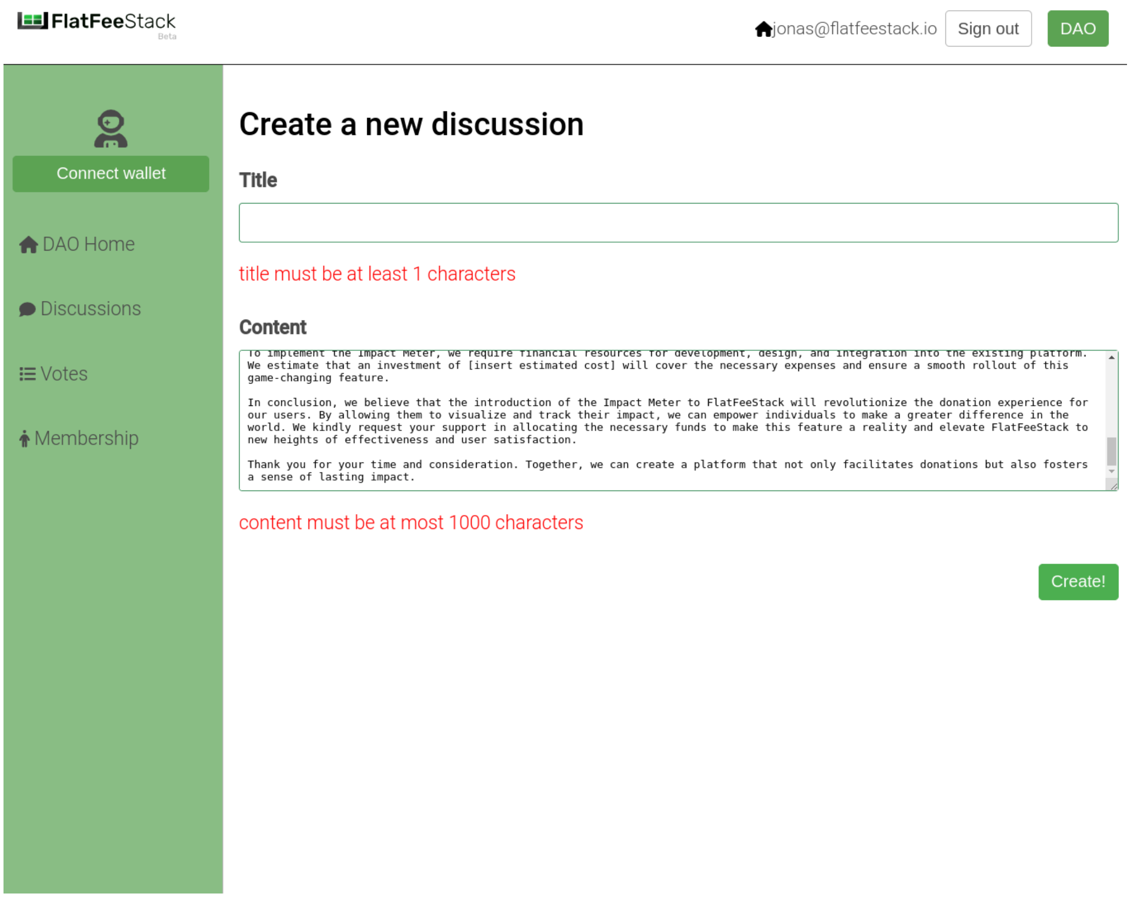
4.6 Monitoring

Monitoring is critical to managing the health and performance of microservices-based applications and their underlying databases. In this chapter, the focus is on the implementation of the monitoring for the different microservices in FlatFeeStack and the database.

Microservices architecture has gained significant popularity due to its ability to facilitate agile development, scalability, and fault isolation. However, the distributed nature of microservices poses monitoring challenges, as we need to track the performance and behavior of individual services and ensure the overall system's stability.

Prometheus[1] addresses these challenges as an open-source monitoring and alerting solution. Prometheus follows a pull-based model, where it regularly scrapes metrics from targets, such as microservices and databases, and stores them in a time-series database. It offers a flexible query language for querying and aggregating metrics and provides a powerful alerting mechanism to notify us of potential issues.

[1]https://prometheus.io/



Create a new discussion

Title

title must be at least 1 characters

Content

```
To implement the Impact Meter, we require financial resources for development, design, and integration into the existing platform.
We estimate that an investment of [insert estimated cost] will cover the necessary expenses and ensure a smooth rollout of this
game-changing feature.

In conclusion, we believe that the introduction of the Impact Meter to FlatFeeStack will revolutionize the donation experience for
our users. By allowing them to visualize and track their impact, we can empower individuals to make a greater difference in the
world. We kindly request your support in allocating the necessary funds to make this feature a reality and elevate FlatFeeStack to
new heights of effectiveness and user satisfaction.

Thank you for your time and consideration. Together, we can create a platform that not only facilitates donations but also fosters
a sense of lasting impact.
```

content must be at most 1000 characters

Figure 4.14: Form to create a new discussion.

Grafana[1], a popular data visualization and dashboarding tool, complements Prometheus by allowing us to create visually appealing and insightful dashboards. With Grafana, users can monitor key metrics, display real-time and historical data, and gain actionable insights into the performance and behavior of microservices and databases.

Prometheus and Grafana are widely used and have a large community. The community offers a rich set of exporters, dashboards, and alerts for monitoring and managing microservices and databases.

4.6.1 Instrumenting PostgreSQL

The PostgreSQL database provides great metrics but does not expose them in a Prometheus-compatible format. To address this issue, the PostgreSQL Exporter[2] is used to scrape metrics from the database and expose them in a Prometheus-compatible format.

[1]https://grafana.com/
[2]https://github.com/prometheus-community/postgres_exporter

4.6.2 Changes in Go-Code

A library called client_golang[1] can be used to receive metrics for a go application. This library provides functions to register metrics and expose them in a Prometheus-compatible format. Per default, this library collects data about the go runtime, like memory usage, garbage collection and more. Nevertheless, it is also possible to register custom metrics. Each microservice registers three custom metrics.

1. **Request Duration:** This metric measures the duration of the HTTP requests in the microservice.
2. **Total Requests:** This metric counts the requests to the different HTTP endpoints in the microservice.
3. **Response Status:** This metric counts the different response status codes the microservice sends.

The microservices had to extend a new endpoint to enable Prometheus to scrape the metrics. This endpoint is called */metrics* and can be seen in Listing 4.13.

```go
router.Path("/metrics").Handler(promhttp.HandlerFor(
     registry,
     promhttp.HandlerOpts{
       Registry: registry,
       // Opt into OpenMetrics to support exemplars.
       EnableOpenMetrics: true,
     },
   ))
```

Listing 4.13: Code to define metrics endpoint.

The registry contains all the metrics that should be collected and exposed.

Also, each microservice had to register a new middleware (shown in Listing 4.14) in its router. The middleware fills in data for the three custom metrics.

```go
func PrometheusMiddleware(next http.Handler) http.Handler {
   return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
     route := mux.CurrentRoute(r)
     path, _ := route.GetPathTemplate()

     timer := prometheus.NewTimer(HttpDuration.WithLabelValues(path))
     rw := &responseWriterWrapper{ResponseWriter: w}
     next.ServeHTTP(rw, r)

     statusCode := rw.statusCode
```

[1]https://github.com/prometheus/client_golang

```go

    ResponseStatus.WithLabelValues(strconv.Itoa(statusCode)).Inc()
    TotalRequests.WithLabelValues(path).Inc()

    timer.ObserveDuration()
  })
}
```

Listing 4.14: Prometheus middleware definition.

Common code for the metric endpoints has been extracted to the FlatFeeStack `go-lib` ↵ (4.7.11).

4.6.3 Dashboards

Based on the available metrics, the decision was made to create four different dashboards.

FlatFeeStack Dashboard

Figure 4.15 shows the FlatFeeStack-Dashboard. It shows metrics which are collected directly from the database.

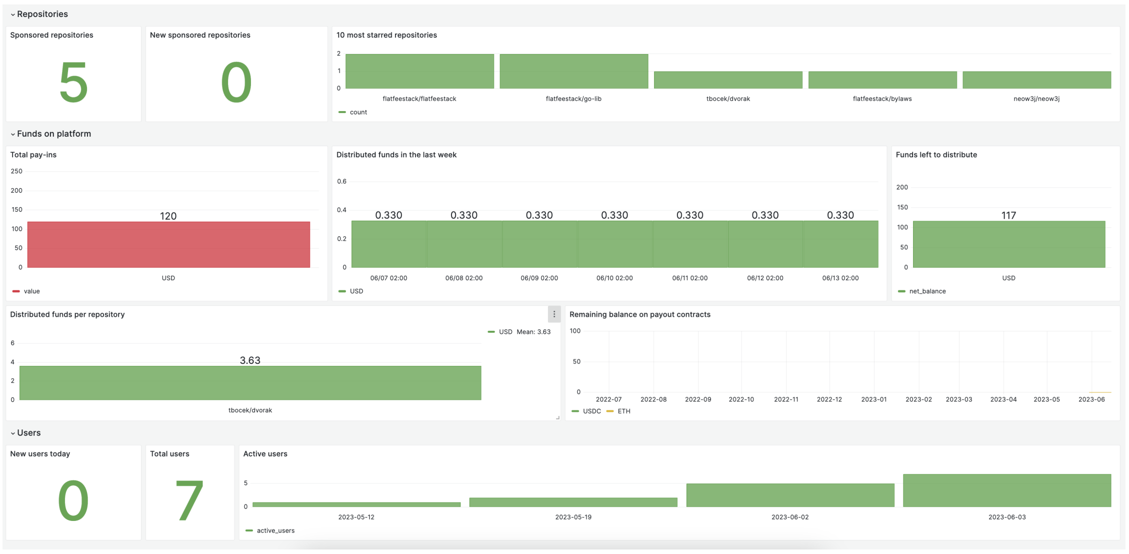


Figure 4.15: FlatFeeStack dashboard.

PostgreSQL Database Dashboard

Figure 4.16 shows the metrics collected by the postgres-exporter. This board is based on a dashboard published by Lucas Estienne [15].

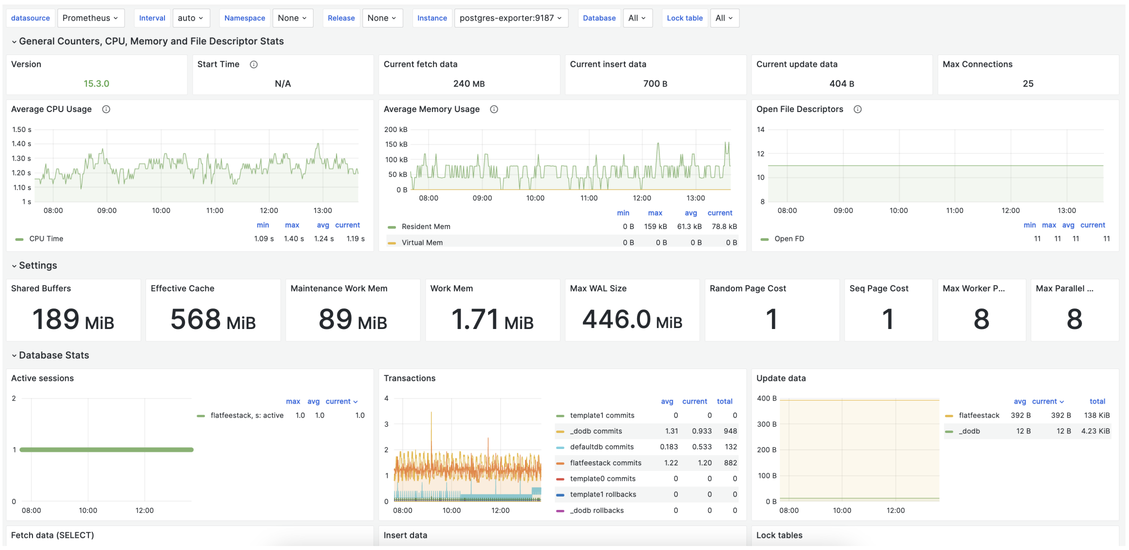

datasource Prometheus
Interval auto
Namespace None
Release None
Instance postgres-exporter:9187
Database All
Lock table All
General Counters, CPU, Memory and File Descriptor Stats
Version 15.3.0
Start Time N/A
Current fetch data 240 MB
Current insert data 700 B
Current update data 404 B
Max Connections 25
Average CPU Usage
CPU Time min 1.09 s max 1.40 s avg 1.24 s current 1.19 s
Average Memory Usage
Resident Mem min 0 B max 159 kB avg 61.3 kB current 78.8 kB
Virtual Mem 0 B 0 B 0 B 0 B
Open File Descriptors
Open FD min 11 max 11 avg 11 current 11
Settings
Shared Buffers 189 MiB
Effective Cache 568 MiB
Maintenance Work Mem 89 MiB
Work Mem 1.71 MiB
Max WAL Size 446.0 MiB
Random Page Cost 1
Seq Page Cost 1
Max Worker P... 8
Max Parallel ... 8
Database Stats
Active sessions
flatfeestack, s: active max 1.0 avg 1.0 current 1.0
Transactions
template1 commits 0 0 0
_dodb commits 1.31 0.933 948
defaultdb commits 0.183 0.533 132
flatfeestack commits 1.22 1.20 882
template0 commits 0 0 0
template1 rollbacks 0 0 0
_dodb rollbacks 0 0 0
Update data
flatfeestack 392 B 392 B 138 KiB
_dodb 12 B 12 B 4.23 KiB
Fetch data (SELECT)
Insert data
Lock tables

Figure 4.16: PostgreSQL database dashboard.

HTTP Dashboard

Figure 4.17 shows the HTTP Dashboard with an example of the backend service. On the left side, the pie chart shows which endpoints are called how many times. The right side shows the average response time of the endpoints.

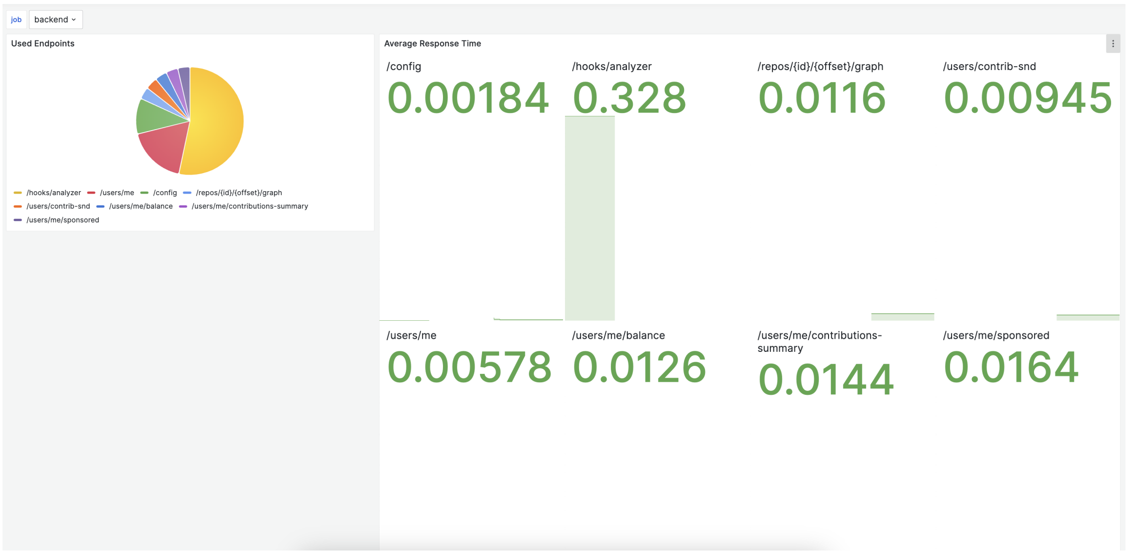


Figure 4.17: HTTP dashboard - backend.

In the dropdown in the top left corner, it is possible to select a different microservice. Figure 4.18 shows the same dashboard for the forum microservice.

The query, shown in Listing 4.15, calculates the average response time. They are shown on the right side of the dashboard 4.18.

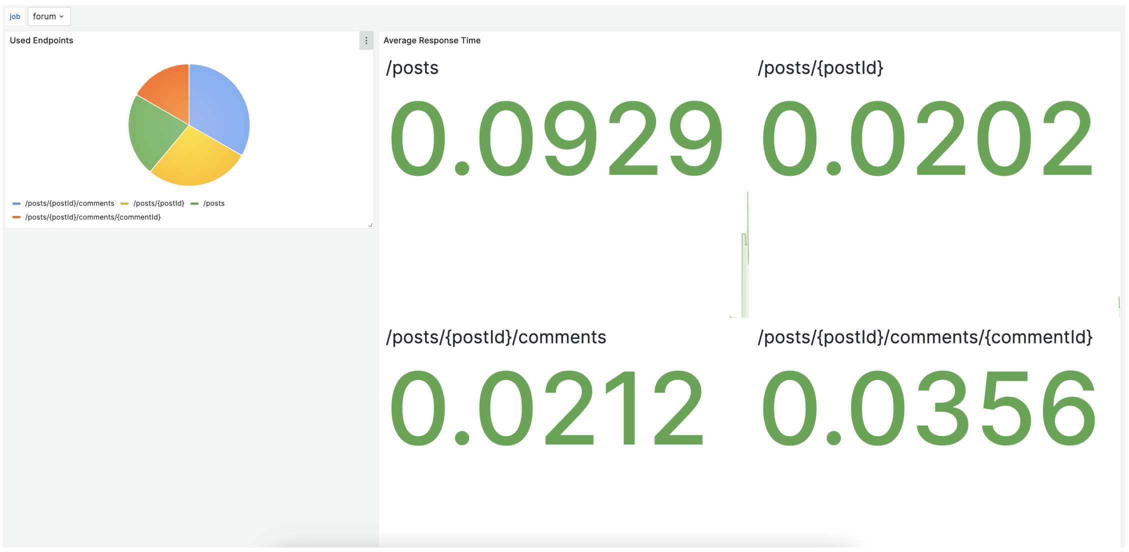


Figure 4.18: HTTP dashboard - forum.

```
sum(http_response_time_seconds_sum{job="$job", path!="/metrics"}) by (path) ↵
/ sum(http_response_time_seconds_count{job="$job", path!="/metrics"}) by ( ↵
path)
```

Listing 4.15: Prometheus query average response time.

It does this by summing the response time for each path and dividing it by the count of requests made to each path. The `$job` variable is provided by Grafana and is used to select the correct microservice.

Go Runtime Dashboard

This dashboard, shown in Figure 4.19, shows the go runtime metrics. This dashboard is based on the dashboard provided by Go Runtime Metrics.

With a switch of the job in the top left corner, it is possible to select a different microservice.

4.6.4 Provisioning

Listing 4.16 shows the YAML file used to provision Prometheus. The YAML file gets mounted into the Docker container, and Prometheus reads it on startup.

```
global:
  scrape_interval: 15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: database
```

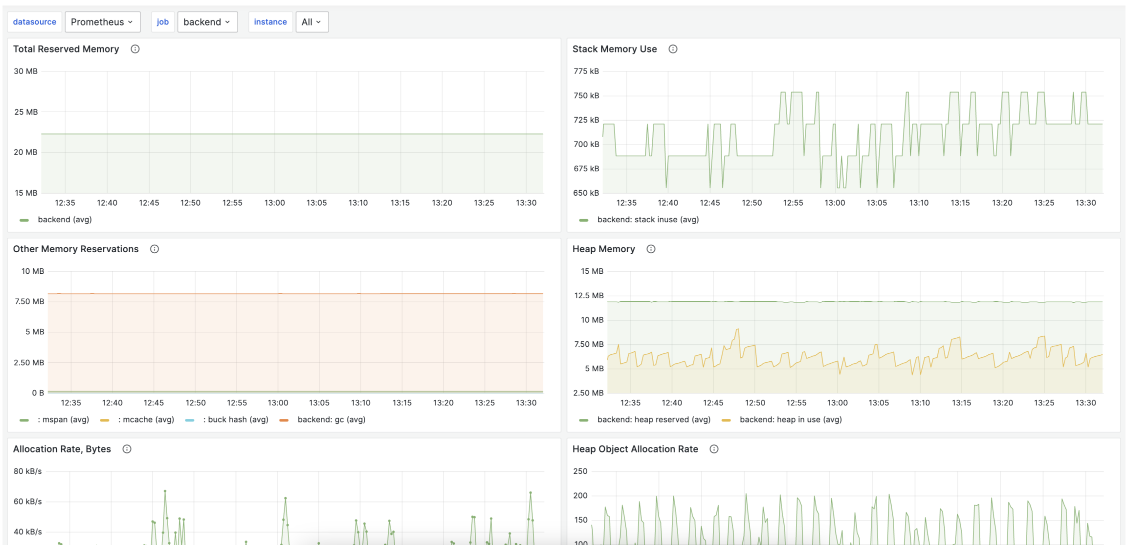


Figure 4.19: Go runtime dashboard - backend.

```yaml
    metrics_path: /metrics
    static_configs:
      - targets: ['postgres-exporter:9187']
  - job_name: analyzer
    metrics_path: /metrics
    static_configs:
      - targets: ["host.docker.internal:9083"]
        labels:
          hostname: 'go-analyzer-server'
          service: 'analyzer'
    ...
```

Listing 4.16: Prometheus configuration file.

As shown in 4.16, a new job is defined for each microservice. One must also define the endpoint target and provide the service name and hostname.

Two different things are needed to provision the Grafana instance. Firstly, the data source must be defined. This project has two data sources: the PostgreSQL database and the Prometheus instance.

Listing 4.17 shows the datasource configuration for the PostgreSQL database. Important to note is the field `uid` because this is used to identify the data source in the dashboard. So if the `uid` is changed, the dashboard must also be updated. The options provided here are for the setup in a local environment.

```yaml
apiVersion: 1

datasources:
  - name: Postgres
    type: postgres
    url: host.docker.internal:5432
    user: grafanareader
    uid: 0YSE7REVz
    secureJsonData:
      password: 'password'
    jsonData:
      database: flatfeestack
      sslmode: 'disable'
      maxIdleConns: 2
      connMaxLifetime: 14400
      postgresVersion: 1500
      timescaledb: false
```

Listing 4.17: Postgres datasource configuration file.

The configuration of the Prometheus data source is shown in Listing 4.18.

```
apiVersion: 1

datasources:
  - name: Prometheus
    type: prometheus
    # Access mode - proxy (server in the UI) or direct (browser in the UI).
    access: proxy
    url: http://prometheus:9090
    uid: 14c03580
```

Listing 4.18: Prometheus datasource configuration file.

The second thing needed is to provide the dashboards. The dashboards are defined in JSON files. These JSON files are mounted into the Docker container, and Grafana reads them on startup. The `path` property is defined for Grafana to find the files inside the container, as shown in Listing 4.19.

```
apiVersion: 1

providers:
  - name: 'FFS'
    folder: 'FlatFeeStack'
    folderUid: 'b8976d22'
    type: file
    disableDeletion: false
    updateIntervalSeconds: 20
    allowUiUpdates: true
    options:
      path: /etc/grafana/provisioning/dashboards
```

Listing 4.19: Grafana dashboards configuration file.

4.6.5 Deployment

It was decided to deploy the monitoring stack on a separate server. The decision was made to deploy the monitoring stack on DigitalOcean, just like the rest of the application, to maintain consistency. A virtual machine is used instead of deploying the stack to the App Platform. On the App Platform, deploying the necessary configuration files proved to be complicated, as App Platform does not allow adding files to a service. The virtual machine is only accessible via SSH and runs Ubuntu 22.10. DigitalOcean calls these virtual machines droplets.

The monitoring stack is deployed with docker-compose. It consists of the following four services:

- Prometheus
- Grafana
- Caddy
- Postgres-Exporter

The Grafana container is configured via environment variables. The full docker-compose service definition is shown in Listing 4.20.

```
grafana:
    image: grafana/grafana:9.5.3
    restart: unless-stopped
    volumes:
      - ${BASE_PATH:-.}/.grafana_data:/var/lib/grafana
      - ${BASE_PATH:-.}/grafana/provisioning:/etc/grafana/provisioning
    environment:
      GF_SECURITY_ADMIN_USER: ${GRAFANA_ADMIN_USER:-admin}
      GF_SECURITY_ADMIN_PASSWORD: ${GRAFANA_ADMIN_PASSWORD:-admin}
      GF_USERS_ALLOW_SIGN_UP: false
      GF_USERS_DISABLE_INITIAL_ADMIN_CREATION: true
    ports:
      - "3001:3000"
```

Listing 4.20: Grafana environment variables.

`GRAFANA_ADMIN_USER` and `GRAFANA_ADMIN_PASSWORD` are only set in the deployment pipeline. If one wants to use the monitoring stack locally, the fallback values are used, which is `admin` for both username and password.

The postgres-exporter also needs an environment variable `DATA_SOURCE_NAME` to define the connection to the database.

To set up the droplet with the needed software, Ansible[1] is used. One playbook is responsible for installing Docker on the droplet and provisioning the YAML files described in section 4.6.4.

The deployment itself is done via GitHub Actions. The action has the following steps:

- Checkout the repository
- Install SSH Keys for the droplet
- Setup Python to run Ansible
- Install Ansible
- Install Docker and provision the YAML files with Ansible
- Start the services with docker-compose

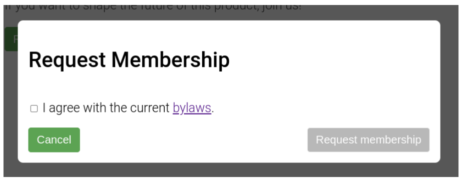
[1]https://www.ansible.com/

4.7 Various Improvements

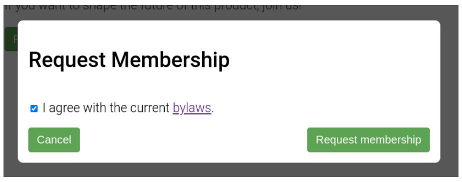
During the Bachelor's thesis, various improvements have been implemented across the different services unrelated to any of the big topics listed earlier in this chapter. This chapter summarises those changes.

4.7.1 Feedback From the Semester Assignment

A couple of feedbacks were received regarding the DAO after the semester assignments were implemented.



(a) Modal before accepting the bylaws.



(b) Modal after accepting the bylaws.

Figure 4.20: Modal asking to accept the bylaws before the request to join the DAO.

- When a member wants to join the DAO, a new checkbox is displayed that they read the bylaws, as seen in Figure 4.20.
- Most modals have been removed from the DAO as accessibility improvement.
- The DAO smart contract received a new property to store the URL to the bylaws. Previously, only the hash was saved. The frontend reads this property to display a link to the current bylaws. The proposal to change the bylaws has been updated to require a URL.

Additionally, a couple of improvements outside the feedback have been added:

- Loading the voting slots and proposals in the frontend has been optimized. Previously, voting slots have been loaded individually without indicating when all have been loaded. As the Ethereum blockchain is relatively fast when loading data, even with more significant amounts, this view has been changed to show a loading indicator when retrieving the data, only showing the view once all data has been fetched.
- The addresses for the DAO contracts are now loaded from the payout service. Previously, those were hardcoded in the frontend, but a more flexible mechanism is needed since the addresses will differ when deploying the contract to different blockchains.
- MetaMask, which the DAO frontend uses to interact with the Ethereum blockchain, exposes two events to the window object notifying when users change the account or connected network in MetaMask. The DAO frontend now listens to those events

and can re-initialize the information that change based on those events. Also, an appropriate message is shown if the chain identifier differs between MetaMask and the contract addresses provided by the payout.

4.7.2 Frontend Improvements

- Svelte checks the site automatically for accessibility issues. Those have been addressed.
- Hot module reload has been implemented to enable faster development.
- Client-side validations were missing in a few places, like when adding a new Git email. Those have been added.
- FlatFeeStack has a page to show all dependencies used in the frontend. This functionality was broken at the start of the thesis but has been restored now.
- A user can review a graph displaying the different contributions for each sponsored repository. This graph did not load correctly but now properly displays contribution weights again, as shown in Figure 4.21.
- The style of the admin page has been aligned with the rest of the site.
- FlatFeeStack now has a favicon.
- Allow users to clear username and remove their profile picture.
- Fix issue were the public badges were only viewable when authenticated, although they should be public.
- Optimized background image to save 80kB.
- Optimized loading of `ChartJS` to save 80kB.

4.7.3 Logging

All components now write their logs in a standard JSON format. This format allows later to better search and filter the logs across different parts when aggregated in a system like Papertrail[1].

4.7.4 No-Reply Addresses

When the analyzer component checks a Git repository, it creates a list of all contributors, including the weight of their contributions. This list previously contained no-reply ad-

[1]https://www.papertrail.com/

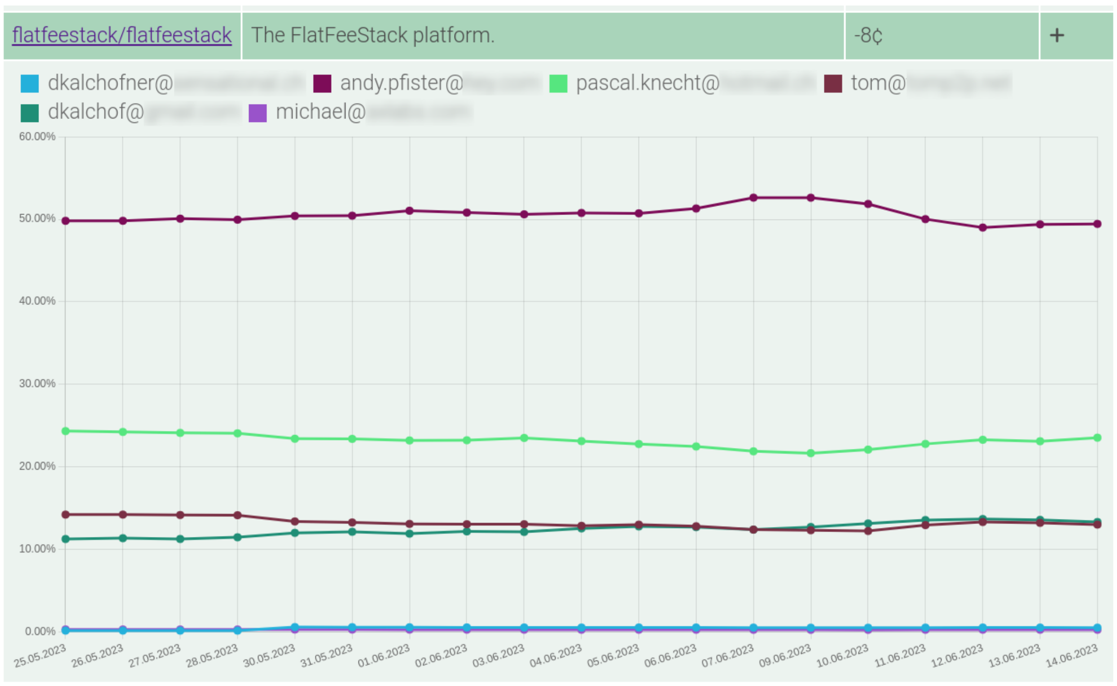


Figure 4.21: Fixed contribution graph.

dresses from GitHub. Commits with those addresses can happen when, e.g., committing something directly from GitHub or merging/squashing pull requests.

FlatFeeStack wants to send an email informing users about received contributions, and those addresses are unreachable, so a filter has been added not to include those addresses in the weight calculation.

4.7.5 OpenAPI Schema

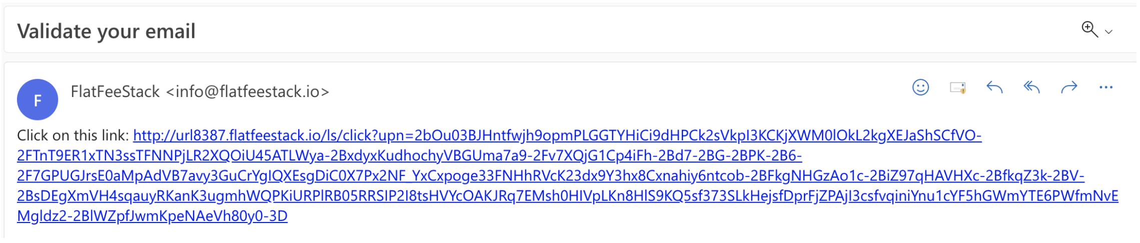
OpenAPI[1] schemas for the backend and forum component have been added. Those schemas are consumed by the frontend to automatically generated TypeScript types. This change uncovered type inconsistencies between the three components, which have also been fixed.

4.7.6 Email Templates

The fastauth and backend components sent various emails informing users about actions on FlatFeeStack, like receiving contributions, confirming the sign-up, or adding a Git email address.

Figure 4.22 shows the previous version of the email template. All these emails were text only, and the combination with the *Click on this link* text made them appear untrustworthy.

[1] `https://swagger.io/specification/`

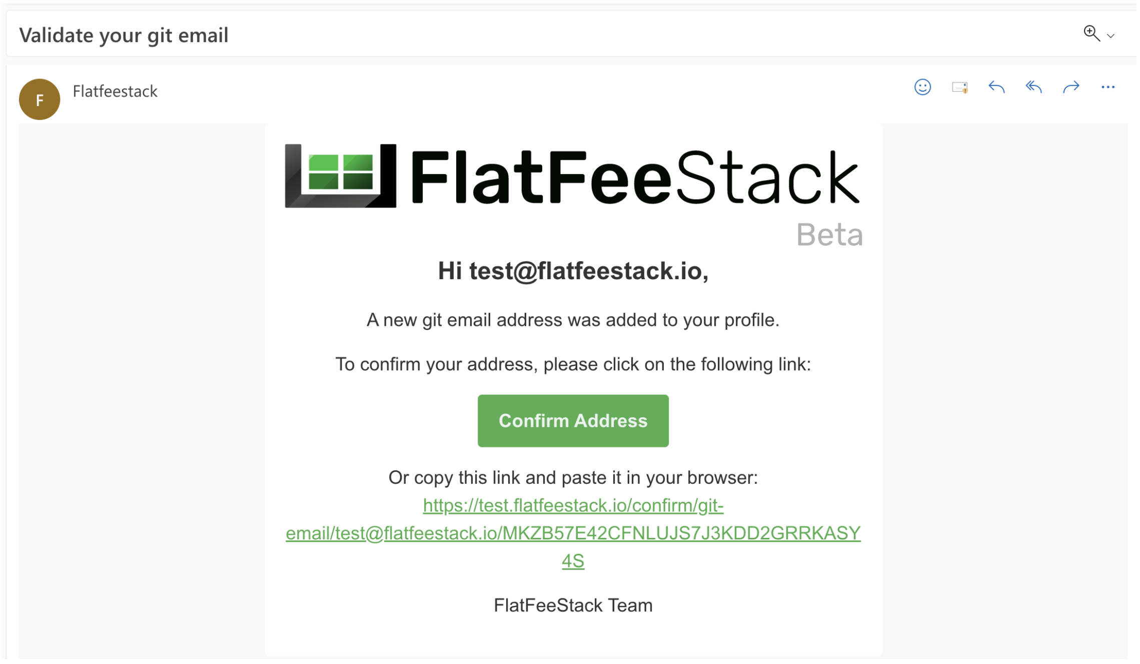

Validate your email

FlatFeeStack <info@flatfeestack.io>

Click on this link: http://url8387.flatfeestack.io/ls/click?upn=2bOu03BJHntfwjh9opmPLGGTYHiCi9dHPCk2sVkpI3KCKjXWM0lOkL2kgXEJaShSCfVO-2FTnT9ER1xTN3ssTFNNPjLR2XQOiU45ATLWya-2BxdyxKudhochyVBGUma7a9-2Fv7XQjG1Cp4iFh-2Bd7-2BG-2BPK-2B6-2F7GPUGJrsE0aMpAdVB7avy3GuCrYglQXEsgDiC0X7Px2NF_YxCxpoge33FNHhRVcK23dx9Y3hx8Cxnahiy6ntcob-2BFkgNHGzAo1c-2BiZ97qHAVHXc-2BfkqZ3k-2BV-2BsDEgXmVH4sqauyRKanK3ugmhWQPKiURPlRB05RRSIP2I8tsHVYcOAKJRq7EMsh0HIVpLKn8HlS9KQ5sf373SLkHejsfDprFjZPAjl3csfvqiniYnu1cYF5hGWmYTE6PWfmNvEMgldz2-2BlWZpfJwmKpeNAeVh80y0-3D

Figure 4.22: Previous email template used for sign up confirmation.

Actual HTML templates were implemented with the FlatFeeStack corporate style to fix these appearance problems. The email-sending mechanism in the backend and the fastauth had to be optimized to allow this. On the one hand, it had to be allowed to send HTML emails and not only plain text; on the other hand, email templates had to be created. The mechanism to allow email templates already existed. Therefore only the correct template path had to be provided.


Validate your git email

Flatfeestack

FlatFeeStack
Beta

Hi test@flatfeestack.io,

A new git email address was added to your profile.

To confirm your address, please click on the following link:

Confirm Address

Or copy this link and paste it in your browser:
https://test.flatfeestack.io/confirm/git-email/test@flatfeestack.io/MKZB57E42CFNLUJS7J3KDD2GRRKASY4S

FlatFeeStack Team

Figure 4.23: New email template used in Git email confirmation.

Figure 4.23 shows the new email template. Corporate styling portrays a professional image and is more inviting.

4.7.7 Improved Error Handling

Error handling existed, but only in some places. Initially, only a generic error message (standard HTTP error dependent on the HTTP status code) got displayed, disregarding whatever the request returned. This disallowed any control over what the user receives.

At first, the assumption was that this came from the backend, respectively, the fastauth services. Through debugging, it was discovered that the Ky library, used for API calls, handles and displays these messages and ignores the actual response from the API.

```
hooks: {
   afterResponse: [
      async (request: Request, options: any, response: Response) => {
      if (response.status !== 200) {
         const body = await response.json();
         if (body.error) {
         throw new Error(body.error);
         }
      }
      },
   ],
},
```

Listing 4.21: Throw actual response error message instead of Ky error message.

Therefore, an `afterResponse` hook (shown in Listing 4.21) had to be implemented to display the actual error message from the request instead of a generic one. In this hook, the HTTP status of the requests gets checked. If it is not 200, the HTTP status for a successful request, then the response should have an error message that can be accessed in the body. It had to be ensured that the backend and fastauth services always return an error message in the body in case of an error, even if it is just a generic one.

The error messages should be mainly done via the backend and fastauth services, as this allows for more control over what is displayed instead of handling this via the frontend. The services only returned error messages in test environments, which was changed to always return error messages. At the start, the error messages were misleading or provided too many potentially risky insights into the application. The decision was made to log every error in detail and return a generic and safe error message for the user.

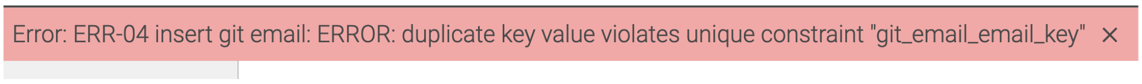

Error: ERR-04 insert git email: ERROR: duplicate key value violates unique constraint "git_email_email_key"

Figure 4.24: Existing git email error message (initial).

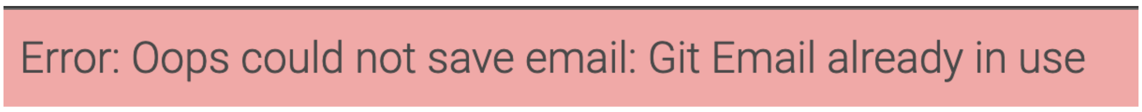

Error: Oops could not save email: Git Email already in use

Figure 4.25: Existing git email error message (new).

Figure 4.24 is an example of the error message before, and Figure 4.25 is the new error message. No database information gets displayed to the user anymore, and the errors are more helpful.

4.7.8 Responsiveness

The UI is relatively simple, displaying various functionalities out-of-the-box correctly on mobile devices. Nevertheless, a multitude of display problems still existed. HTML tables are the main issue. If not set up for mobile display, these easily break the site and cause a vertical overflow of the content, allowing the user to scroll from side to side instead of only up and down. To tackle this overflow caused by HTML tables, a standard HTML/CSS solution got implemented. Add a label data attribute to the individual row columns with the column's name on mobile, hide the table header, display the stacked rows, and add the label data attributed. The usability increased drastically with only this modification applied to all used tables. Additional fixes had to be implemented to ensure all the functionalities were usable on mobile, most of which were minor padding/margin or fixed width issues. As the use of mobile devices increases, it is essential that all the main functionalities can be accessed and used well on smaller view-ports and not only in desktop view.

Figure 4.26 displays the navigation and the invitation table before, and the same view after the changes. The navigation is now displayed correctly, given the current width and the table is displayed in the stacked design, and no vertical scroll is possible anymore.

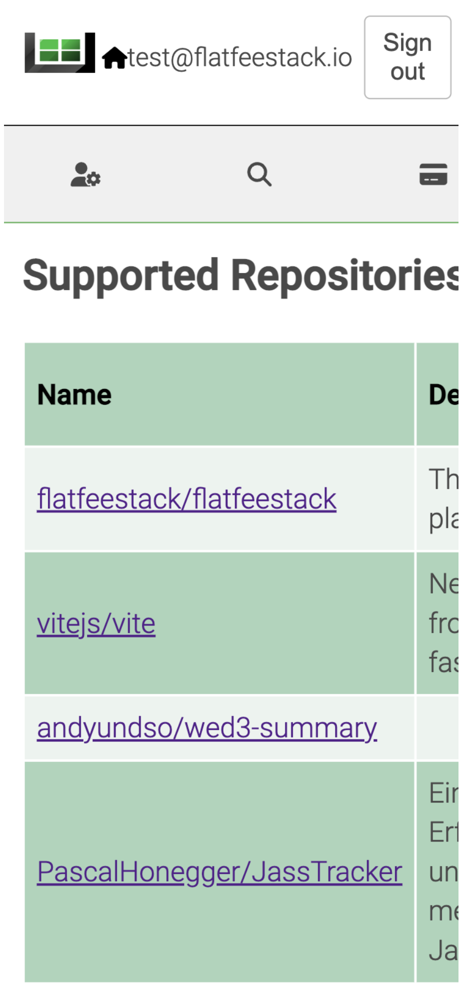
4.7.9 Route Protection

While exploring all functionalities, it was discovered that routes could be directly accessed via the URL. No sensitive data was visible to the unauthenticated user. Nevertheless, the page structure was visible, giving a not very confident impression if no route handling exists. As most routes load data with Svelte's `onMount()` hook, these requests will immediately fail as no auth token is available, displaying an error message.

The first approach, checking the user state in the components and routes, did not work as expected. For example, most routes implement a `Navigation` component around the actual content. Checking in the `Navigation` component would make sense, but because of how the data is loaded, the navigation's `onMount()` function is loaded after the `onMount()` function of the route itself. This allows the page to flicker and be visible for a few seconds to the user before the check takes place.

As no generic middleware exists to handle this authorization, other options were evaluated, such as SvelteKit[1]. This approach was investigated further, as it would allow a de-

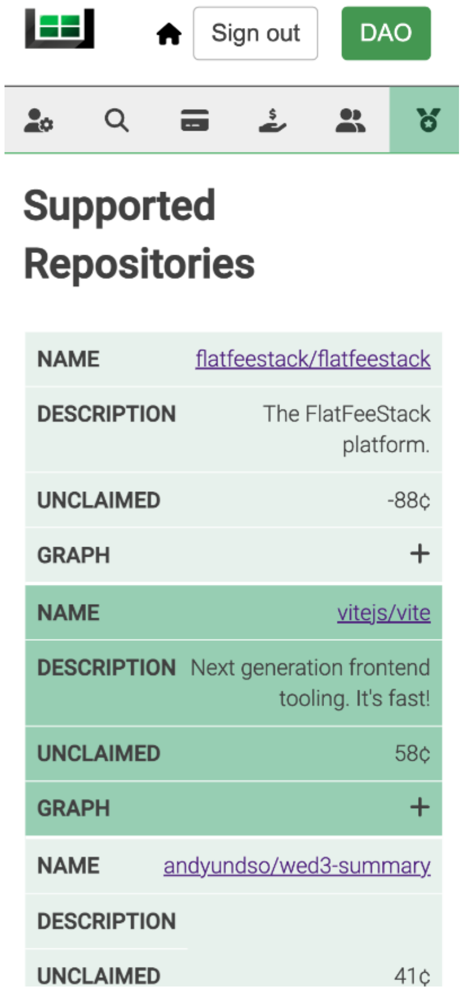
[1]https://kit.svelte.dev/



Supported Repositories

Name	De
flatfeestack/flatfeestack	Th pla
vitejs/vite	Ne frc fas
andyundso/wed3-summary	
PascalHonegger/JassTracker	Eir Erf un me Ja

(a) Badges - supported repositories table display on mobile (initial)



Supported Repositories

NAME	flatfeestack/flatfeestack
DESCRIPTION	The FlatFeeStack platform.
UNCLAIMED	-88¢
GRAPH	+
NAME	vitejs/vite
DESCRIPTION	Next generation frontend tooling. It's fast!
UNCLAIMED	58¢
GRAPH	+
NAME	andyundso/wed3-summary
DESCRIPTION	
UNCLAIMED	41¢

(b) Badges - supported repositories table display on mobile (new).

Figure 4.26: Badges - supported repositories table initial and new.

fault template that all sub-pages use to handle the authentication. After investing some time, this approach was disregarded, as too much time would have to be invested in making the current setup work with SvelteKit. The final solution was to create a custom `PrivateRoute` component and a route middleware component.

```
<PrivateRoute path="/user/admin" admin={true}>
   <Admin />
</PrivateRoute>
```

Listing 4.22: Wrap route with auth guard.

Listing 4.22 shows the wrapping the `Admin` component with a `PrivateRoute`. Furthermore, it is possible to add the `admin` parameter, with which the user role additionally gets verified. This authentication guard ensures that the user is authenticated and, if set, an admin. The routeing middleware then checks if the user is authenticated. This check was done with the user grabbed from the state. The issue with this approach is that the state needs a few milliseconds to load, and the check often came before the user was fully loaded, causing it to think no user was available.

The solution to this was first to check if a user is in the state. If yes, take it; if not, create an asynchronous call to fetch the user from the fastauth service.

```
onMount(async () => {
   try {
      userFromStoreOrAPI = $user.id == undefined
         ? await API.user.get() : $user;
   if (!userFromStoreOrAPI.id) {
      navigate("/login");
   }
   if (admin &&
      (!userFromStoreOrAPI.role || userFromStoreOrAPI.role != "admin")
   ) {
      $error = "Oops you are not allowed to view this resource";
      navigate("/user/search");
   }
   } catch (e) {
      $error = "Log in or create an account to access FlatFeeStack.";
      navigate("/login");
   }
});
```

Listing 4.23: Check if user is available or load async from API.

Listing 4.23 provides the code of how this check is implemented. The `async` call ensures that we have the correct data, either a user or know that the call should be blocked and forwarded to the login page. With the help of this implementation, hard reload and regular reloads, and link clicks work as expected and do not cause false redirecting because of loading issues.

4.7.10 New Staging Deployment

As many components were changed during the preparation of the deployment to DigitalOcean, the existing staging deployment was no longer working. According to discussions with the advisor and external co-examiner, a new deployment has been set up that connects to a MacMini in the Axelra office and deploys an updated version of the FlatFeeStack deployment using docker-compose. The old deployment retrieved the images from the GitHub container registry. This reference was changed to get the images from DigitalOcean.

The new deployment, like the old one, is based on GitHub Actions. For each feature branch, a button is available to deploy to staging directly. For each push on the main branch, a deployment to staging is automatically started.

4.7.11 Extraction of Common Code

The initial architecture analysis (2.3.4) identified a significant presence of duplicated code. This redundancy negatively impacts the system's efficiency, maintainability, and consistency.

Moreover, while working on implementing the forum (4.5) and monitoring endpoints (4.6), it became apparent that there is an urgent requirement for a centralized codebase. This codebase would serve as a repository for standard functionalities that can be leveraged across various services. Consolidating these functionalities enhances code reuse, streamlines development efforts, and promotes uniformity across the system.

To address these concerns, a dedicated GitHub repository[1] has been created to house the extracted code, which now serves as a dependency for the various services.

The repository encompasses several go-modules, each fulfilling a distinct role:

- **auth**
  This module contains the necessary code for validating JWT tokens and an implementation for utilizing BasicAuth.
- **database**
  This module contains the code that establishes connections to the database and executes SQL statements from a file.
- **email**
  The email module encapsulates the code required for sending emails via SendGrid[2] and facilitates the retrieval and application of email templates.
- **environment**
  This module provides the necessary code for reading environment variables, ensuring the availability of default values when needed.
- **prometheus**
  In this module, the code creates a Prometheus registry and registers the default metrics utilized by all services.

[1] `https://github.com/flatfeestack/go-lib`
[2] `https://sendgrid.com/`

The advantages it brings during the build process drove the decision to maintain this codebase as an external repository, separate from the mono repository. By utilizing go-modules, the integration and management of this common code become significantly smoother and more streamlined.

4.8 Test Plan

As the frontend has no automated testing, manual testing is required. A testing plan was derived from the use cases to ensure the functionalities worked as expected.

The test plan, attached in Appendix C.1, goes through each use case and gives concise test cases to ensure the functionalities work as intended. Edge cases and problematic inputs are also included to ensure that different cases and inputs are tested. These include various options for submitting an email address or negative values in numeric fields and more.

Before the application goes live, testing will be done on the staging environment instead of the local development environment. The environment should be as close to the production environment as possible to prevent problems arising from the different setups.

The first testing protocol was created before the initially planned go-live on May 5th. Because there were too many unknowns and open issues, the decision was made to reschedule the go-live. A detailed testing protocol can be found in Appendix C.2.

During the first execution of the test plan, various validation problems were found. For example, `foo@bar` was considered a valid email address, empty values could be submitted in forms or negative values for the number of seats one wants to purchase.

A more significant problem was the possibility that users could block Git email addresses. Once a Git email was added, it could not be added by any other users, although it was not confirmed. This would have allowed blocking contributions for the actual owner of the Git email address. The issue was solved, by allowing the Git email address to be added, if the user has not added it themselves yet (to prevent duplicate Git email addresses) and if the email was not yet confirmed.

Before the next planned go-live on June 2nd, another round of testing was completed, and the detailed testing protocol can be found in Appendix C.3. The mentioned issues could be resolved, and the test was successful. Nevertheless, certain test cases could not be tested for various reasons (f.e. crypto payout is not fully implemented with the new planned version).

After the first test protocol was created and the problems were addressed, more test cases were found and added, ensuring better overall coverage of the functionalities. This explains why certain test cases were not filled out in the first protocol but in the second one.

4.9 Non-Functional Requirements

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

4.9.1 Functionality

- **Acceptance criteria:** Unit Tests are executed for all subsystems in the main branch.
  Result: Partially fulfilled.

  No unit tests were written in the frontend, and verification was conducted manually. This sometimes resulted in code getting merged, which broke functionality in other parts of the application. Unit tests partially cover new code in the backend services (backend, forum, payout).

- **Acceptance criteria:** The application passes all acceptance tests before release.
  Result: Partially fulfilled.

  There are no automated acceptance tests. Nevertheless, the test protocol (4.8) is executed manually before a production release.

- **Acceptance criteria:** User feedback and bug reports are regularly reviewed and addressed to improve functionality.
  Result: Fulfilled.

  At this stage of development, most feedback came from the advisors. These were collected in GitHub Issues[1] and addressed.

- **Acceptance criteria:** Changes to functionality are carefully communicated and documented to avoid confusion or unexpected behavior.
  Result: Fulfilled.

  With the help of pull requests[2] and GitHub Issues[3], changes to functionality are communicated and documented.

4.9.2 Extensibility

- **Acceptance criteria:** Time required to fix bugs or install updates is within the specified limits.
  Result: Fulfilled.

  For library updates, there is the Renovate[4] in place.

[1]https://github.com/flatfeestack/flatfeestack/issues?q=is%3Aissue+
[2]https://github.com/flatfeestack/flatfeestack/pulls?q=is%3Apr+
[3]https://github.com/flatfeestack/flatfeestack/issues?q=is%3Aissue+
[4]https://www.mend.io/renovate/

- **Acceptance criteria:** The application's architecture supports modular design and component reusability.
  Result: Fulfilled.

  This requirement is fulfilled given the microservice architecture of the system. Also, some time was invested in extracting code into a library to reuse in all microservices (4.7.11).

- **Acceptance criteria:** Regular code reviews are conducted to ensure adherence to extensibility best practices.
  Result: Fulfilled.

  This requirement is fulfilled because no code gets merged without a code review and passing tests. Pushes to the main branch without a pull request have been disabled to enforce code reviews.

4.9.3 Robustness

- **Acceptance criteria:** Manual testing with invalid or malicious input.
  Result: Fulfilled.

  The manual testing as outlined in section 4.8 was conducted with invalid input.

- **Acceptance criteria:** Automated testing performed with invalid or malicious input.
  Result: Partially fulfilled.

  The tests were designed to test positive cases and also negative cases. However, there are no special tests for every edge case.

- **Acceptance criteria:** Robust exception handling and error recovery mechanisms are implemented.
  Result: Partially fulfilled.

  Robust exception handling was already implemented. There were some changes made to make it consistent across the microservices.

4.9.4 Code Quality

- **Acceptance criteria:** Static analysis runs with each commit.
  Result: Partially fulfilled.

  No static analysis tools were added, but the ones in place are running during the build process.

- **Acceptance criteria:** code follows established coding conventions and style guidelines.
  Result: Partially fulfilled.

  Prettier[1] is used to format the ETH smart contracts and the frontend code. The code style and conventions are examined during pull requests. However, they were not documented. Neither style nor coding conventions are documented for the NEO smart contract and the Go microservices.

- **Acceptance criteria:** Code complexity is regularly reviewed and minimized where possible.
  Result: Not fulfilled.

  Where possible, the code was simplified following the Boy Scout Rule guidelines. However, any untouched code was not analyzed or refactored.

4.9.5 Maintainability

- **Acceptance criteria:** Deployment pipelines are set up to automatically deploy the application to staging or production environments.
  Result: Fulfilled.

  There are GitHub Actions that can be used to deploy to staging (section 4.7.10) or production (section 4.3.2).

- **Acceptance criteria:** Every commit is built and tested automatically.
  Result: Fulfilled.

  Every commit is built and tested automatically with GitHub Actions. Depending on the service, unit, integration or smoke tests are used.

- **Acceptance criteria:** Environment configurations and dependencies are version controlled and easily reproducible.
  Result: Fulfilled.

  This requirement is fulfilled because all resources are in version control.

4.9.6 Operability

- **Acceptance criteria:** The application provides robust logging and monitoring capabilities, allowing administrators to track and analyze system behavior and performance.
  Result: Fulfilled.

[1]https://prettier.io/

This requirement is fulfilled because all microservices use the same logging framework, and DigitalOcean collects the logs. The monitoring is done with Grafana and Prometheus (4.6).

- **Acceptance criteria:** Proper error handling and informative error messages are implemented to facilitate troubleshooting and debugging processes.
  Result: Fulfilled.

  Every error is logged with a stack trace and a message.

- **Acceptance criteria:** Regular health checks are performed on the application to ensure its operational status.
  Result: Partially Fulfilled.

  Monitoring is configured, but health thresholds are not set. Also, there is no alerting in place (5.2.4).

- **Acceptance criteria:** System metrics are collected and analyzed to detect performance bottlenecks or resource constraints.
  Result: Fulfilled.

  Various system metrics are collected and analyzed with Grafana and Prometheus (4.6).

Chapter 5

Conclusion

This chapter reflects on the outcome of this Bachelor's thesis. The work done is summarised and compared with the original assignment by the advisor. Finally, the chapter explores further developments to enhance or complete existing functionality.

5.1 Summary

This Bachelor's thesis revolved around making FlatFeeStack available to the public. A set of tasks were defined at the start with the advisor, which are needed to complete the platform. The main focus was evaluating a suitable PaaS provider to run a new production instance and deploying the platform afterwards. However, there were also additions to the existing DAO or merging the different microservices into a mono repository. During the whole thesis, existing functionalities were documented and tested, resulting in use-case diagrams and bug fixes/improvements to those functionalities. Finally, parts of the platform were silently launched on June 2nd, and the first open-source projects were supported. The platform is available at flatfeestack.io[1].

At the start of the project, most of the work was research. Existing papers about FlatFeeStack were read, functionalities reverse-engineered, and a suitable development environment for each project member was set up. This work resulted in the list of tasks to do until the go-live, initially scheduled for April 28th.

Evaluating the different PaaS providers was an exciting task, as it highlighted that despite all of them offering the same at the core, user-friendliness and usability differ a lot. DigitalOcean was a solid choice for the needs of FlatFeeStack. A few issues were encountered at the initial deployment, like services not starting at all, but there, customer support was quick and eager to help.

[1]`https://flatfeestack.io/`

The additions to the DAO primarily conducted the addition of the new forum component. Using the API-first approach to develop the new microservice proved to be a good decision, as it allowed the generation of most of the usual repetitive code needed for an HTTP server, and the OpenAPI specification also served as a base to generate parts of the frontend code. Listening to events on the ETH blockchain from the forum component was an exciting challenge, as the documentation on the topic is sparse, and the connection is not straightforward.

As the focus was making FlatFeeStack public, a big emphasis was put on testing the existing functionalities. Some of the time was initially allocated to this task, but it was much more than anticipated. The papers documented different platform parts, so a complete overview was needed. The behavior documented in those papers has been altered as the advisors continued to work on the project by themselves. However, after this thesis, all of FlatFeeStack's functionality is documented and tested, giving an excellent base to continue iterating on the project.

The go-live was postponed several times.

- From April 28th to May 5th, the team underestimated the work for the go-live and needed additional time to polish their work.
- The advisor proposed moving the date from May 5th to May 12th as he was working on a new iteration of the DAO that still needed to be prepared.
- On May 12th, during the initial testing of the test system, it was found that the pay-in using Stripe was not functioning correctly. This issue arose because the advisor made recent code changes that the project team had not validated.
- After May 12th, it was decided to launch the platform on June 2nd, including the necessary payout USDC contract, leaving the DAO out for now as its revision was ongoing.

After all, comparing the final result to the original assignment, most of the original tasks have been fulfilled:

- The new discussion platform is implemented.
- Most of the used libraries for the existing components have been updated. There are a couple of leftovers, like Gradle v8. The reason for this could be accounted to two factors:
    1. Their release was towards the end of the thesis (like hardhat-toolbox v3).
    2. Other dependencies, like neow3j with Gradlev8, did not have compatibility with each other at that time.

- The deployment infrastructure has been built using GitHub Actions and Docker. Deployments to test and production can be done using one click from GitHub.
- The silent go-live was, with a few exceptions, successful.

Inheriting an existing platform with several years of development proved challenging. However, the platform is now available and ready to sponsor awesome open-source projects.

5.2 Future Work

5.2.1 Update DAO to OpenZeppelin v4.9

The FlatFeeStack DAO currently works with ETH block numbers as its clock. As the block time can vary, it is difficult to approximate when an activity like voting on proposals will happen in real-world time. The DAO frontend, implemented in the semester assignment, calculates the difference between the current block number and a given block number, multiplies this with the average block time, and uses this value to display in the frontend [1].

The FlatFeeStack DAO is built upon a modified version of the OpenZeppelin Governor contracts. A new version of those was released on May 23rd, allowing customizing the clock mode [16]. Internally, the Governor contracts now call a clock function which the user can overwrite. That clock function needs to return an integer value, so it would be possible to use Unix timestamps natively in the contract. Using unix timestamps for ETH contracts is typically not recommended, as the miner can influence this value by several seconds. However, as FlatFeeStack DAO has quite long time ranges (one day for voting, several weeks to hand in proposals), a difference of several seconds does not matter. Overall, using unix timestamps in the contract would simplify the DAO and its frontend.

Once the clock mode change has been integrated, the FlatFeeStack DAO will also be ready for launch.

5.2.2 Check Integration with NOWPayments

With the semester assignment of Endres and Lesi, NOWPayments was integrated into FlatFeeStack to allow pay-ins using various cryptocurrencies, right now ETH and NEO GAS [7]. This integration was not tested during this thesis. However, it is expected that some updates need to be applied, as the integration is mostly the same since they handed in their assignment.

5.2.3 Tackle Remaining Major Library Updates

As mentioned in section 5.1, a couple of major library updates still need to be applied in FlatFeeStack.

hardhat-toolbox v3 has been mentioned already. This release was published late into the thesis on June 8th [17]. It makes the Hardhat suite run with ethers.js v6. However, the testing suite for the ETH smart contracts for FlatFeeStack uses the hardhat-upgrades packages from OpenZeppelin, where ethers v6 still needs to be supported [18]. The package by OpenZeppelin enables proxy contracts in the testing suite to match the production environment. Once the OpenZeppelin package is updated, the smart contract testing suite must also be adapted to work with ethers v6 [19].

Gradle is a build tool for Java and is used for the NEO smart contracts, where v8 has been released. Currently, the `neow3j` library does not support it, so this update is postponed.

`@tsconfig/svelte` in the frontend also received a major update. Here, the frontend runs into a type error with the new configuration. The QR code library, used for the pay-in with NOWPayments, provides invalid types which have been ignored by TypeScript so far. Doing this update might be obsolete when revising the integration with NOWPayments (5.2.2).

5.2.4 Activate Alerting in Grafana

Grafana has a feature called *Alerting*. This feature makes sending alerts to channels like Slack, Mail, and others possible. Implementing this feature would be a significant improvement. So the FlatFeeStack team would be notified if a service stops working or the database is unreachable.

5.2.5 Secure Metric Endpoints

Currently, the metric endpoints used for Prometheus are publicly available, which can be a potential security risk. Therefore, making the endpoints only reachable from the Prometheus instance would be better.

5.2.6 Auto-Close Discussions

Figure 3.9 that the forum component automatically closes discussions when the voting for a proposal is done. This functionality was not implemented during the thesis.

Generally, the state of a proposal is a calculated value. Therefore, the forum would need a pulling mechanism to regularly query the state of proposals linked to open discussion, only to close them when the state function returns that the voting is completed.

The required engineering work to properly implemented this mechanism did not fit into the available time for the Bachelor's thesis.

Glossary

BasicAuth Basic authentication is a simple method of user authentication where credentials (username and password) are encoded and sent in the HTTP header for each request. 96

Boy Scout Rule When it comes to programming, the Boy Scout rule suggests that developers should always leave the codebase in a better condition than it was before. This means making small improvements like code refactoring, cleaning, and updating documentation to maintain the overall quality and manageability of the codebase. 101

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. 14, 88

CCHF Centi Franc stablecoin (CCHF) is a coin that maintains a value tied to that of swiss francs. It is a digital currency that uses blockchain technology, allowing for secure and transparent storage, transfer, and transactions with swiss francs in the digital space. 55

Cron In Unix-like operating systems, Cron is a job scheduler that enables users to automate command or script execution at scheduled intervals or predetermined times. 22

CRUD CRUD represents four fundamental operations commonly used when working with data in a software system or database. These operations include creating new records, retrieving existing data, updating or modifying existing records, and deleting unnecessary data. 69

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–iii, 3, 4, 14, 15, 25, 28, 43–45, 52, 59, 71, 76, 88, 103, 104, 106

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. 1

ERC20 ERC20 is a standard protocol for creating and using tokens on Ethereum, making digital assets compatible and easy to integrate within the network. 55, 66

ETH Ethereum, also known as ETH, is a blockchain platform that allows for the creation and execution of smart contracts and decentralized applications (DApps). Transactions and computational services within the network are conducted using its native cryptocurrency, Ether (ETH). ii, 8, 9, 21, 24, 25, 38–42, 44, 55, 59, 60, 66–68, 76, 101, 104, 106, 156, 163, 169

HS256 The HMAC-SHA256 algorithm is used in JSON Web Tokens (JWTs) to create a digital signature using a shared secret key and the SHA-256 hashing algorithm. This ensures that the token's authenticity and integrity are maintained. 31

JWT JSON Web Token (JWT) is a concise and self-contained way of transmitting authentication and authorization data between parties in the form of a JSON object, while maintaining security. 27, 31, 61, 70, 96

KYC Know your customer (KYC) is a process to verify the identity of a person. 14

NEO NEO is a blockchain platform that seeks to revolutionize the economy by offering a platform for creating and executing smart contracts and DApps. It also includes features like digital identity and asset digitization to enhance its functionality. 8, 9, 24, 25, 39–41, 59, 60, 66, 101, 106, 107, 156, 163, 169

PaaS Platform-as-a-Service (PaaS) is a cloud computing model that simplifies the application development process for developers. It provides pre-configured platforms and tools to build, deploy, and manage applications without the need for extensive infrastructure setup and management. PaaS also offers scalability, load balancing, and automatic updates, making it a convenient option for developers. 15, 28, 29, 103, 129

RESTful RESTful (Representational State Transfer) is a popular architectural style for designing networked applications. It prioritizes principles like statelessness and uniform resource representation, allowing for effective communication between clients and servers over the Internet that is scalable and interoperable. 71

SMTP SMTP, which stands for Simple Mail Transfer Protocol, is a commonly used method for sending and receiving email messages across the internet. It ensures dependable delivery between mail servers. 22, 27

SPA A Single Page Application (SPA) is a type of web application designed to fit all its components within a single web page. This design utilizes JavaScript to update the content dynamically without reloading the whole page. As a result, it provides a more seamless and interactive user experience. 21

Tezos Tezos is a blockchain platform that facilitates the development and management of smart contracts and decentralized applications (DApps). It employs a self-modifying protocol that enables stakeholders to vote on suggested improvements and modifications to the network, ensuring its durability and flexibility in the long run. 38, 41

USDC USD Coin (USDC) is a stablecoin that maintains a value tied to that of the US dollar. It is a digital currency that uses blockchain technology, allowing for secure and transparent storage, transfer, and transactions with US dollars in the digital space. 42, 55, 66, 68, 104

WebSocket Websockets allow for real-time, two-way data exchange between a client and server through a single, persistent connection. 22, 50, 52, 76

List of Figures

List of Listings

List of Tables

Bibliography

[1] P. Knecht and A. Pfister, “Flatfeestack as a decentralized autonomous organization,” Februar 2023.

[2] “How to deploy from monorepos,” https://docs.digitalocean.com/products/app-platform/how-to/deploy-from-monorepo/, DigitalOcean, accessed: 2023-03-09.

[3] “Scaling elements,” https://cloud.google.com/appengine/docs/legacy/standard/python/config/appref#scaling_elements, Google Cloud, accessed: 2023-03-18.

[4] “Connect from app engine standard environment,” https://cloud.google.com/sql/docs/postgres/connect-app-engine-standard, Google Cloud, accessed: 2023-03-18.

[5] “Container redeploy,” https://www.virtuozzo.com/application-platform-docs/container-redeploy/?lang=en, Virtuozzo, accessed: 2023-03-18.

[6] “Automatic vertical scaling,” https://www.virtuozzo.com/application-platform-docs/automatic-vertical-scaling/, Virtuozzo, accessed: 2023-03-18.

[7] M. Endres and A. Lesi, “Kryptowährungen als zahlungsmittel bei flatfeestack,” Studenarbeit, Ostschweizer Fachhochschule, April 2022.

[8] J. Brunner, “Payment flow for an open source donation platform,” Master’s thesis, University of Zurich, February 2021.

[9] M. Bucher, “Design and implementation of a fee optimization mechanism in blockchain-based payments for an open source donation platform,” Master’s thesis, University of Zurich, December 2021.

[10] “Simplify neo contract to store a user id instead of address,” https://github.com/flatfeestack/payout-neo-contracts/commit/c2ebfdab24e97c8e2c6fdcef0816174045375969, FlatFeeStack, accessed: 2023-06-14.

[11] "Discourse pricing," https://www.discourse.org/pricing, Discourse, accessed: 2023-06-14.

[12] "How do you merge two git repositories?" https://stackoverflow.com/a/10548919, StackOverflow, accessed: 2023-03-25.

[13] "Introduction to github actions," https://docs.docker.com/build/ci/github-actions/, Docker, accessed: 2023-06-07.

[14] "Usd coin (usdc)," https://etherscan.io/token/0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48, Etherscan, accessed: 2023-06-07.

[15] "Misc grafana dashboards," https://github.com/lstn/misc-grafana-dashboards, Lucas Estienne, accessed: 2023-06-14.

[16] "v4.9.0," https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/releases/tag/v4.9.0, OpenZeppelin, accessed: 2023-06-06.

[17] "Hardhat toolbox v3.0.0: ethers v6, bigints and more!" https://github.com/NomicFoundation/hardhat/releases/tag/%40nomicfoundation/hardhat-toolbox%403.0.0, Nomic Foundation, accessed: 2023-06-13.

[18] "Support ethers@6.0.0," https://github.com/OpenZeppelin/openzeppelin-upgrades/issues/805, OpenZeppelin, accessed: 2023-06-13.

[19] "Migrating from v5," https://docs.ethers.org/v6/migrating/, ethers, accessed: 2023-06-06.