



University of
Zurich^{UZH}

Payment Flow for an Open Source Donation Platform

Jonas Brunner
Zürich, Switzerland
Student ID: 17-709-650

Supervisor: Eder John Scheid, Dr. Thomas Bocek,
Dr. Guilherme Sperb Machado
Date of Submission: February 4, 2021

Abstract

With platforms such as GitHub, it is easier than ever to collaborate together on open source software. It is not surprising that the use of open source software has grown over the last years. Today, from web servers to front-end frameworks, large parts of the web and software generally rely on open source software. However, many open source projects disappear after a short period, often due to lack of funding. While there exist donation platforms that address this problem, the setup is mostly time-consuming and complicated.

This thesis presents a payment flow for a donation platform that minimizes the required configuration steps on both, sponsor- and contributor-side. The platform distributes the donations among the projects and their contributors based on the analysis of a commit evaluation engine. Thereby, the platform follows an optimistic donation approach, where users can sponsor projects without registered contributors. While donations can be made using credit card payments, the payout to contributors is implemented using crypto payments. Besides the architecture and implementation of the payment flow, this thesis evaluates the transaction fees for different payout approaches. It is shown that a "pull payment" approach using an Ethereum smart contract provides a transparent and secure way to pay contributors. Additionally, the straightforward registration process is shown based on two use cases.

Mit Plattformen wie GitHub ist es einfacher denn je, gemeinsam an Open-Source-Software zu arbeiten. Es ist daher nicht überraschend, dass der Einsatz von Open-Source-Software in den letzten Jahren zugenommen hat. Von Webservern bis hin zu Front-End-Frameworks basieren heute grosse Teile des Webs und der Software im Allgemeinen auf Open-Source-Software. Allerdings verschwinden viele Open-Source-Projekte nach kurzer Zeit wieder, oft aufgrund mangelnder Finanzierung. Es gibt zwar Spendenplattformen, die dieses Problem adressieren, aber die Einrichtung ist meist zeitaufwendig und kompliziert. In dieser Arbeit wird ein Zahlungsablauf für eine Spendenplattform vorgestellt, die die notwendigen Konfigurationsschritte sowohl auf Sponsoren- als auch auf Entwickler-Seite minimiert.

Die Plattform verteilt die Spenden auf die Projekte und deren Entwickler basierend auf der Analyse eines “Commit Evaluation Engines”. Dabei verfolgt die Plattform einen optimistischen Spendenansatz, bei dem Benutzer auch Projekte ohne registrierte Entwickler sponsern können. Während Spenden über Kreditkartenzahlungen getätigt werden können, wird die Auszahlung an Entwickler über Krypto-Zahlungen realisiert. Neben der Architektur und Implementierung des Zahlungsflusses werden in dieser Arbeit die Transaktionsgebühren für verschiedene Auszahlungsvarianten evaluiert. Es wird gezeigt, dass ein Pull-PaymentAnsatz unter Verwendung eines Ethereum Smart-Contracts einen transparenten und sicheren Weg zur Auszahlung von Entwicklern bietet. Zusätzlich wird der unkomplizierte Registrierungsprozess anhand von zwei Anwendungsfällen gezeigt.

Acknowledgments

I would like to thank the two initiators of this project Dr. Thomas Bocek and Dr. Guilherme Sperb Machado. Their vision “*Software Engineers should be able to make a living with open source software*” has motivated me to work on this donation solution and learn new technologies and concepts.

I would also like to thank Eder John Scheid, who supported me from the academic side. Finally, I would like to thank Prof. Dr. Burkhard Stiller for giving me the opportunity to be part of this project at the Communication Systems Group at the University of Zurich.

While the past six months have been a rewarding experience for me, I hope this will be even more rewarding for the open source contributors at some point.

Contents

Abstract	i
Acknowledgments	iii
1 Introduction and Motivation	1
1.1 Thesis Contributions	2
1.2 Thesis Outline	2
2 Background	3
2.1 Online Payments	3
2.1.1 Payment Process	3
2.1.2 Chargebacks	4
2.1.3 Comparison of Payment Platforms	4
2.2 Blockchains	5
2.2.1 Ethereum	7
2.2.2 Tezos	8
3 Related Work	9
3.1 Open Source Donation Solutions	9
3.1.1 Open Source Donation Platforms	9
3.1.2 General Donation Platforms	10
3.1.3 Payment Solutions	10
3.2 Comparison of Payment Flows	11
3.2.1 Payment and Payout Options	12

4	Payment Flow	13
4.1	Design	13
4.1.1	Project Fanning	13
4.1.2	Contributor Fanning	15
4.1.3	Payout Aggregation	15
4.1.4	Solution/Donation Flow	16
4.1.5	Architecture	19
4.1.6	Database Design	21
4.2	Implementation	23
4.2.1	Stripe	24
4.2.2	Smart Contract	26
4.2.3	SQL Calculations	28
4.2.4	Scheduler	31
4.2.5	API	33
4.2.6	Frontend	34
4.2.7	Caddy Server	36
5	Evaluation and Discussion	37
5.1	Use Case #1 - Sponsor	37
5.2	Use Case #2 - Developer	40
5.3	Administration	41
5.4	Cost Evaluation	44
5.4.1	Scenario	44
5.4.2	Ethereum	45
5.4.3	Tezos	47
5.5	Discussion	48
6	Summary and Future Work	51
	Bibliography	53

<i>CONTENTS</i>	vii
Abbreviations	57
List of Figures	57
List of Tables	59
A Installation Guidelines	63
A.1 Installation	63
A.1.1 Prerequisites	63
A.1.2 Starting the project	64
B Contents of the CD	65

Chapter 1

Introduction and Motivation

The usage of Open Source Software (OSS) in Swiss companies has constantly grown during the past years [1]. With software, such as Docker or NodeJS, OSS has polished its image and found a way into the daily development workflow of many companies. While many large open source projects emerge from and are backed by large companies, smaller open source projects which are maintained by individuals often lack funding [2]. In the JavaScript environment, Node Package Manager (npm) tries to solve this problem by adding the possibility to add and retrieve funding information directly from the Command-Line Interface (CLI) (*i.e.*, npm fund). However, this only serves as an index, and developers have to set up a donation workflow themselves using a Payment Service Provider (PSP) or a donation platform.

It is hard for developers to assess which solution or platform generates the most donations for their use case, so they often end up creating multiple project pages on different platforms to reach as many sponsors as possible. For example, the frontend framework Vue.js [3] is active on three different donation platforms. On most platforms, project setup is a time-consuming step that usually involves identity verification and creating a detailed description of how donations will be used. However, for certain open source projects, the usage of donations might also differ across donation platforms. In the case of Vue.js [3], the linked Patreon account [4] is a personal account of the founder *Evan You*, while the linked Open Collective account [5] is used to support the other maintainers. The use of multiple donation platforms with different payout mechanisms makes the funding of open source projects a time-consuming and complicated process.

From a sponsor's view, this process is also not satisfying, as often registrations on different donation platforms are needed to sponsor multiple projects. Numerous registrations are not only inconvenient but also increase the inhibition threshold to support lesser-known projects with a small amount of money. Furthermore, if a project uses multiple donation platforms in parallel, the sponsor needs to decide in advance which contributor he wants to support. This gives the sponsor a responsibility he should not have to bear, and it might lead to unfair distribution of donations if the contribution activity of the contributors changes over time.

Based on these observations, the need for an OSS donation platform with fair distribution among the contributors and minimal setup effort for the project owners becomes visible.

Thus, this thesis proposes a payment flow for such a platform taking into account the calculation of the contribution of each developer and the correct payout via cryptocurrency for a higher granularity, lower fees, and a straightforward registration process.

1.1 Thesis Contributions

This thesis contributes to the open source ecosystem in a way, that the design and implementation of a fair and simple to use Open Source Donation Platform (OSDP) is presented. The OSDP platform accepts credit card payments and pays out the contributors in cryptocurrencies. The distribution of donations among contributors is calculated using Falter’s commit evaluation engine [6].

A scalable architecture is defined in a first research-oriented step, which includes components for the credit card payment, cryptocurrency conversion, and the payout. Additionally, the fee structure of different PSPs is evaluated against the OSDP use case and compared to existing solutions. In a second and engineering-oriented step, the payment flow is implemented and integrated with the commit evaluation engine [6]. The implementation includes all payment relevant components as well as a frontend, which allows the simple interaction with the system.

1.2 Thesis Outline

This thesis is organized as follows: Chapter 2 details and compares online payment solutions. In addition, the characteristics of crypto payments are discussed based on two blockchains. Chapter 3 presents related work to this thesis, which includes a comparison of existing donation platforms. Following, Chapter 4 details the payment flow design and implementation. Chapter 5 discusses use-cases and a cost evaluation of the payment flow. Finally, Chapter 6 concludes this thesis and presents future work.

Chapter 2

Background

As this thesis presents a payment flow design for an open source donation platform, an understanding of how online payments work in general is required. Thus, Section 2.1 discusses the different parties involved in online payments. Section 2.2 presents two different blockchains, namely Ethereum and Tezos, and their relation to online payments.

2.1 Online Payments

2.1.1 Payment Process

Accepting online payments is a two-step procedure that includes multiple parties. The first step is the authorization process (*cf.* Figure 2.1) which includes the following stakeholders:

- **Cardholder** (*i.e.*, a customer): A person who wants to buy goods or services using a credit card.
- **Issuing Bank** (Issuer): A bank that issues credit cards to consumers and grants credits.
- **Merchant**: A business that wants to sell goods or services using credit card payments.
- **Acquiring Bank** (Acquirer): The bank of the merchant that enables access to the card network and claims the open balances from the issuer.
- **Payment Gateway**: Technical infrastructure that enables the secure transmission of credit card information to the acquirer.
- **Card Network**: Proprietary networks that enable communication between acquiring and issuing banks (*e.g.*, VisaNet by Visa [7])

The authorization process happens in a matter of seconds and determines if the issuing bank accepts or declines the payment based on the cardholder's credit balance. Depending on bank rules or regulatory requirements, additional authentication (*e.g.*, 3D Secure) can be enforced in the authentication process [8]. When the issuing bank accepts a payment, the message is sent back to the payment gateway so that the merchant can confirm the payment. From a cardholder perspective, the online payment is completed after a successful authentication process.

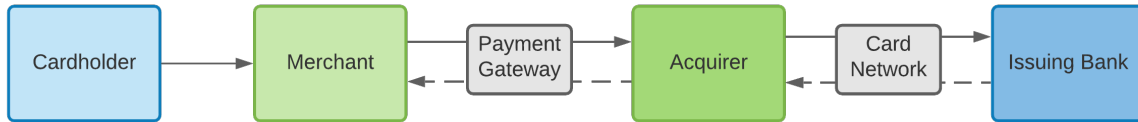


Figure 2.1: Online payment flow according to [8]

The settlement process is the second step in which the merchant receives its money. In contrast to the authorization process, it can take up to a few days. Therefore, some acquirers provide access to the money, before they receive it from the issuing bank [9]. Still, most of them hold back a certain amount as a reserve for possible chargebacks.

2.1.2 Chargebacks

When cardholders disagree with previously made transactions, they can dispute the transaction by filing a chargeback. Approximately 3% of credit card transactions had been disputed in 2019 based on a dataset of 25.1 million transactions [10]. Chargebacks were primarily intended to protect customers, but they have also been misused against merchants, known as chargeback fraud or friendly fraud. This is due to the fact that merchants bear the burden of proof, and customers are not obligated to return a product if a chargeback succeeds.

From a merchant perspective, chargebacks are always associated with additional costs because companies such as Stripe charge a fixed fee per disputed payment without considering the outcome [11]. In this sense, additional authentication using 3D Secure is interesting for merchants as it shifts the liability of chargebacks from the merchant to the issuing bank [12].

2.1.3 Comparison of Payment Platforms

The notion of payment platforms is rather general because payment service providers differ in their scope of functions. While some providers serve only as payment gateway, others also take an acquiring bank's role, thus combining multiple services into a single fee. Additionally, some payment providers focus on pre-made checkout pages, whereas others are more tailored to developers by providing Software Development Kits (SDKs) in

Table 2.1: Payment Scenarios

Scenarios	# Monthly Paying	# Yearly Paying	Int. Payments	Monthly Price
Scenario 1	200	50	90%	CHF 10.00
Scenario 2	20'000	10'000	95%	CHF 10.00

different languages. Thus, comparing different platforms is difficult as it depends heavily on the use case and the type of payments.

The discrepancy of costs between different use cases can also be seen when comparing three payment providers based on two different subscription scenarios (*cf.* Table 2.1).

Table 2.2 presents the cost for three payment providers, namely Stripe [13], Payrex [14] and Datatrans [15]. Payrex and Datatrans are both swiss companies, while Stripe is active in 42 countries. Most providers split their fee into two parts, (*i*) A fixed amount per transaction and (*ii*) a percentage of the transaction amount. Some of the providers also charge a fixed monthly fee independent of the transaction amount. Compared to the other two providers, Datatrans only serves as a payment gateway, which means that an additional contract with an acquiring bank is needed. Thus, the variable percentage fee is only a rough estimation.

It can be seen that Stripe is the cheapest provider for use cases with lower sales volumes (Scenario 1) but the most expensive for large volumes. On the other hand, the additional costs for international payments drive a large part of the costs of Payrex and Datatrans. In general, flat-fee solutions such as Stripe are good for starting businesses as they are easily set up. With growing sales volume, it can be cheaper to choose a flexible solution such as Datatrans, as the separation of payment gateway and acquiring bank offers a new basis for negotiation.

One problem that occurs across all payment providers is the unequal fees for small transactions. Due to the fact that payment providers usually charge a fixed amount between CHF 0.20 and CHF 0.30 for a transaction (independent of the transaction amount), it can quickly add up for low volume payments (*e.g.*, under CHF 10.-). For a credit card payment of CHF 5.-, transaction fees can add up to 9% of the transaction amount (in the case of Stripe). However, the challenge of micropayments can not only be solved by choosing a suitable payment service but by aggregating multiple small transactions into one (*e.g.*, paying a monthly subscription yearly). The treatment of unequal fees for small payments differs across existing donation platforms. Some platforms ask their users to pay for multiple weeks in advance [16], other platforms apply their fees to the received amount after transaction fees [17].

2.2 Blockchains

Blockchains are decentralized databases that prevent the modification of previously added data. By relying on cryptographic hash functions, each block in a blockchain is connected

Table 2.2: Payment service provider comparison. Values marked with “*” symbol are estimated.

Payment Provider	Stripe	Payrexx Pro	Payrexx Plus	Datatrans
Scope	Gateway, Acquirer	Gateway, Acquirer	Gateway, Acquirer	Gateway
Focus	Developers	Pre-made Checkout Pages	Pre-made Checkout Pages	Developers
Fix Costs	CHF 0.00	CHF 49.00	CHF 29.00	CHF 125.00
Cost / Transaction	CHF 0.29	CHF 0.20	CHF 0.20	CHF 0.29
%/ Transaction	2.9%	1.7%	2.2%	*1.5%
%/ Int. Transaction	2.9%	3.2%	3.7%	*2.5%
Total scenario 1 / year	CHF 1'407.-	CHF 1'810.-	CHF 1'690.-	CHF 2'787.-
Total scenario 2 / year	CHF 142'100.-	CHF 125'588.-	CHF 137'348.-	CHF 132'800.-

to its previous block, which enables immutability. The individual transactions in a block are verified by participating computers in the network, the so-called “Miners.” The security of a blockchain increases the more computers participate in the network. This is because, in the case of Proof-of-Work (PoW), it requires more than half of the network’s computational power to alter the blockchain in a dishonest way[18]. While the blockchain technology enables many different use cases [19, 20] it is most known for its digital currencies built on top of it (*e.g.*, Bitcoin).

Compared to traditional payment methods such as credit card payments, blockchain technology enables decentralized payments. Crypto payments remove other intermediaries (*e.g.*, banks) and provide a higher level of transparency as each transaction is public and verifiable by everyone. However, without a central authority, trust is a problem. By requiring proof before new records (*e.g.*, payments) can be added to the blockchain, the problem of trust can be resolved. This mechanism is called “consensus algorithm,” and there exist different approaches for decentralized networks to find consensus on a valid copy of the blockchain:

- *Proof-of-Work (PoW)* is based on problems that are hard to solve but easy to verify. In the context of blockchain, the proof of work is usually provided by calculating a hash that matches a certain pattern. Calculating a hash with a certain pattern is a trial and error task. Thus, to participate in the consensus algorithm, a miner must invest computational power to calculate as many hashes as possible. Because

computational power is always associated with costs, the PoW consensus algorithm can be used to secure a blockchain. However, PoW has also been criticized for its large energy consumption [21].

- *Proof-of-Stake (PoS)* has emerged as an alternative to the energy-consuming PoW algorithm. Instead of investing computational power, miners have to stake coins. The staked coins are frozen for a specific time and serve as a security guaranty if a miner acts fraudulently. While each blockchain has its own rules regarding the PoS, one common variant is that the number of stakes increases the probability of getting chosen to mine the next block.

While there does not exist a central authority that collects fees for crypto payments, the miners who secure the network need to be incentivized to invest resources. This means that crypto payments, similar to traditional payments, have associated fees. In the following sections, two different blockchains are presented regarding their use case for crypto payments.

2.2.1 Ethereum

In terms of market capitalization, Ethereum is the second most popular blockchain with its cryptocurrency called “Ether”. It has a built-in Turing-complete programming language, which enables smart contracts and decentralized applications [22]. The original implementation of Ethereum uses a PoW consensus algorithm. With the announced Ethereum 2.0 update [23], the consensus algorithm will shift to PoS during the next few years. Furthermore, the number of transactions will be significantly increased by relying on shard chains. While those changes will influence transaction fees in the future, this thesis is based on the Ethereum 1.0 protocol [22].

Fees

Ethereum uses the so-called “gas” as compensation for the miners who validate the transactions. Each transaction defines a *gas price* and a *gas limit*. On the one hand, the gas limit defines the upper limit of gas one is willing to pay for a transaction. Usually, the gas limit requires little attention, as many wallets and libraries provide the functionality to estimate the required amount of gas. On the other hand, the gas price defines how much one is willing to pay for a single unit of gas. In contrast to the gas limit, the gas price is constantly fluctuating. Normally, gas price increases when the network is congested, as miners include those transactions in a block that bring the greatest reward (*i.e.*, with the highest value for $\text{gas limit} \times \text{gas price}$).

Smart Contracts

Smart contracts are applications that are deployed to the blockchain and run on the Ethereum Virtual Machine (EVM). Similar to all transactions on the blockchain, smart

contracts are accessible by anyone and cannot be altered after they have been deployed. Because smart contracts can execute transactions automatically if certain requirements are fulfilled, they enable applications such as token systems or smart *multisignature* escrow [22].

2.2.2 Tezos

Tezos is a smart contract platform with its own stack-based smart contract language, called *Michelson*. Tezos has an on-chain-governance, which means that coin holders can vote on proposed changes of the Tezos protocol [24]. This replaces the fork-based governance that is known from popular blockchain forks, such as Bitcoin Cash. The consensus algorithm of the Tezos blockchain is based on PoS, and the validators that keep the network secure are called “Bakers”. With a market capitalization of about 1.5 billion USD as of December 2020 [25], Tezos is not among the top cryptocurrencies that are commonly used for payments. However, it is an emerging platform, especially in the area of Decentralized-Finance (DeFi). With the recently approved Delphi upgrade [26], Tezos lowers the gas consumption by about 75%. Thus, enabling further use cases in DeFi.

Fees

Tezos uses the notion of “gas” and “gas limit” to set a hard cap on the number of steps that a program is allowed to run for [24]. This solves the problem of long-running programs that only provide an incentive for one miner but not for the others who also have to verify it. The *gas* does not have a directly associated price such as Ethereum does. Instead, fees for a transaction must fulfill the condition presented in Equation 2.1 with *opbytes* being the byte size of the serialized operation. The default unit for fees in Tezos is Mutez with $1\text{ tez} = 1'000'000\text{ mutez}$.

$$\text{minfees} \geq 100 + (\text{gas} * 0.1) + (\text{opbytes} * 1) \quad (2.1)$$

Smart Contracts

Tezos defines smart contracts as stateful accounts with executable code [24]. Each contract has an associated manager who can spend the funds of the contract if the contract is defined as spendable. Tezos also introduces a storage fee for each byte increase in the storage of a contract. However, the fee does not go to any bakers, but it is “burned” (*i.e.*, destroyed). Tezos smart contracts also support formal verification by mathematically proving properties. This technique is more powerful to avoid bugs than the commonly known unit tests.

Chapter 3

Related Work

Even before GitHub [27], one of the major OSS hosting platforms, was launched [28], Bonaccorsi and Rossi mentioned the danger of “altruistic individuals” and “selfish firms” in open source development [29]. As of now, the open source ecosystem has grown rapidly with over 40 Million public repositories on GitHub only [30]. Hence, it is not surprising that platforms have emerged in the meantime that try to incentivize open source developers to contribute.

While financial rewards are the most obvious incentive, [31] shows that not all developers are willing to accept rewards, especially when they feel that the financial support could take influence on the innovation process. A recent study by Overney [32] shows that only a small fraction (0.04%) of public repositories on GitHub ask for donations. Still, the demand for open source donations has increased significantly during the past three years, with PayPal [33] being the top requested donation method across all `Readme.md` files on GitHub [32]. This is especially interesting because PayPal is a general digital payment service that is not specifically tailored towards open source funding.

The following sections present and discuss different solutions that are used to collect donations in open source development.

3.1 Open Source Donation Solutions

There exist different solutions to collect donations and to build a sustainable income as an open source contributor. Some platforms are designed for a specific use case while other platforms address a broader target group often including creators and artists. Therefore, the different solutions are grouped into three categories in the following subsections, namely *(i)* Open Source Donation, *(ii)* General Donation, and *(iii)* Payment Solutions.

3.1.1 Open Source Donation Platforms

Open Collective[34] is a fully transparent donation platform based on monthly recurring payments. While many of the registered “collectives” represent open source projects, the

platform can also be used for other non-profit projects. Each collective has a publicly visible balance from which money can be withdrawn by submitting an expense. The expenses (*i.e.*, the money to be distributed to contributors) are approved by administrators of the collective and are publicly visible on the collective page.

Issuehunt [35] is a platform which is specifically tailored to open source software donations. Open issues can be funded (*i.e.*, creating a bounty) to incentivize more developers to work on a task. The developer who resolves an issue first gets the funded amount.

Tidelift [36] provides a managed open source solution for software development teams. With the cheapest subscription being 1500\$ per month and the direct integration into the CI workflow, the platform is built for enterprise-level customers. Tidelift brings open source maintainers and professional users together in a way that users can rely on maintained software, and maintainers are being compensated for their work.

3.1.2 General Donation Platforms

Patreon [37] is a well-established platform on which content creators of various kinds can generate sustainable income. Although the platform does not explicitly promote the open source donation use-case, there are still many OSS contributors relying on Patreon [32]. While music and video creators often use Patreon as a paywall for additional or early-access content, OSS contributors most of the time only offer small goodies for considerable higher prices such as access to discord channels or a logo placement in the `Readme.md` file of the project repository [4, 38].

Flattr [39] is another donation platform aiming at creators of any kind. What sets the platform apart from others is, that a flat-rate sponsorship amount can also be divided among different creators. This allows users to support many different creators without overstraining the wallet. According to [32], Flattr is the third most used donation platform among open source developers.

Liberapay [40] is a non-profit open source platform for recurring donations to creators of various kinds. By directly integrating with payment providers such as Stripe [13] or PayPal [33], the money is directly transferred to the creator's payment service provider account.

Ko-Fi [41] is a platform that uses the notion of "Buying someone a coffee" to motivate people to donate to creators of all kinds. Similar to Liberapay, Ko-Fi also directly integrates with the creator's Stripe [13] or PayPal [33] account and does not charge any platform fees. Ko-Fi is financed through a "Gold" membership, which allows creators to receive recurring payments and to gain analytics insights.

3.1.3 Payment Solutions

GitHub Sponsors [42] is a donation functionality which is directly included in the GitHub platform [27]. With GitHub Sponsors, users can sponsor individuals and organizations

with recurring payments. As of now, GitHub Sponsors is still in beta, and GitHub itself covers the payment fees. It can be assumed that GitHub will introduce a fee in the future.

PayPal [33] is a well-established payment service provider that is available in more than 200 countries. Having a PayPal account, receiving money is quite straightforward, *e.g.*, by creating a *paypal.me* link or using a donation button.

3.2 Comparison of Payment Flows

While the section above categorized the existing platforms based on their use-case, this section compares existing solutions regarding their payment flow.

Donations can reach OSS developers either directly or indirectly, as shown in Figure 3.1. Both indirect and direct payments require a PSP which provides access to different credit card and payment networks, such as VisaNet by Visa [7]. Indirect payments are first transferred to the bank account of the donation platform. Therefore the developer receives the money from the platform itself. With direct payments, developers receive the sponsored amount directly from the PSP to a connected bank account. Based on the existing solutions presented in chapter 3.1, direct payments can be split up into two groups, resulting in three different payment flows (Figure 3.1):

1. **Direct Payments:** By directly using a payment service provider to accept donations, other intermediaries can be omitted. This means that a developer can, for example, directly link his PayPal account in the `Readme.md` file of a repository to accept donations. Even though this minimizes the fees, it may exclude companies or individuals that need a receipt to manage their expenses.
2. **Direct Payments using a project index:** Platforms that only facilitate direct payments between the sponsor and the developer are denoted as project indices. This extra layer can provide additional functionality, such as repeating payments, and it increases the visibility of the project to attract more sponsors. While the fees are kept low, an additional registration step on the project index platform is required.
3. **Indirect Payments using a donation platform:** Indirect payments provide the greatest flexibility regarding the payout and the interaction between sponsor and developer. Donation platforms can, for example, add functionality to distribute money between different developers of a project or provide extra content/rewards to sponsors. The increased flexibility comes at the price of a platform fee and sometimes also an additional payout fee.

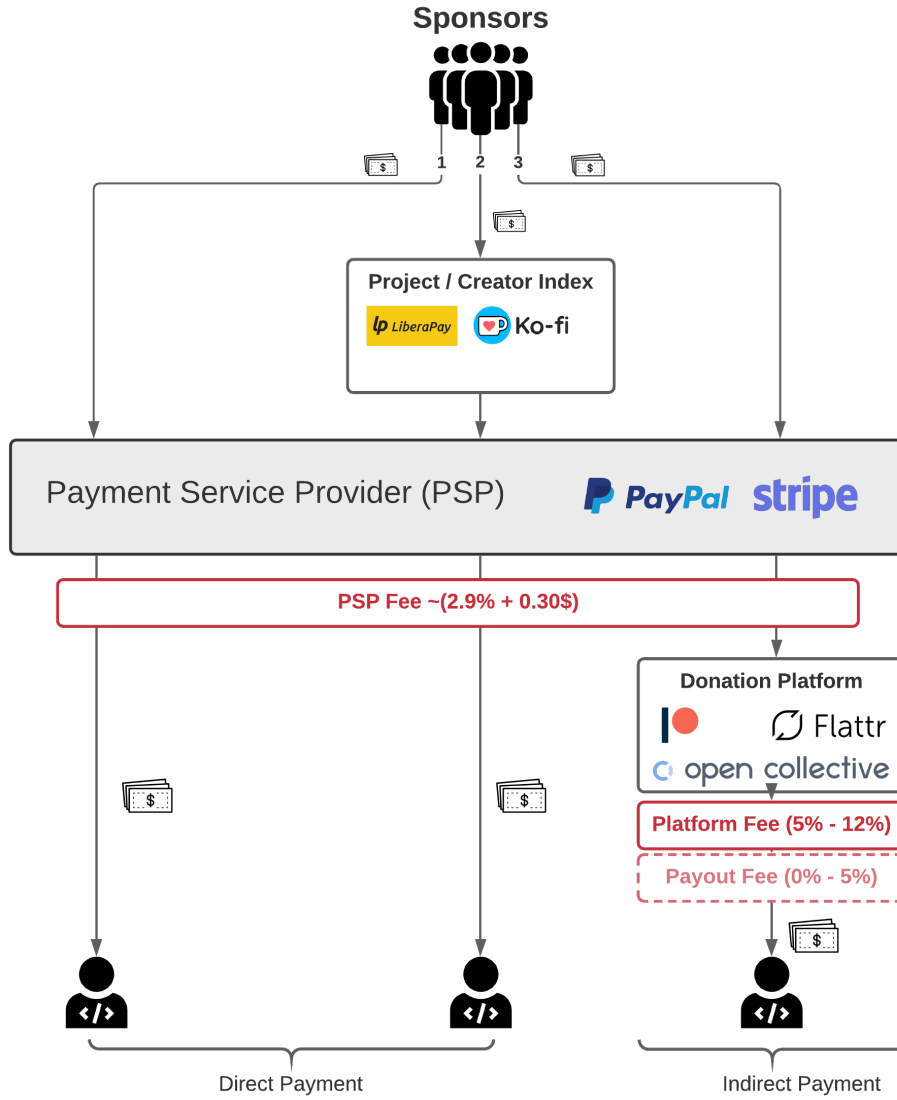


Figure 3.1: Payment flow comparison of existing platforms

3.2.1 Payment and Payout Options

The above-mentioned platforms accept credit card payments and often directly integrate with the contributor’s payment service provider, such as PayPal [33]. This strategy is very common and requires multiple registrations for contributors, namely (i) project page setup on the donation platform, (ii) registration on the PSP, including an identity verification (KYC). Other platforms that use their own PSP are still obligated to perform a KYC in order “to fight fraud, money laundering and the financing of terrorism [43].” Even though there are different payout strategies, they all require an identity verification, which is a bureaucratic and time-consuming step.

Furthermore, neither platform allows the payment or the payout using cryptocurrencies, such as Ether or Tezos. The introduction of such currencies allows for faster and cheaper payments compared to traditional financial services.

Chapter 4

Payment Flow

This chapter presents the payment flow for an open source donation platform where users can donate using their credit card and contributors (*i.e.*, developers) receive their money in cryptocurrencies. Section 4.1 presents the design of the payment flow and Section 4.2 details its implementation.

4.1 Design

The platform's payment flow is designed so that the sponsored money is automatically distributed among the project contributors. This means that the support action (*i.e.*, adding a project to the sponsor list) happens on a project level, but the payout is done on an individual level to each contributor. Thus, the money fans out on two stages, as shown in Figure 4.1.

The fees (PSP fee and platform fee) that are generated during the payment flow are marked in red (Figure 4.1). While the PSP Fees depend on the chosen provider's fee structure (*cf.* Section 2.1.3), the platform fee can be freely chosen to cover the operating costs of the platform.

4.1.1 Project Fanning

The fanning out of money on the project level is caused by the following requirements of the platform:

- The user donates a flat amount per month.
- The user can support as many projects as he/she wants and is able to change them at any time.
- The donation amount of a user is equally divided among his/her sponsored projects.

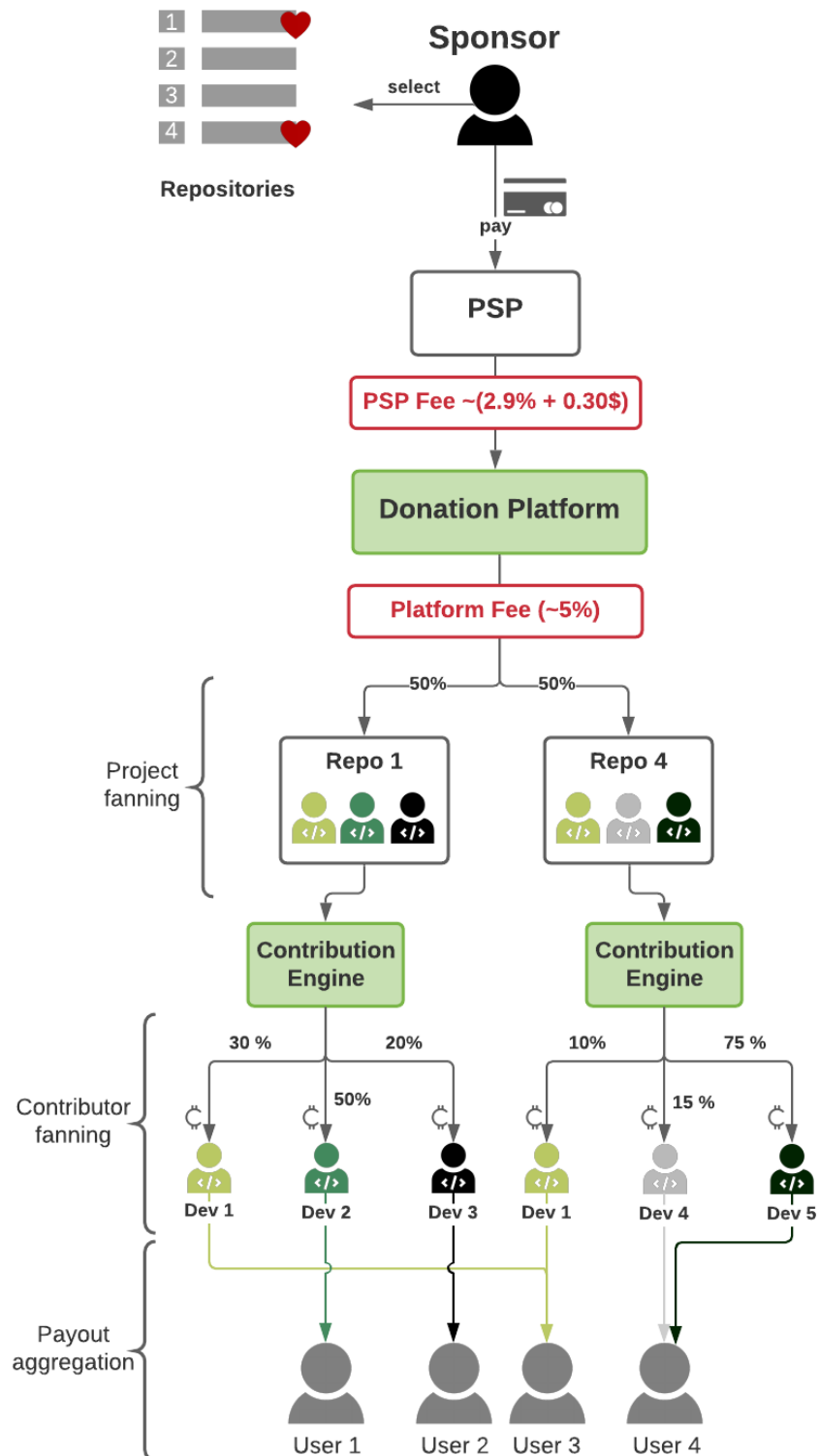


Figure 4.1: Payment Flow

With the requirement that a user can add and remove projects from his/her sponsor list at any time, the equal division of a user's sponsor amount (*e.g.*, at the end of the month) is not fair in every case. This means that a project sponsored on the last day of the month should not receive the same amount as a project that has already been on the sponsor list for the entire month. Thus, the project balances are calculated on a daily basis, which reflects sponsoring- and unsponsoring-events in a fairer way. The daily balance of a project can be represented as shown in Equation 4.1, where *ProjectSponsors* denotes the set of all sponsors of a specific project, *S_{amount}* denotes the monthly donation amount of a sponsor, and *d* denotes the number of days in a month.

$$DailyProjectBalance = \sum_{S \in ProjectSponsors} \frac{S_{amount}}{d \cdot |S_{sponsorList}|} \quad (4.1)$$

4.1.2 Contributor Fanning

The fanning out of money on the contributor level stems from the fact that the payout happens on an individual level and not on a project level. Thus, each contributor receives a share of the daily project balance. A contributor's weight is calculated using the commit evaluation engine proposed in [6] which computes a score based on git contributions. The weight calculations are based on a window of three months, and the weights are updated monthly.

A window of multiple weeks avoids the unfair distribution on less active repositories. For example, with a weekly window, one small update to a `Readme.md` file (with no other commits in this week) could generate an equal reward as a large pull request the week before. On the other hand, a large window also means that the payout to a contributor is spread over multiple weeks. In an extreme case, this means that the reward for one single commit is paid out in chunks over the duration of three months. The payout schedule is further discussed in Section 4.1.4

4.1.3 Payout Aggregation

A project contributor is identified by its git email address. Because a user can have multiple git email addresses configured (*e.g.*, a work-email on the office computer and another email on the personal laptop), there exists a one-to-many relationship between users and contributors. Furthermore, a single contributor can be involved in multiple projects and generate income from multiple sources. Therefore, a payout aggregation is needed, which maps the rewards of contributors to the platform's registered users.

The fanning and aggregation of payments are based on calculations over different time intervals. Section 4.1.4 discusses the design decision regarding the payment flow. The architecture of a system that enables this payment flow is shown in Section 4.1.5. Section 4.1.6 describes the entity-relationship model of the database to facilitate the payment flow.

4.1.4 Solution/Donation Flow

This section organizes the payment flow into three parts and details how the fanning out and aggregation of payments (*cf.* Figure 4.1) can be incorporated.

Payment

Accepting credit card payments requires integration with a PSP. The fees of different PSPs have already been discussed in Section 2.1, and it has been shown that fees are often disproportionately high for small payments. Having a large part of the payment cut off for different fees makes a donation platform unattractive, as its primary goal is to facilitate payments between sponsors and contributors. Thus, to reduce fees, sponsors can choose the payment interval and are incentivized to pay their monthly fee in larger chunks (*e.g.*, quarterly or monthly).

While this strategy helps to reduce the payment fees, it adds complexity because each payment interval is associated with different fees (*cf.* Table 4.1). This means that the received net amount from a sponsor depends on the chosen payment interval. Even though all sponsors pay the same amount at the end of the year, only the received net amount can be used to determine how much money can be distributed. Consequently, further integration with a payment service provider is needed to store the received net amount of each payment. This prevents the distribution of money that was never received.

Table 4.1: Fees for different payment intervals

Payment interval	Fees
CHF 10.- monthly	$(10 * 0.029 + 0.3) * 12 = \text{CHF } 7.08$
CHF 30.- quarterly	$(30 * 0.029 + 0.3) * 4 = \text{CHF } 4.68$
CHF 120.- yearly	$120 * 0.029 + 0.3 = \text{CHF } 3.78$

Distribution

The distribution of money to projects and their individual contributors is a crucial part of the payment flow. As previously discussed, the balances of projects are calculated daily to reflect sponsoring- and unsponsoring events as closely as possible. This enables the daily calculation of contributor balances. While these calculations occur daily, the contribution weights are updated monthly. This is caused by rate-limits of git hosting services, such as GitHub [27] and the fact that analyzing a repository with thousands of commits is a resource-intensive task [6]. The monthly updates of the contribution analysis are executed in a rolling procedure and not, for example, at the end of each month. When a project is sponsored for the first time, an analysis of the repository is triggered. Subsequent daily checks ensure that the analysis of a repository is not older than one month. An example schedule of the tasks that are executed within one month is depicted in Figure 4.2.

The contributor balances are identified by their git-email. While a registered user on the platform can have multiple verified git-email addresses, it is also possible that a balance cannot be mapped to a user because no platform account for a given git-email exists. As opposed to other platforms that require a KYC check before a project or an individual can

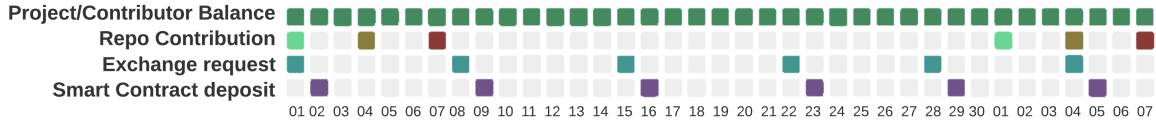


Figure 4.2: Example schedule of payment-related tasks

receive money, this thesis presents an optimistic payment flow. This means that sponsors can donate to projects which do not have any registered contributor on the platform yet. The optimistic payment flow prevents the unfair distribution of money in cases where only one (or only a few) contributors of a project are registered on the platform. In an extreme case without the optimistic payment flow, one contributor could receive the balance of a whole project if he/she was the only registered contributor of a project. However, optimistically allocating money to unregistered contributors carries the risk of locking money.

Therefore, to avoid large amounts of money being locked to contributors that never register, a process to reallocate old balances is needed. Thus, an expiration time of 6 months for individual balances is introduced. It has to be noted that the balance of an individual contributor consists out of multiple daily rewards from different projects. This means that the balance is reallocated in a rolling process and not the whole balance at once. Using this rolling process, a user can claim the balances from the last six months (if any) when he/she signs up. This offers a compromise between locking balances for a long time and taking away rewards from individual contributors.

The flow of expiring and reallocating balances is depicted in Figure 4.3. If a balance expires, the balance is added back to the project and distributed again on the next day together with the daily amount from all the sponsors. Because the project balance is reallocated using the contribution analysis from the past three months, expired balances are distributed among the active contributors. This omits an infinite loop of reallocating expired balances to inactive contributors.

Payout

By paying out the balance in cryptocurrencies, receiving the funds consists only of verifying an email address and adding a payout address. This simplifies the process of receiving money as a contributor because there is no need to first register a project or to do an identity verification. However, paying out in cryptocurrencies also requires an exchange from fiat currency to crypto. Thus, the balances of all registered contributors with a payout address are summed weekly to generate an exchange request. An exchange request is then sent to the administrator who is in charge of exchanging fiat to crypto. Once the fiat money is exchanged, the administrator enters the exchange rate into the system. Each balance (in fiat money) is bound to an exchange request, which is used to calculate the payout value in crypto. The flow of a single balance is visualized in Figure 4.3. In the last step (*Initiate Payout Aggregation*), all balances of a user are combined to one payout.

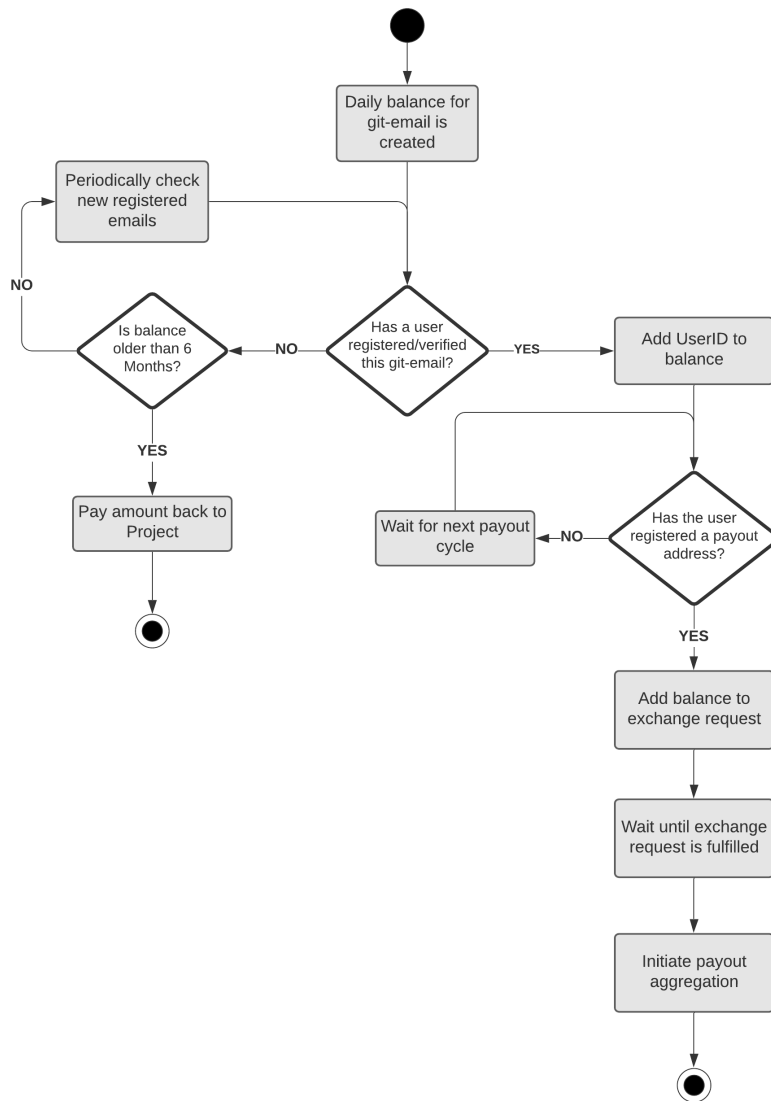


Figure 4.3: Payout flow with reallocation of expired balances

The payout to the registered user follows a “pull payment” process. Instead of periodically creating transactions for all users with a balance greater than zero, the balances are regularly deposited into a smart contract. To receive the payment, users can call a method on the smart contract which initiates the transfer of funds. The pulling of payments shifts the transaction fee to the recipient of the payment. Other platforms cover the transaction fees of payouts with their platform fee. However, with the fanning that happens on the project and contributor level, a sponsor supporting five projects should consequently have a lower platform fee than a sponsor supporting 20 projects. To take it even further, a sponsor supporting a project with only one contributor should, therefore, pay less in fees than a sponsor that supports a project with 20 contributors. Shifting the transaction fee to the recipient of the money thus prevents the unfair use of platform fees to cover transaction costs.

Further, using a smart contract increases transparency and security for the contributors in terms of assigned but not yet paid out balances. Even if a contributor decides to claim his/her balances only yearly (to reduce transaction costs), his/her funds are secured and only claimable by him/herself in the smart contract. Even though the pull payment process would also work without a smart contract, there was no guarantee that a donation platform is actually in possession of this money. Thus, using a smart contract with a pull payment process provides the optimal level of transparency and security from a user perspective.

4.1.5 Architecture

The payment flow of a donation platform requires multiple components that integrate with each other and with external services such as payment providers. This spans from a frontend where a user can enter his/her credit card details to a payout engine that integrates with the smart contract deployed on the blockchain.

The proposed architecture (depicted in Figure 4.4) is based on multiple containerized applications. Containerization allows the flexible deployment on various infrastructure and is, combined with a container orchestration platform, easy to scale. The following components are required to facilitate the above-described payment flow.

- **Database:** The database stores user and project-related data and thus holds the latest state of which user supports which project. Furthermore, the database stores intermediate calculations such as daily balances and data related to the payout.
- **API:** The API (Application Programming Interface) connects to the database and provides various endpoints to manipulate the data. The API checks if incoming requests are authenticated and decides if a user is authorized to manipulate certain data. Furthermore, the API integrates with the payment gateway. It has to be noted that credit card details never touch the API, only metadata about payments.
- **Frontend:** The frontend provides the user interface in order to register and sponsor projects. It directly connects to the payment gateway to transfer the encrypted

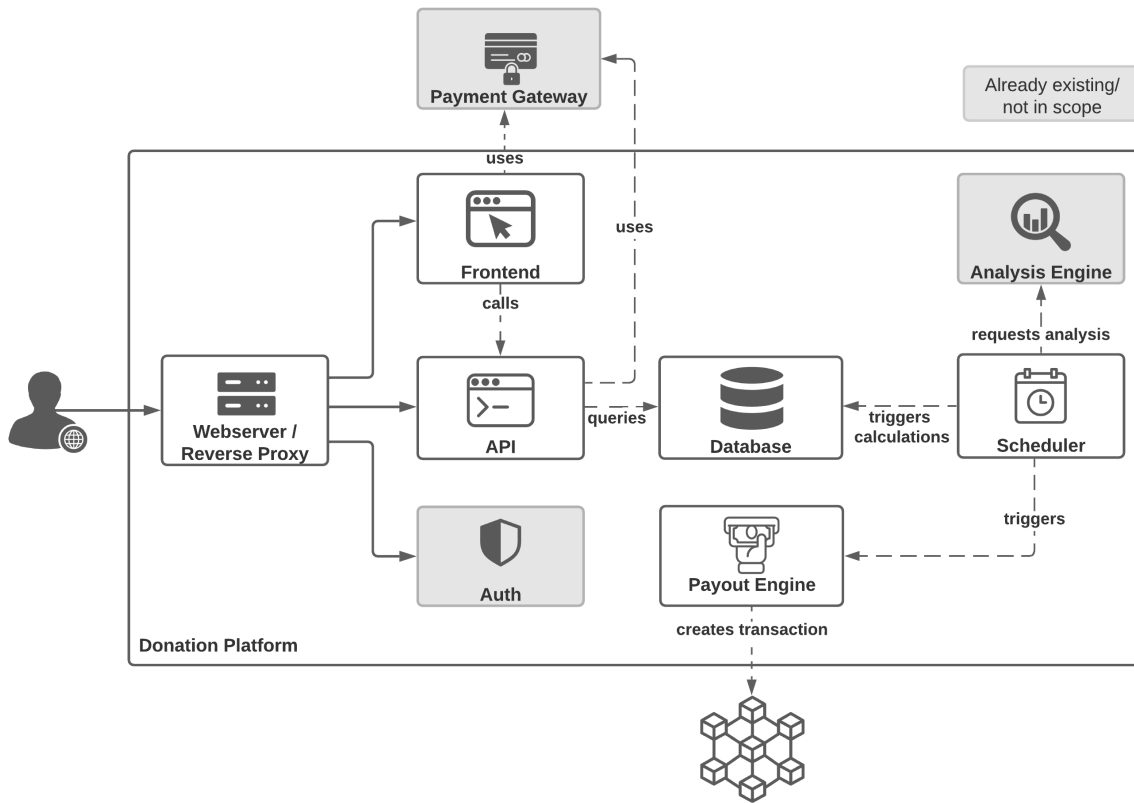


Figure 4.4: Architecture

credit card details and uses the endpoints provided by the API to query data from the database.

- **Authentication Service:** The authentication service manages the identity of users and grants access to request data from the API. It has to be noted that the architecture is based on a stateless (token-based) authentication. This means that the API does not need to connect to the authentication service, but the identity of a user can be validated using a shared secret.
- **Webserver / Reverse Proxy:** The webserver / reverse proxy is the entry point for requests coming from the internet and delegates incoming requests to the appropriate services. In the proposed architecture, the reverse proxy only allows requests to the frontend, the API, and the authentication service. It has to be noted that the frontend “calling” the API is also routed over the reverse proxy.
- **Analysis Engine:** The analysis engine is based on Falter’s commit evaluation engine [6]. It analyzes the git repositories of the sponsored projects. The analysis engine can only be called internally and is triggered by the scheduler.
- **Scheduler:** The scheduler triggers repeating tasks (*cf.* Figure 4.2) and aggregates data in the database.
- **Payout Engine:** The payout engine connects to the smart contract on the blockchain

and deposits the money. Payouts are triggered by the scheduler, which means the payout engine only executes payments but is not aware of the application logic or the data in the database.

4.1.6 Database Design

The data is stored in a relational database. This offers the possibility to do all the calculations and aggregations directly on the database level with queries that would not be possible with, for example, a document-based database. Figure 4.5 details the ER-Diagram to facilitate the above-described payment flow. The two most important tables are the *User* and *Repo* table, as most other tables are related to one or both of these tables.

- **User:** The user table contains all user-related data except authentication data, which is externally managed in the authentication service. The table *pay_out_address* and *git_email* extend the user data. They are separated from the *user* table because there exists a one-to-many relation between *user* and *pay_out_address* respectively *git_email*. The sum of *payout* entries is the balance of a user. Each payout is linked to an *exchange* entry and contains a transactions hash (*tx*) once the *payout* is included in a transaction to the smart contract.
- **Repo:** The *repo* table contains the data about projects that can be supported. A repository accumulates *daily_repo_balances* which stem from the sponsors that support the repository. Additionally, a *repo* can also accumulate general *repo_balances* if, for example, a *git_user_balance* expires. If a *repo_balance* is again distributed among the current active contributors, a *repo_balance_event* is added which closes the particular *repo_balance*.
- **Git user balance:** The *git_user_balance* table contains the balances of all contributors. The *git_user_balance_event* table contains information what happened with the balance. If someone has registered the *git_email* of a *git_user_balance*, an event with type PAY_USER is created, otherwise an event with type PAY_REPO is created and the balance is included in a *repo_balance*.
- **Sponsor Event:** The *sponsor_event* table contains the mapping between *user* and *repo*. Because each action (*e.g.*, sponsoring or unsponsoring a repository) is stored, it is possible to get the sponsored repos of a user at any timestamp, not only at the current time.
- **Payments:** The *payments* table contains all received payments, including subscription start- and end-dates. The *amount* field in the table refers to the received net amount (after payment fees), which is used to calculate the exact sponsor amount to each repository.
- **Analysis Request:** The *analysis_request* table keeps track of all requests made to the analysis engine (*cf.* subsection 4.1.5) and is used to determine, if a new analysis has to be requested. The *contribution* table contains the response of the analysis request, *i.e.*, the score (weight) for each contributor.

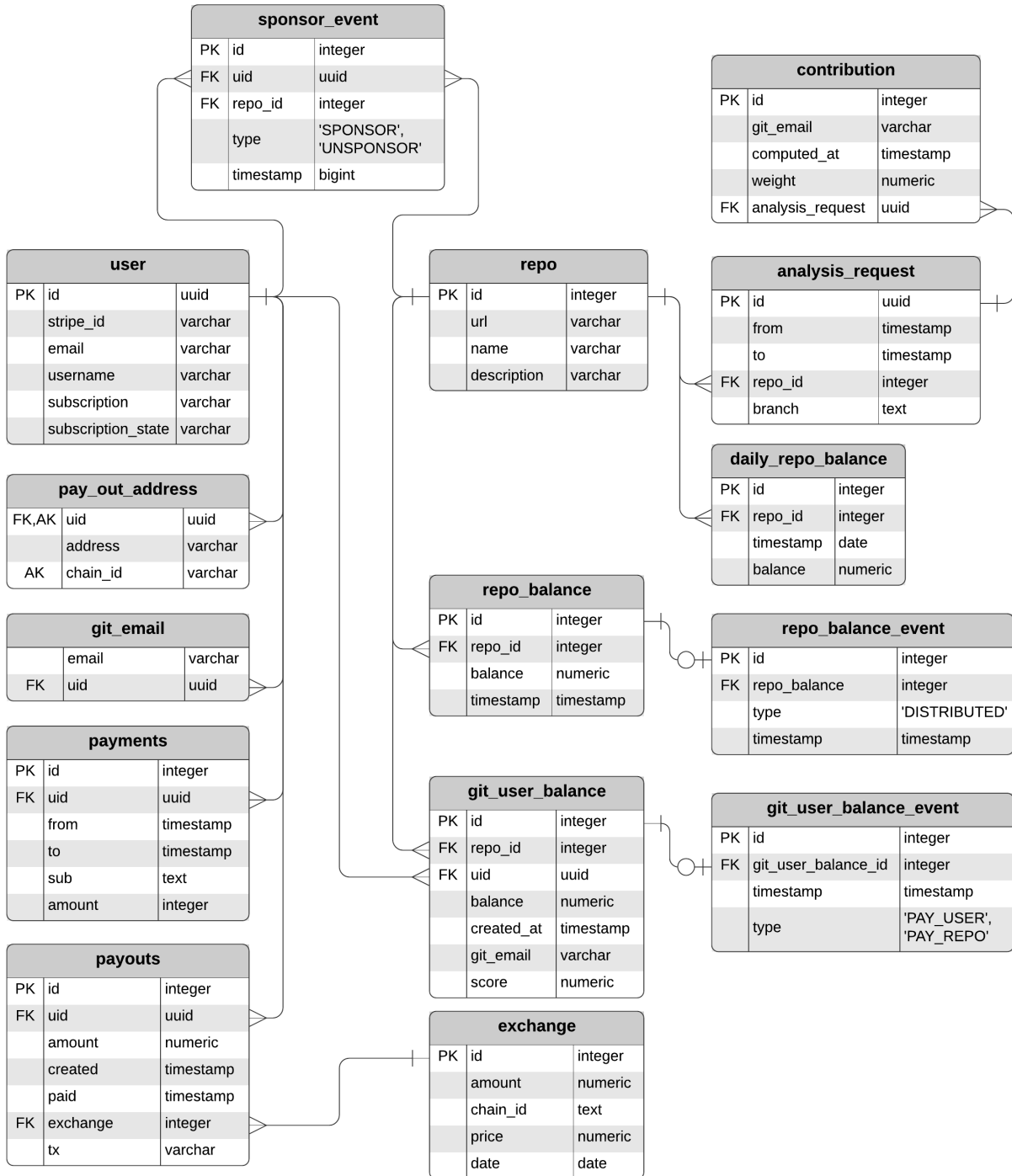


Figure 4.5: ER-Diagram

4.2 Implementation

This section details the implementation (*cf.* Figure 4.6) of the proposed design. The payment flow source code is available at [44]. The containerization of the different services is done using Docker. Additionally, the Docker Compose tool allows the creation of multi-container applications, which means that all services can be started with a single command. With the default network created by Docker Compose, the containers can look up the *hostname* of other containers to retrieve their IP address. This makes the communication between containers straightforward (*e.g.*, using REST-APIs), as the `scheduler` container can call the `payout` container using `http://payout:PORT/resource`.

Listing 4.1 shows an excerpt of the `docker-compose.yml` file which is used to create the multi-container application. It shows how only the reverse-proxy service exports ports to the outside (line 30-31). On the other hand, the database service can only be reached from within the Docker network using the hostname `db`.

```

1  version: "3"
2  volumes:
3    caddy_data:
4    postgres_data:
5  services:
6    db:
7      container_name: flatfeestack-db
8      build:
9        context: ./
10       dockerfile: ./Dockerfile-Postgres
11     volumes:
12       - postgres_data:/var/lib/postgresql/data
13     environment:
14       - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
15       - POSTGRES_USER=${POSTGRES_USER}
16       - POSTGRES_DB=${POSTGRES_DB}
17
18     reverse-proxy:
19       container_name: "flatfeestack-caddy"
20       build:
21         context: ./
22         dockerfile: ./Dockerfile-Caddy
23       volumes:
24         - caddy_data:/root/.caddy
25       ports:
26         - "80:80"
27         - "443:443"

```

Listing 4.1: Extract of the `docker-compose.yml` file

As Figure 4.6 depicts, the different applications to facilitate the payment flow have mostly been implemented in Go. Only the frontend relies on a different programming language and uses JavaScript. The calculation of the donation amounts is predominantly done on database-level using SQL. The following sections highlight implementation decisions regarding the payment flow.

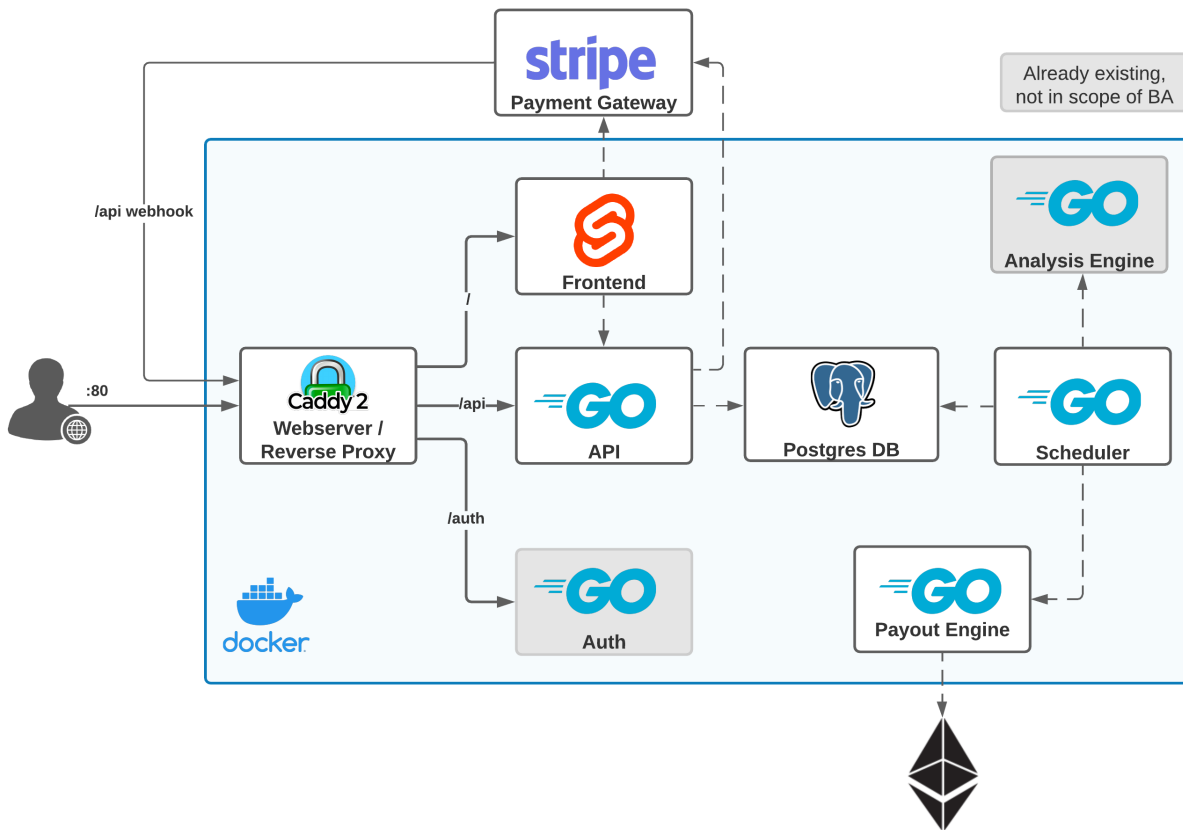


Figure 4.6: Implementation of the architecture

4.2.1 Stripe

Stripe [13] has been chosen as payment provider for this implementation because it offers SDKs and libraries for JavaScript and Go. The well-documented API of Stripe further allows the tight integration into the payment flow, which is needed to distribute the correct amount of money among the contributors. Stripe also allows repeating payments (*i.e.*, subscriptions) without charging a fixed monthly price. Following components of Stripe have been used:

Stripe.js is the JavaScript SDK of Stripe. The library also provides Stripe Elements, which can be used to build the payment form and to collect sensitive data (*i.e.*, credit card details). Stripe.js tokenizes the form inputs so that sensitive credit card information never touches the server of the donation platform. Furthermore, the library provides functionality to handle 3D Secure.

Stripe-go is the official Go library of Stripe. It offers types for all Stripe resources and abstracts the data fetching and authentication. The stripe-go library is used in the API service to provide a *webhook* which receives payment-related events from Stripe. For the donation platform, the following events are of interest:

- `customer.subscription.created`

- `customer.subscription.updated`
- `customer.subscription.deleted`
- `invoice.payment_failed`
- `invoice.payment_succeeded`

The subscription events are used to update the subscription state of a user in the database. The state in the database is then used to decide if the credit card form has to be shown to the user in the frontend or if the user already has a subscription and can start sponsoring projects. In the background, a Stripe subscription consists of multiple invoices (*e.g.*, monthly, quarterly or yearly), which means that the actual payments are separated events from the subscription. Thus, the invoice events `payment_failed` and `payment_succeeded` are used to save the last received net amount of a user in the database. By relying on the received net amount, it can be omitted that unpaid sponsor amounts are distributed among contributors (*i.e.*, if a user has an active subscription and sponsored repositories, but the issuing bank declined the latest payment).

Listing 4.2 shows the implementation of the *StripeWebhook* in the API service. The signature of each request is verified using the `webhook.ConstructEvent` method and the shared secret. The different event types are handled in the switch case (line 20).

```

1 func StripeWebhook(w http.ResponseWriter, req *http.Request) {
2     const MaxBodyBytes = int64(65536)
3     req.Body = http.MaxBytesReader(w, req.Body, MaxBodyBytes)
4     payload, err := ioutil.ReadAll(req.Body)
5     if err != nil {
6         log.Printf("Error reading request body: %v\n", err)
7         w.WriteHeader(http.StatusBadRequest)
8         return
9     }
10    event, err := webhook.ConstructEvent(payload,
11                                       req.Header.Get("Stripe-
12                                           Signature"),
13                                       "whsec_XXXXXXXXXX")
14    if err != nil {
15        w.WriteHeader(http.StatusForbidden)
16        log.Printf("Error evaluating signed webhook request: %v", err)
17        return
18    }
19    // Unmarshal the event data into an appropriate struct depending on
20    // its Type
21    switch event.Type {
22    case "customer.subscription.created":
23        var subscription stripe.Subscription
24        err := json.Unmarshal(event.Data.Raw, &subscription)
25        w.WriteHeader(checkSubscription(&subscription))
26    // ...other event types
27    }

```

Listing 4.2: Stripe Webhook in the API service

4.2.2 Smart Contract

The payout is based on the Ethereum blockchain. Therefore, the smart contract is implemented using the Solidity language. The smart contract's main functionality is to store the funds of each registered contributor in a transparent and secure way and allow contributors to claim their funds at any time. The balances are stored using a mapping between `address` and `uint256`. Because arithmetic operations in Solidity "wrap" on overflow (*i.e.*, if an integer overflows, the most significant bits are lost), OpenZeppelin's `SafeMath` [45] library is used for all arithmetic operations. The `SafeMath` library throws an error if an overflow happens and thereby ensures that incorrect transactions are rolled back.

The smart contract implements the following three methods:

- **Fill:** The *fill* method is used to update the balances of addresses (*i.e.*, contributors) and can only be called by the owner of the contract. Thus, on smart contract deployment, the owner's address is stored in the `_owner` variable. This variable is used to ensure that only the owner can call the *fill* method. Further checks ensure that the input arrays `addresses_` and `balances_` have the same length, as each balance needs to be assigned to a specific address. The for-loop (line 20) updates the stored balances of the smart contract based on the two input arrays of the method. Additionally, the sum of all input balances is calculated in the loop. The sum is used to ensure that the assigned balances do not exceed the value of the transaction (*i.e.*, the amount of ether that is deposited to the smart contract).
- **Release:** By calling the release method, a user can claim the balances that are assigned to his address. The transaction costs, in the form of gas, are thereby covered by the user who requests the funds. Before a transfer is initiated, it is verified that the balance of the requested address is greater than zero (line 33). To avoid race conditions when the function is called rapidly multiple times, the balance of the address is set to zero before the transfer is initiated. The order of lines 35 and 36 is very important, as race conditions have already been exploited to hack the DAO, which eventually led to a hard fork of the Ethereum blockchain to restore stolen balances [46].
- **BalanceOf:** The `balanceOf` method returns the balance of an address. Since the default value of all possible mappings in Solidity is 0, the `balanceOf` method always returns a value, even if the requested address has never been registered in the smart contract.

```

1  contract Flatfeestack {
2      using SafeMath for uint256;
3      mapping(address => uint256) balances;
4      address private _owner;
5      event PaymentReleased(address to, uint256 amount);
6
7      constructor () {
8          _owner = msg.sender;
9      }
10     /**
11      * @dev Fill contract with array of balances
12      * @param addresses_ , balances_ payouts to update
13      */
14     function fill(address[] memory addresses_, uint256[] memory
15         balances_) public payable {
16         require(msg.sender == _owner, "Only the owner can add new
17             payouts");
18         require(addresses_.length == balances_.length, "Addresses and
19             balances array
20             must have the same length");
21         uint256 sum;
22         for (uint i=0; i < addresses_.length; i++){
23             balances[addresses_[i]] = balances[addresses_[i]].add(
24                 balances_[i]);
25             sum = sum.add(balances_[i]);
26         }
27         if(sum > msg.value) {
28             revert("Sum of balances is higher than paid amount");
29         }
30     }
31     /**
32      * @dev Triggers a transfer of the assigned funds.
33      * total shares and their previous withdrawals.
34      */
35     function release() public {
36         require(balances[msg.sender] > 0, "PaymentSplitter: account has
37             no balance");
38         uint256 _balance = balances[msg.sender];
39         balances[msg.sender] = 0;
40         msg.sender.transfer(_balance);
41         emit PaymentReleased(msg.sender, _balance );
42     }
43     /**
44      * @dev Return balance
45      * @return value of 'number'
46      */
47     function balanceOf(address address_) public view returns (uint256){
48         return balances[address_];
49     }
50 }

```

Listing 4.3: Ethereum Smart Contract

4.2.3 SQL Calculations

The calculation of contributor balances is done directly on the database level and is a multi-step procedure. First, the daily repo balances have to be calculated. The balances of the repos are then joined with the analysis engine's contribution weights, thereby creating *git_user_balance* entries. In a final step, all the git balances can be combined to payouts on a per user basis. The stored procedure functionality of PostgreSQL is used to make the queries more readable.

The following helper procedures are used to calculate the daily repo balances. All queries are based on the ER-diagram in Figure 4.5.

get_payment_at

The procedure returns the latest payment of a user that is active for a specific date. If there is no active payment on this date, the procedure returns null.

```

1 CREATE OR REPLACE FUNCTION get_payment_at(uuid, date)
2 RETURNS TABLE(id integer, uid uuid, "from" timestamp, "to" timestamp,
3               sub text, amount integer)
4 LANGUAGE sql
5 AS $$
6 SELECT p.id, p.uid, p.from, p.to, p.sub, p.amount
7 FROM (SELECT MAX("to") as "to", uid
8       FROM payments
9       WHERE "to"::DATE >= $2
10            AND "from"::DATE <= $2
11            AND uid = $1
12       GROUP BY uid) as latest
13 JOIN payments p ON p.to = latest.to AND p.uid = latest.uid;

```

Listing 4.4: PostgreSQL stored procedure - get_payment_at

get_monthly_sponsor_amount_at

Based on the *get_payment_at* procedure, this procedure calculates the monthly sponsor amount of a user. This additional step is needed, because payments can cover different time intervals (*e.g.*, quarterly or yearly payments), and the net amount differs between those (*cf.* Section 4.1.4).

```

1 CREATE OR REPLACE FUNCTION get_monthly_sponsor_amount_at(uuid, timestamp)
2 RETURNS integer
3 LANGUAGE sql
4 AS $$
5 SELECT amount / (EXTRACT(YEAR FROM age) * 12 + EXTRACT(MONTH FROM age))
6 ::int AS monthly_amount
7 FROM (SELECT age("to", "from"), uid, amount from get_payment_at($1, $2::
8       DATE)) as p;

```

Listing 4.5: PostgreSQL stored procedure - get_monthly_sponsor_amount_at

get_sponsored_repos_at

Since there does not exist a simple mapping between `user` and `repo` but the sponsoring of a repository is based on `sponsor_events`, a procedure to get all sponsored repos at a certain point in time is needed. The below procedure selects the latest sponsor event per repo and user (it is possible that a user sponsors and unsponsors a repository multiple times) and returns all entries where the latest event was of type `'SPONSOR'`.

```

1 CREATE OR REPLACE FUNCTION get_sponsored_repos_at(date)
2 RETURNS TABLE(repo_id integer, user_id uuid)
3 LANGUAGE sql
4 AS $$
5 SELECT s.repo_id,
6        s.uid AS user_id
7 FROM (SELECT sponsor_event.uid,
8              sponsor_event.repo_id,
9              max(sponsor_event."timestamp") AS "timestamp"
10        FROM sponsor_event
11        GROUP BY sponsor_event.uid, sponsor_event.repo_id) as latest
12 JOIN sponsor_event s ON latest.uid = s.uid AND latest.repo_id = s.
13                        repo_id AND latest."timestamp" = s."timestamp"
14 WHERE s.type = 'SPONSOR'::sponsorevent;
15 $$

```

Listing 4.6: PostgreSQL stored procedure - `get_sponsored_repos_at`**Daily Repo Balance**

With all the above helper procedures, the `get_daily_repo_balance` procedure can be written compactly. The daily sponsor amount of a user (using another `days_in_month` helper procedure and the `get_monthly_sponsor_amount` procedure) is divided by the number of repos a user supports. Daily repo balances are rounded to three decimal places. It has to be noted that the net amount of the payments is already in the smallest currency unit (*e.g.*, “Rappen” in case of swiss francs), and the daily repo balance is, therefore, 10^3 times more precise than what can be paid out. The additional precision is used so that the amount can be divided among the contributors. The last `WHERE` clause (line 9) omits all sponsor amounts of users without an active payment.

```

1 CREATE OR REPLACE FUNCTION get_daily_repo_balance_at(date)
2 RETURNS TABLE(repo_id integer, balance numeric)
3 LANGUAGE sql
4 AS $$
5 SELECT repo_id,
6        ROUND(((get_monthly_sponsor_amount_at(user_id, $1)::FLOAT /
7              days_in_month($1)) /
8              (countsponsoredrepos_at(user_id, $1)::FLOAT)::NUMERIC,6) as
9              sponsor_amounts
10 FROM get_sponsored_repos_at($1)
11 WHERE get_monthly_sponsor_amount_at(user_id, $1) is not NULL
12 $$

```

Listing 4.7: PostgreSQL stored procedure - `get_daily_repo_balance`

Inserting Daily Repo Balances

The insertion of daily repo balances into the database is triggered by the scheduler (*cf.* Section 4.2.4) and executed using a transaction. The daily repo balance is the sum of daily sponsor amounts (`get_daily_repo_balance_at`) and the existing `repo_balance` (*i.e.*, expired balances). In a first step, those two values are summed and inserted into the `daily_repo_balance` table. In a second step, a new `repo_balance_event` is added for all repo balances included in the preceding summation. The `repo_balance_event` “closes” an existing repo balance. The usage of a transaction prevents an inconsistent state, where a `repo_balance` is included in the daily calculation but not closed with a `repo_balance_event`.

```

1 BEGIN TRANSACTION;
2 INSERT INTO daily_repo_balance (repo_id, balance, "timestamp")
3   (SELECT daily_repo_id, SUM(daily.balance+COALESCE(existing.balance
4     ,0)), ($1)::date
5     FROM get_daily_repo_balance_at(($1)::date) daily
6     LEFT JOIN
7       (SELECT b.repo_id, b.balance FROM repo_balance b WHERE NOT
8         EXISTS (SELECT * FROM repo_balance_event e WHERE e.
9           repo_balance = b.id)) AS existing
10    ON daily.repo_id = existing.repo_id
11   GROUP BY daily.repo_id, existing.repo_id) RETURNING repo_id,
12     balance;
13
14 INSERT INTO repo_balance_event(repo_balance, "type")
15   (SELECT id, 'DISTRIBUTED'::repobalanceevent FROM repo_balance b
16     WHERE NOT EXISTS
17       (SELECT * FROM repo_balance_event e WHERE e.repo_balance=b.id));
18 COMMIT;

```

Listing 4.8: Inserting daily repo balances

From repo balances to git user balances

The contributor fanning (*cf.* Section 4.1) is achieved by assigning the `repo_balance` to the respective git user. This query is triggered daily by the scheduler after the repo balances have been calculated. First, the latest analysis request (`latest_analysis`) that is not older than one month is selected, and the contribution weights and repo balances are joined to the request. The contribution weight and the daily repo balance are multiplied (line 4). Since a git user can earn balances from multiple repositories, the sum of `daily_repo_rewards` is calculated (line 3). At the bottom (line 15), `git_email` is joined to the result so that userIDs can be assigned to balances of registered emails. The result is inserted in the `git_user_balance` table.

```

1 INSERT INTO git_user_balance (git_email, balance, repo_id, score, uid)
2   (SELECT ub.git_email, ub.balance, ub.repo_id, ub.score, g.uid FROM
3     (SELECT git_email, SUM(daily_repo_reward) balance, repo_id, SUM(
4       weight) as score
5     FROM (SELECT c.git_email, c.weight * drb.balance as
6       daily_repo_reward, drb.repo_id, c.weight
7     FROM latest_repo_balance_at($1) drb
8     JOIN (SELECT repo_id, MAX("to") as "to"
9       FROM analysis_request a
10      WHERE a."to" > ($1 - '1 MONTH'::interval)::DATE

```

```

9         GROUP BY repo_id) as latest_analysis
10        ON latest_analysis.repo_id = drb.repo_id
11        JOIN analysis_request ar ON drb.repo_id = ar.repo_id AND
        latest_analysis."to" = ar."to"
12        JOIN contribution c ON c.analysis_request = ar.id
13        WHERE drb.timestamp = ($1)::date) as rewards
14        GROUP BY git_email, repo_id) as ub
15    LEFT JOIN git_email g ON ub.git_email = g.email);

```

Listing 4.9: Calculation of git user balances

Calculating payouts

From the list of all `git_user_balances`, the payouts can be calculated by summing the open balances grouped by userID. The open `git_user_balances` that were assigned to a payout are then “closed” by inserting a `git_user_balance_event` with type `'PAY_USER'`.

```

1 BEGIN TRANSACTION;
2 WITH unclaimed_balances AS (
3     (SELECT b.id, b.uid, b.git_email, b.balance
4     FROM git_user_balance b
5     WHERE NOT EXISTS
6         (SELECT * FROM git_userbalanceevent e WHERE b.id = e.
          git_user_balance_id)
7     AND uid IS NOT NULL)
8 )
9 INSERT INTO payouts(uid, amount) (SELECT uid, SUM(balance) FROM
    unclaimed_balances GROUP BY uid);
10 INSERT INTO git_user_balance_event(git_user_balance_id, "type") (SELECT
    id, 'PAY_USER'::userbalanceevent FROM unclaimed_balances ub);
11 COMMIT;

```

Listing 4.10: Creating payout entries

4.2.4 Scheduler

The scheduler triggers and executes multiple repeating tasks according to the example schedule shown in Figure 4.2. The service is written in Go and uses the goCron [47] job scheduling package. The scheduled tasks are listed below:

Daily Runner

The daily runner is always started after midnight and executes all the calculations for the previous day, which include:

- Calculate and store the daily repo balances (*cf.* Section 4.2.3).
- Calculate and store the git user balances based on the contribution weights of the latest analysis request (*cf.* Section 4.2.3).
- Calculate payout per user (*cf.* Section 4.2.3).

- Check for expired `git\_user\_balances` (*i.e.*, balances that are older than six months and not assigned to a registered user). The expired balances are paid back to the repository in the form of a `repo_balance` and distributed to the active contributors on the next day.

Update expired analysis requests

When calculating the `git\_user\_balance`, an analysis request of the repository which is not older than one month is required. The scheduler ensures that the analysis request and contribution weights are always up to date. This is achieved by daily checking and updating analysis requests that are older than 25 days. Even though this task is executed daily, it is not part of the daily runner as it runs one hour prior. The buffer of 5 days is added for the case where the analysis request can not be fulfilled, for example, because of rate limits. If an analysis request is unsuccessful, it is retried on the next day.

The response of the analysis request can take multiple seconds or even minutes for large repositories [6]. Thus, the scheduler provides a webhook which is called by the analysis request once the analysis of the repository is completed. The webhook receives the results of the analysis and stores them in the database. If a repository is sponsored for the first time (and, thus, added to the database), an initial analysis request is triggered by the API when handling the sponsor event. Still, the results are sent to the webhook of the scheduler.

Create exchange requests

Every week, the sum of all payouts is calculated so that the amount can be exchanged from fiat to crypto. The exchange request is then visible to the administrator in the frontend. After creating the exchange entry, the scheduler updates the payouts with the respective `exchangeID`. The payout can be triggered as soon as the exchange request is fulfilled (*i.e.*, the administrator has entered an exchange rate).

Check for open payouts

The scheduler regularly checks for open payouts that have a fulfilled exchange request. The check is simple and can be done hourly. This ensures that once an exchange request is fulfilled, the smart contract's `fill` method is triggered within one hour. To deposit the money to the smart contract, the scheduler calls the payout engine which then interacts with the smart contract.

```

1 func main() {
2     scheduler := gocron.NewScheduler(time.Local)
3     // Calculate the daily repo balance for the day before
4     _, err := scheduler.Every(1).Day().At("00:01").Do(DailyRunner)
5     checkOrThrow(err)
6
7     // Check for repos that have had sponsors in the last month but an
8     // analysis that is older than 25 days
9     _, err = scheduler.Every(1).Day().At("23:00").Do(GetRepoContribution)
10    checkOrThrow(err)
11
12    // Sum payouts to check how much money has to be transferred to an
13    // exchange
14    _, err = scheduler.Every(1).Week().Do(CreateExchangeEntries)
15    checkOrThrow(err)
16
17    // Check for open payouts
18    _, err = scheduler.Every(1).Hour().Do(PayoutTask)
19    checkOrThrow(err)
20
21    scheduler.StartAsync()
22
23    // only internal routes, not accessible through caddy server
24    router := mux.NewRouter()
25    router.HandleFunc("/hooks/analysis-engine", AnalysisEngineHook).
26        Methods("POST", "OPTIONS")
27
28    log.Fatal(http.ListenAndServe(":8080", router))
29 }

```

Listing 4.11: main function which initializes the scheduler and the webhook

4.2.5 API

The API service provides access to the resources in the database by providing a REST-API. To handle HTTP requests, the *gorilla/mux* package [48] is used. This package is also used in all other services that provide REST endpoints.

Based on the shared secret, the API checks if the incoming requests are authenticated. By implementing the stripe webhook (*cf.* listing 4.2), the API handles the payment and subscription logic. This means that the API also handles the deduction of platform fees. For the current state, the platform fee is zero, so the API does not deduct anything from the received net amount of a payment. Listing 4.12 shows an excerpt of the API's main function which initializes the different endpoints. The `AuthMiddleware` function, which is used as middleware, validates the token of the incoming requests. The stripe webhook does not use the token-based authentication (and is whitelisted in the middleware) but checks the signature of the incoming webhook request inside the handler function (*cf.* Listing 4.2).

The other endpoints provided by the API are mainly used to create and retrieve resources from the database. They do not contain significant business logic, as most of the calculations are triggered by the scheduler.

```

1 func main() {
2     opts = NewOpts()
3     db = initDb()
4
5     stripe.Key = opts.StripeSecret
6
7     // Routes
8     router := mux.NewRouter()
9     apiRouter := router.PathPrefix("/api").Subrouter()
10    apiRouter.Use(AuthMiddleware())
11    //user
12    apiRouter.HandleFunc("/users/me", GetMyUser).Methods("GET", "OPTIONS")
13    apiRouter.HandleFunc("/users/me/connectedEmails", GetMyConnectedEmails
14        ).Methods("GET", "OPTIONS")
15    apiRouter.HandleFunc("/users/me/connectedEmails", AddGitEmail).Methods
16        ("POST", "OPTIONS")
17    //...
18    apiRouter.HandleFunc("/hooks/stripe", StripeWebhook).Methods("POST", "
19        OPTIONS")
20 }

```

Listing 4.12: Excerpt of the API main function which uses Gorillas’ mux package

4.2.6 Frontend

The frontend is written in JavaScript using Svelte [49]. Compared to other frontend frameworks, Svelte uses an additional compile step on build time, which reduces the browser workload and increases the speed. Regarding the payment flow, the main functionalities of the frontend are:

- Let the user register a new account and login afterward
- Let the user choose a payment interval and provide a credit card input form
- Let the user sponsor / unsponsor repositories
- Let the user add payout related information (*e.g.*, , ethereum address)
- Facilitate the easy claiming of balances from the smart contract

Several listed functionalities only provide an interface to update resources in the database (using the API). Those implementations are not further discussed since they do not directly affect the payment flow. The payment is also not further discussed, as it is based on the components mentioned in Section 4.2.1.

An essential step in the payment flow of the OSDP is the payout. Since the platform relies on a “pull payment” approach (*cf.* Section 4.1.4), the user has to claim his/her balance from the smart contract actively. The frontend uses the *Web3.js* [50] library to provide a mechanism for the user to conveniently interact with the smart contract. Listing 4.13 shows the implementation of a “Request funds” button, which can be used to claim the

balances from the smart contract. The svelte component relies on an injected Ethereum provider (*e.g.*, MetaMask [51]), which can usually be accessed using the `window.ethereum` object. If the button is clicked, the `requestFunds` method is called. The method first checks if the user has a Web3 enabled browser using the `ethEnabled` method. With the address and the application binary interface (ABI) of the deployed smart contract, a new contract object can be instantiated (line 24). The `release` method on the contract object is then called with the first (*i.e.*, the selected) account in MetaMask. Afterward, the user only has to sign the transaction to receive his/her funds. The entire flow of claiming the balance in the smart contract is detailed in Section 5.2.

```

1 <script lang="ts">
2 import Web3 from "web3";
3 import { ABI, CONTRACT_ADDRESS } from "../types/contract";
4
5 let metamask = null;
6 const ethEnabled = () => {
7   if (window.ethereum) {
8     metamask = new Web3(window.ethereum);
9     try {
10       window.ethereum.enable();
11       return true;
12     } catch (e) {
13       console.log("No MetaMask (or similar) installed");
14       return false;
15     }
16   }
17   return false;
18 };
19
20 const requestFunds = async () => {
21   if(!ethEnabled()) return;
22
23   const accounts = await metamask.eth.getAccounts();
24   const contract = new metamask.eth.Contract(ABI, CONTRACT_ADDRESS);
25   contract.methods.release().send({ from: accounts[0] });
26 };
27 </script>
28
29 <button class="button" on:click="{requestFunds}">Request funds</button>

```

Listing 4.13: Web3.svelte component

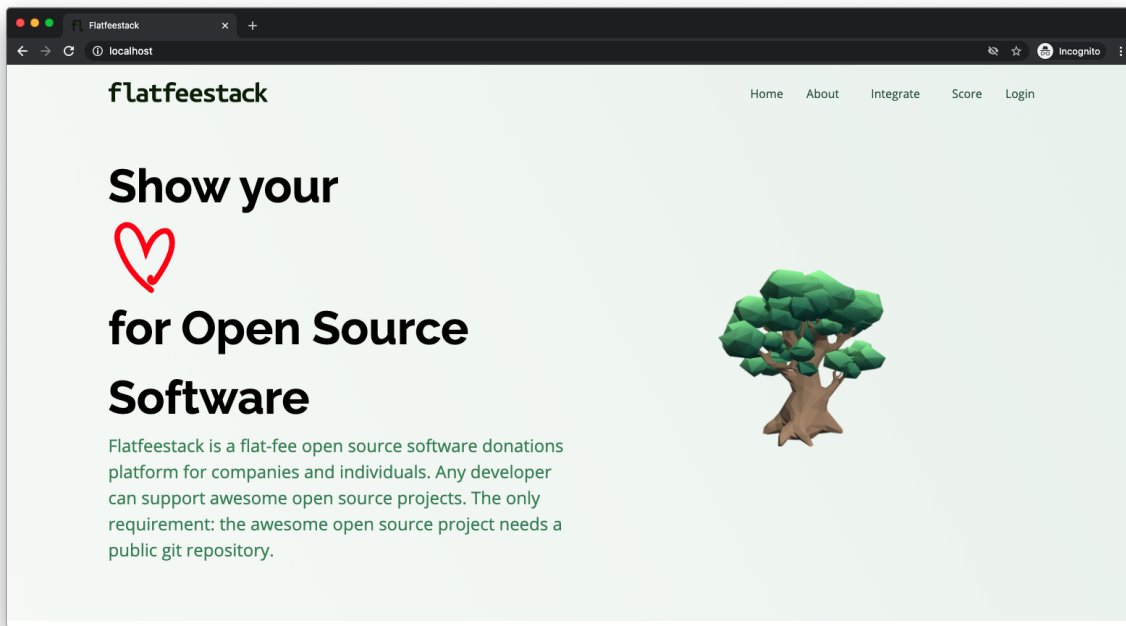


Figure 4.7: Donation Platform Frontend

4.2.7 Caddy Server

Caddy 2 [52] is used as web server and reverse proxy. The incoming requests to `/auth`, `/api` are delegated to the respective containers using their hostname. All other incoming requests are directed to the `var/www/html` folder using the `try_files` command. The configuration of the Caddy server is stored in a *Caddyfile* (cf. Listing 4.14).

```

1  :80
2  root * /var/www/html
3  route /auth/* {
4      uri strip_prefix /auth
5      reverse_proxy auth:8080
6  }
7  route /api/* {
8      reverse_proxy api:8080
9  }
10 @notProxy {
11     not {
12         path /api/* /auth/* /demo/*
13     }
14     file {
15         try_files {path} {file} /index.html
16     }
17 }
18 rewrite @notProxy {http.matchers.file.relative}
19 file_server

```

Listing 4.14: Caddyfile of the reverse proxy

Chapter 5

Evaluation and Discussion

This chapter evaluates the design and implementation of the presented payment flow. While previous chapters discussed the technical design and implementation of the payment flow, this chapter discusses the platform's user interface and the required user interactions in two use cases. The use of cryptocurrencies provides advantages in transparency and execution speed of payments. However, there are drawbacks in terms of costs. Thus, this chapter compares the use of crypto payments for donation platforms to classic fiat payments in terms of cost and user-friendliness.

Section 5.1 presents the user interactions that are needed to create a subscription and sponsor projects. Section 5.2 presents the user interface from the view of a developer who wants to receive money. The administrative tasks related to the payment flow are discussed in Section 5.3. A cost evaluation of the payment flow is done in Section 5.4 based on the two blockchains presented in Chapter 2.2. Finally, the feasibility, together with advantages and disadvantages of the payment flow, are discussed in Section 5.5.

5.1 Use Case #1 - Sponsor

Following the process of related donation platforms (*cf.* Chapter 3.1), the payment happens in fiat money using credit card transactions. However, a payment is not bound to a fixed project. This means that the subscription is initially created without having decided which projects to sponsor. This flow is also reflected in the user interface, as after registration / login (*cf.* Figure 5.1) the user is directly asked to create a subscription (*cf.* Figure 5.2).

The registration/login is based on email and password as there is no additional information needed to collect credit card payments.

Based on the fee structure described in subsection 2.1, the yearly payment interval is highlighted and pre-selected (*cf.* Figure 5.2). A user is also free to choose the monthly payment interval. However, in this case, less money can eventually be distributed to the projects.

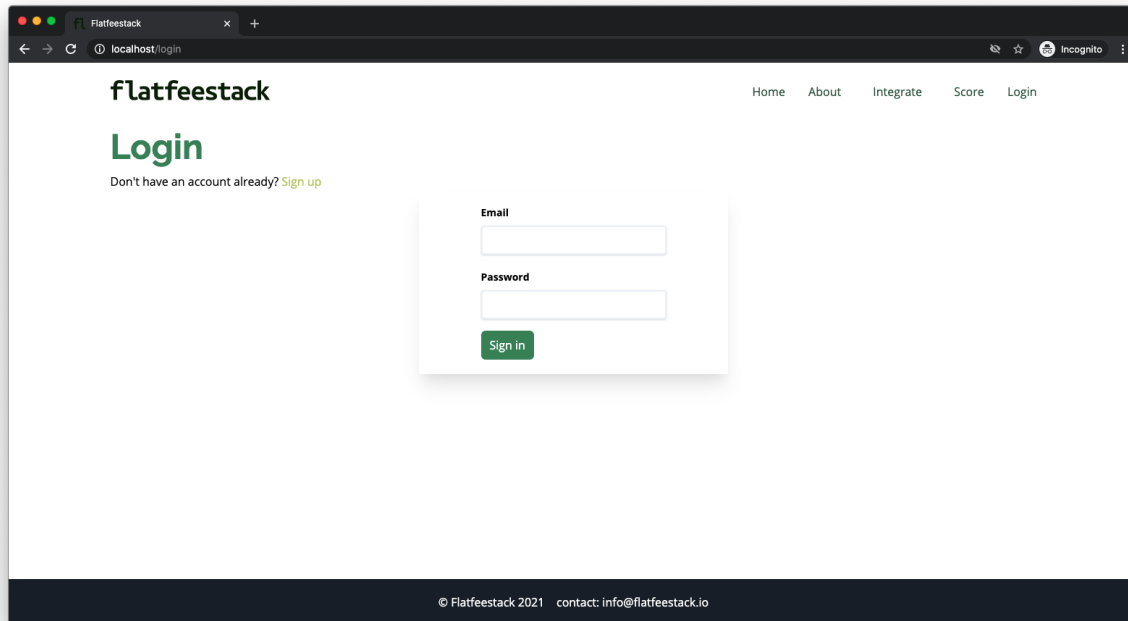


Figure 5.1: Donation Platform Login

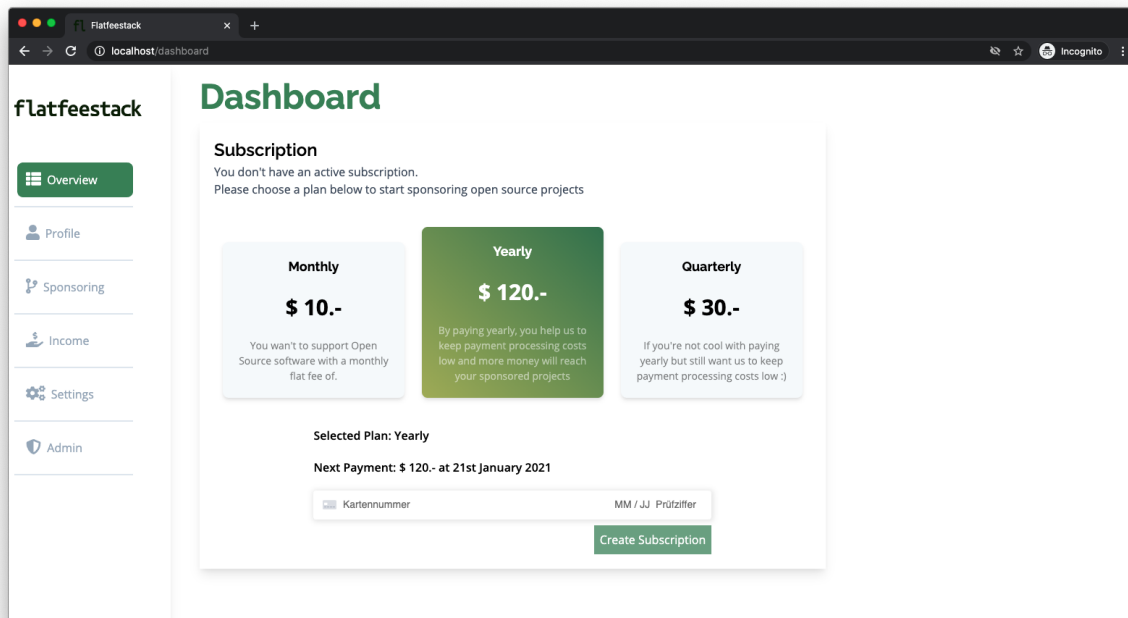


Figure 5.2: Credit card payment

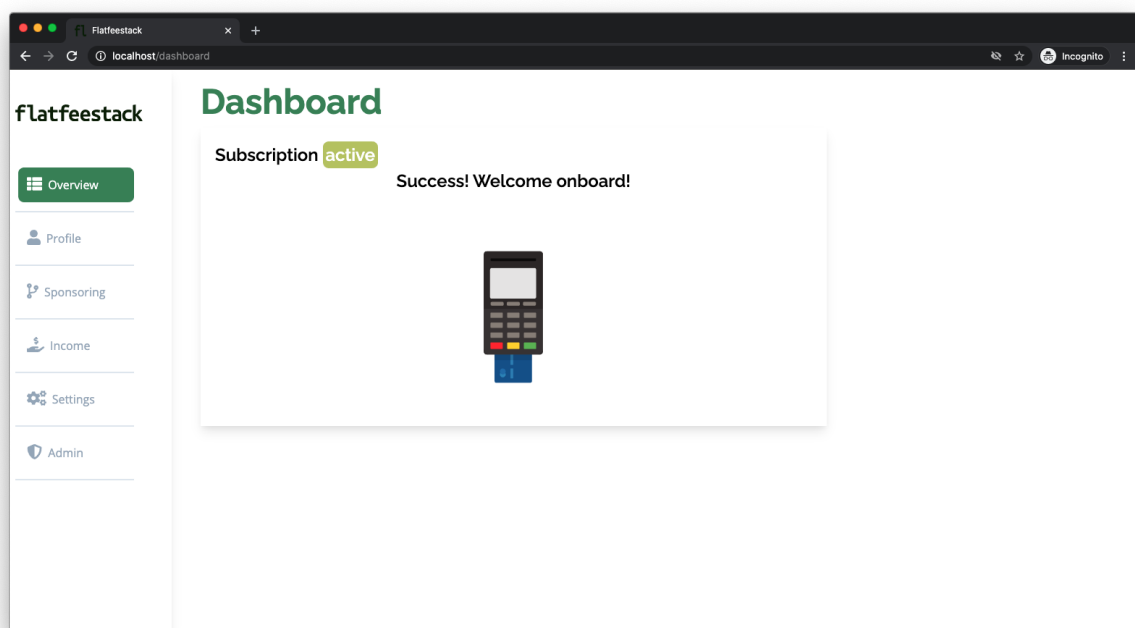


Figure 5.3: Subscription activation success screen

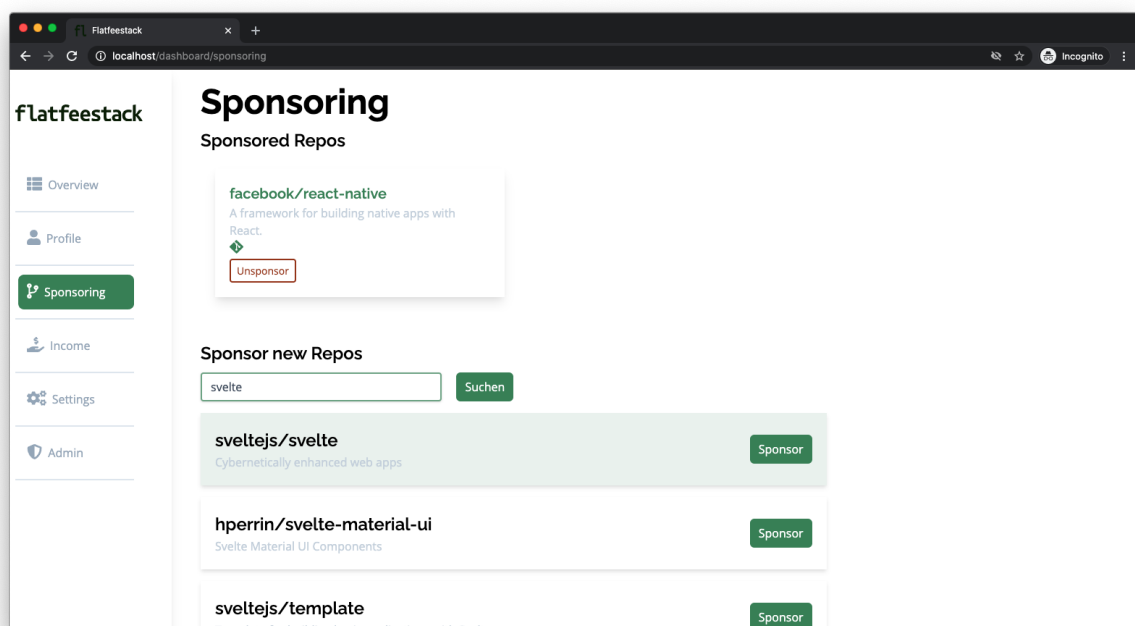


Figure 5.4: Selection of repositories to sponsor

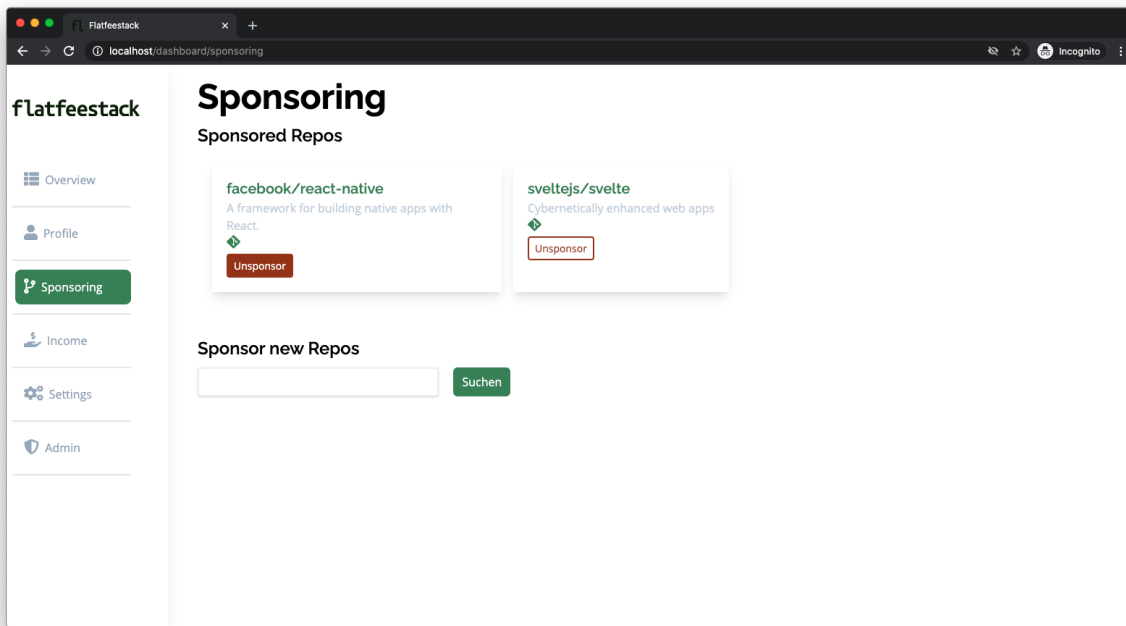


Figure 5.5: Unsponsoring repositories

After a successful payment with an active subscription, a user can start to sponsor repositories (*cf.* Figure 5.4) under the *Sponsoring* tab in the dashboard. The selection of the projects happens independently of the payment and can be changed at any time. Figure 5.5 shows how sponsored projects can be unsponsored with a single click.

While the user interactions to sponsor projects are minimal, this initial version could be improved by providing feedback regarding the actual value that each project receives from a user. Additionally, the yearly payment is recommended and highlighted, but the user can not see the direct consequences on the payment fees. In the sense of transparency (as with crypto payments), a breakdown of a user’s sponsor amount should be displayed. For example, this could be represented in a pie chart which shows the division of the sponsor amount into payment fees, platform fees, and the amount that has been sponsored to each project (which might differ based on the point in time a user has started to sponsor a project).

5.2 Use Case #2 - Developer

From a developer’s point of view, only a few steps are required to receive money. It has to be noted that a developer who wants to receive money uses the same registration and dashboard as the sponsors. In fact, there is no clear separation between sponsor and developer accounts, which theoretically makes it possible to sponsor and receive money with the same account. However, the payment flow is not optimized towards this use case. If a developer had a monthly sponsor income higher than his/her monthly sponsor amount, the monthly credit card payment could be omitted to reduce payment fees. Taking away

the sponsor amount directly from the crypto balance of a user is not supported by the presented architecture. Thus, this use case mainly focuses on developers who are not sponsors at the same time.

Figure 5.6 shows the *Income* tab of the donation platform dashboard. To receive money, a user has to add (and verify) his git email addresses. The number of git email addresses is not limited, which means that multiple git accounts can be combined into one donation platform account. As soon as a git email address is registered, balances for this account (if not empty) are accumulated. Additionally, the address of a user's crypto account or wallet must be registered (*cf.* Figure 5.7). For users who have never used cryptocurrencies, this poses an additional configuration effort outside the donation platform. However, the effort to create a new Ethereum account can be compared to the effort of a full identification (KYC) that is required for other donation platforms.

The *Request funds* button (*cf.* Figure 5.8) uses Web3 [50] to claim the user's balance by calling the `release()` function of the smart contract (*cf.* Section 4.2.2). This requires the user to have a Web3 enabled browser or an installed extension like MetaMask [?]. By clicking on the button, a transaction to the smart contract is created that can be reviewed and signed in a Web3 modal (*e.g.*, Metamask). A user is, therefore, able to claim his balance at any time from the smart contract. The MetaMask extension even presents an error if a user tries to claim a non-existing balance from the smart contract. After the release function has been called, the user receives his funds (*cf.* Figure 5.9). A drawback is that the user needs to have ether in order to pay the transaction cost to claim his balances. This is further discussed in subsection 5.5.

Figure 5.7 only shows the bare minimum of functionality that is needed to facilitate the described payment flow. For a production environment, there is again feedback missing for the user. To make a considered decision when and how often to request funds, a user needs to see his balance in the smart contract and the sum of open payouts. While the balance in the smart contract can be queried by everyone, it is a task that an average platform user cannot be expected to perform.

5.3 Administration

While most of the payment related tasks are automated by the scheduler (*c.f.*, Section 4.2.4) or directly triggered by the user, there remains one task that needs to be done manually: The conversion from fiat to crypto. While this task could also be automated, it would exceed the scope of this thesis as it requires a bank account with an API and algorithmic trading. Instead, the platform provides an administration section where exchange rates can be manually added. This enables the use case, where one person is in charge of trading cryptocurrencies and can flexibly insert the exchange rate once the required fiat amount has been exchanged at a favorable time. It has to be noted that the exchange rate can only be filled in once the required balance is in the smart contract owner's account. This is because the scheduler checks hourly if there are open payouts, and a payout is initiated as soon as the associated exchange request is completed.

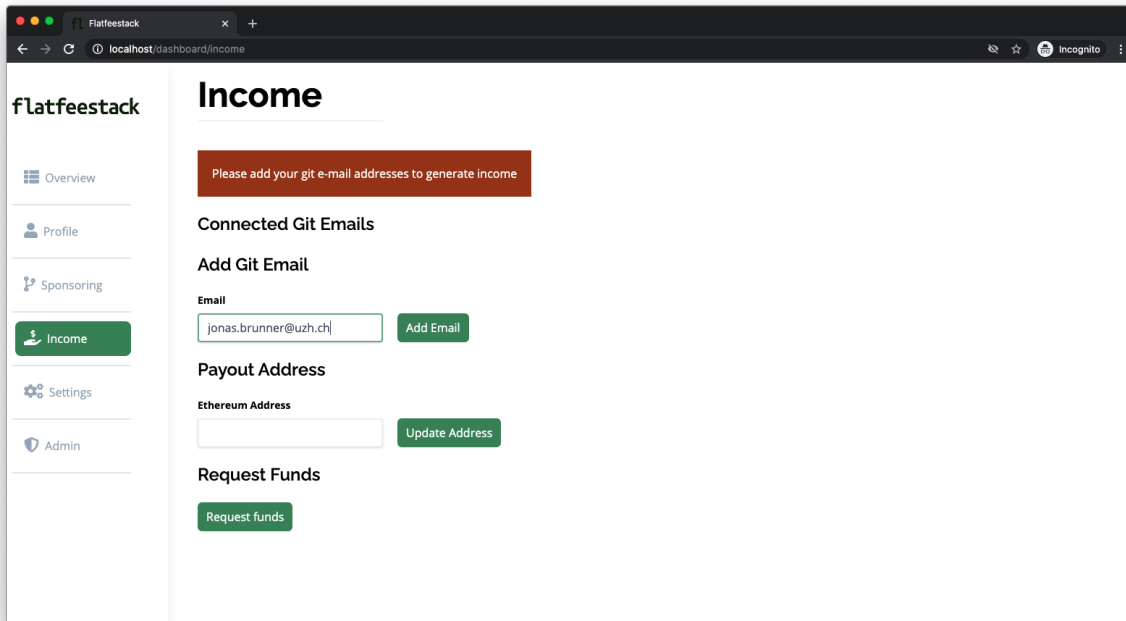


Figure 5.6: Adding git emails

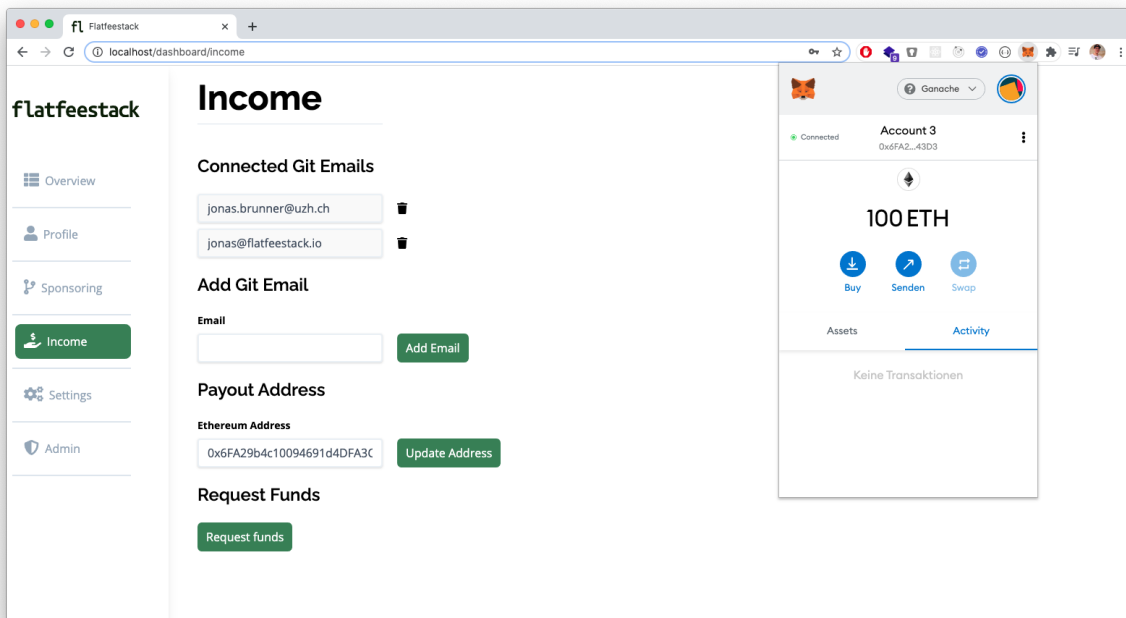


Figure 5.7: Configuring payout address

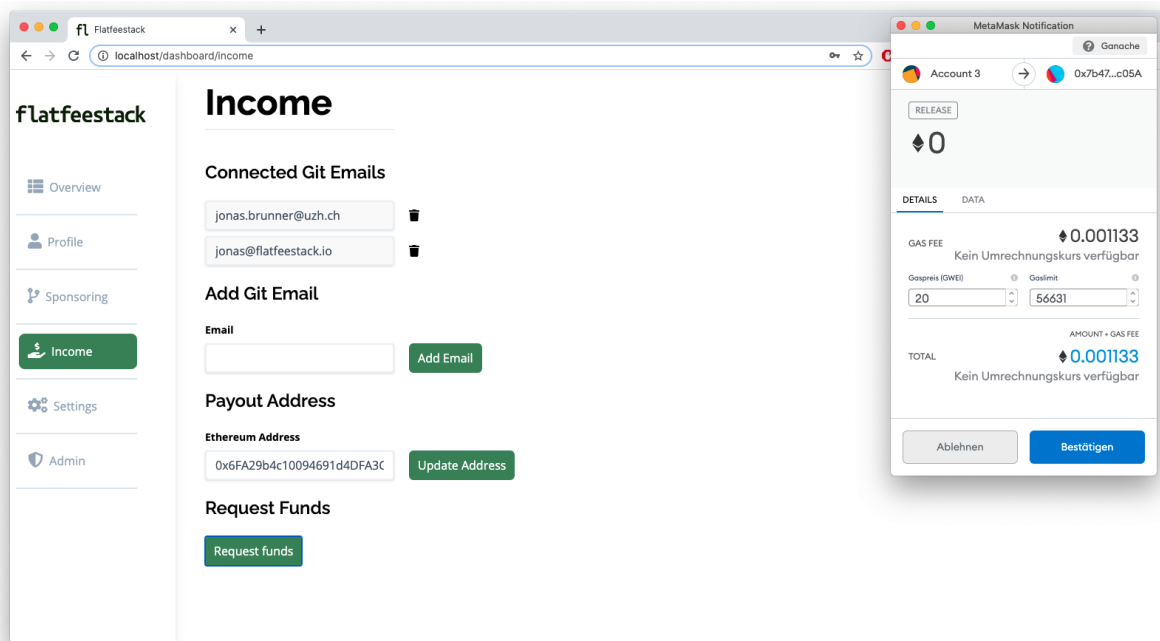


Figure 5.8: Smart contract release function call

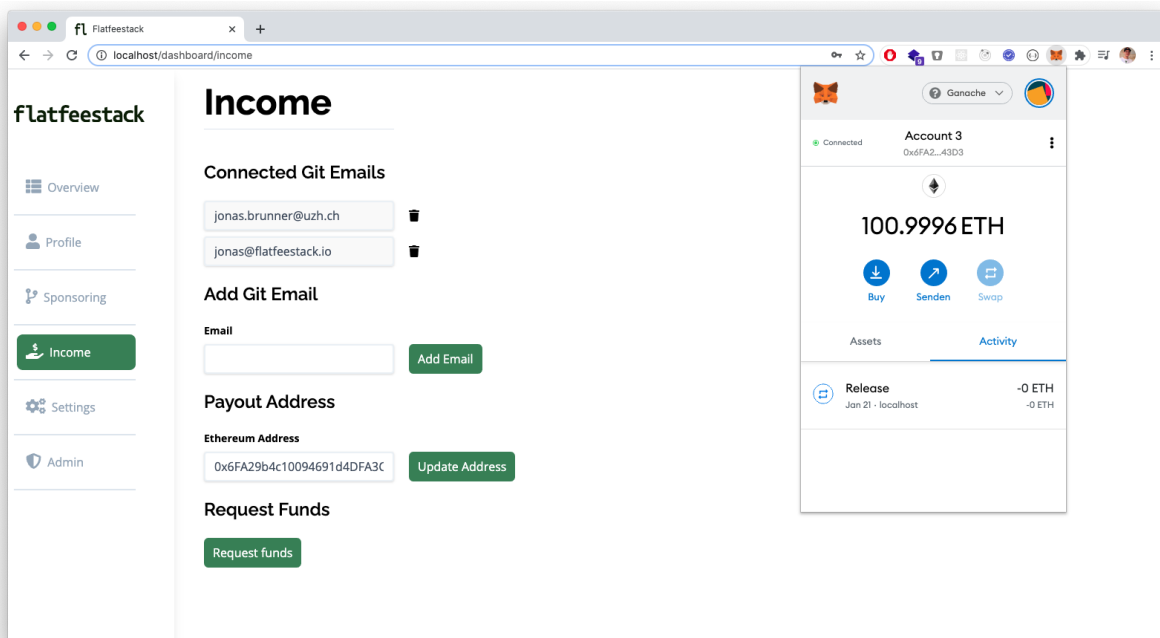


Figure 5.9: New balance after release function has been called

Figure 5.10 shows the *Admin* tab in the donation platform dashboard, which is only available to users with administrator privileges. The weekly generated exchange requests are displayed as cards and show how much fiat money has to be exchanged to which cryptocurrency. Once an exchange rate has been filled in, the exchange request is completed and turns green. The completion date is then shown on the top right of the exchange request card. While figure 5.10 only shows one open exchange request, it is also possible for multiple requests at once to be open.

For example, if the Ethereum price is extremely volatile in one week, the administrator might decide to exchange the fiat money at a later point in time to avoid developers losing money because of bad exchange rates. However, this is a trade-off between instant payments and speculation on favorable exchange rates. This thesis does not aim at recommending one variant over the other but shows the complexity of decision-making for this manual administration task.

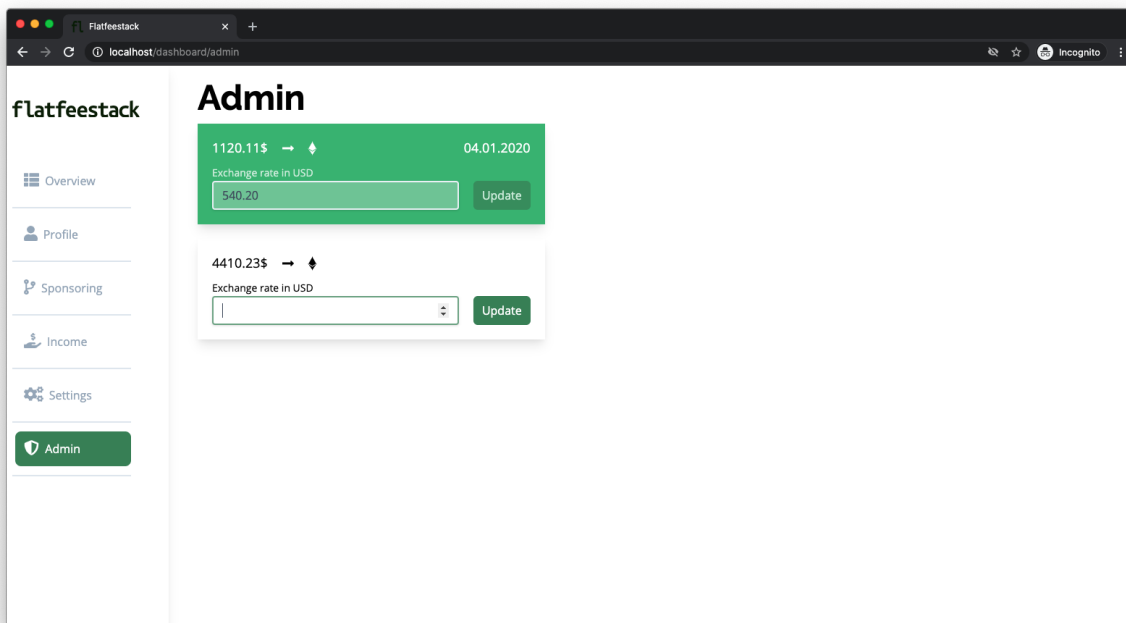


Figure 5.10: Exchange requests

5.4 Cost Evaluation

The cost evaluation of the presented payment flow mainly focuses on the payout. The fee structure of online payments has already been discussed in Section 2.1 and can only be optimized by larger payment intervals and volume discounts.

5.4.1 Scenario

The cost calculation is based on the following scenario:

- There exist 1000 registered sponsors on the platform with a monthly sponsor amount of 10\$.
- Each sponsor supports on average 10 repositories. It is assumed that 3 of the repositories are unique to the sponsor (*i.e.*, not sponsored by anyone else). The remaining 7 repositories are also sponsored by others. To account for the overlap of sponsored repos, it is assumed that there exists a pool of 1000 popular repositories which contains all repositories that are sponsored by more than one sponsor.
- For each repository, an average of 5 active contributors is assumed. This includes popular repositories with hundreds of contributors and smaller open source projects that are only maintained by an individual.
- Out of all contributors, it is assumed that 20% are registered on the platform with a valid payout address.
- Out of all registered contributors, it is assumed that 10% of the contributors receive a payout for the first time.

Based on the above scenario, 4000 unique repositories are sponsored each month, which results in a total of 20'000 contributors. With the assumption that every fifth contributor is registered on the platform, 4000 contributors receive a payout. This means that in the above scenario, each sponsor generates an average of 4 payouts per month. In the following sections, this scenario is applied to two different blockchains.

5.4.2 Ethereum

Prices of cryptocurrencies often fluctuate. Only in the last two months of 2020, ETH price has risen by 90.7%. To compensate for short-term price fluctuations, all calculations are based on average values over the entire year of 2020 (*cf.* Table 5.1).

Table 5.1: Average ETH prices for 2020 [53]

ETH average values 2020	
Average ETH gas price	60,72505376 Gwei
Average ETH price	335.46\$

The following calculations are based on real transactions that have been executed on a local Ethereum blockchain using Ganache [54]. The measured gas consumption (*cf.* Tables 5.2, 5.3 and 5.4) is afterwards combined with the average gas price of the Ethereum mainnet. In the case of Ethereum, two different implementations are taken into account: A pull payment process using a smart contract (*cf.* subsection 4.2.2) and direct transactions.

The smart contract costs are divided into two parts: The "fill" of the smart contract by adding multiple balances at once and the transaction costs to claim the funds from the smart contract.

Smart contract - Fill

The gas consumption of the fill method depends on the number of balances and addresses that are updated. Because new addresses require additional storage, the first balance update of an address is more expensive than the subsequent ones. Additionally, there exists a fixed gas consumption per invocation, which means that the gas consumption per added balance can be decreased by including more balances in one invocation. It has to be noted that the maximum number of addresses that can be updated is bound by the gas limit of an Ethereum block, which is currently at around 12 million [55]. This limits the number of new addresses to 551 and the number of existing addresses to 1782 per invocation.

Table 5.2: Smart contract - costs per fill invocation

Smart contract - Fill	Gas consumption	Cost Gwei	Cost \$
Fix cost per Fill invocation	23300	1'414'893	0,47
Update new address	21'700	1'317'733	0,44
Update existing address	6720	408'072	0,13

Based on the above scenario, each user causes four payouts per month, with 10% of the payouts being to new addresses. The payout fees per sponsor can therefore be quantified as follows:

$$4 \times \left(\frac{9}{10} \times 0.13\$ + \frac{1}{10} \times 0.43\$ \right) = 0.64\$ \quad (5.1)$$

This equals 6.4% of the monthly sponsor amount (10\$). The payment fees of PSPs account for another 5.9% of the 10\$ sponsor amount (for monthly payment interval). In total, 12.3% of a sponsor's monthly amount is consumed for payout and payment fees using a smart contract with the pull payment approach.

Smart contract - Release

The release function of the smart contract is called by creating a transaction. In terms of gas, this transaction is only slightly more expensive than a regular transaction from one account to another. In contrast to the invocation of the fill method, the transaction costs are paid by the receiver of the funds. Thus, the transaction costs of the release method can be considered as payout fee which has to be paid by the contributor. As before, the fee is fixed, and thus, the fee percentage depends on the transaction amount.

For an acceptable payout fee of <5%, the payout amount needs to be greater than 8.80\$. In the above scenario, a total sponsor amount of 10'000\$ is distributed among 4000 contributors. This results in an average monthly payout of 2.50\$ per contributor, which means that the funds can only be claimed with a fee of <5% after about 3.5 months. Certainly, the equal division among all contributors does not reflect a real use case precisely as popular repositories accumulate much higher balances, and not all contributors

contribute equally. Still, it shows that the fanning out of sponsor amounts does not allow instant payouts to all contributors with a reasonable payout fee.

Table 5.3: Smart contract - costs per release invocation

Smart contract - Release	Gas consumption	Cost Gwei	Cost \$
Cost per invocation	21800	1'323'806	0,44

Direct transaction

Direct transactions are the most straightforward solution to payout developers. However, the sender has to pay all transaction costs, and there is no possibility to group multiple transactions into one. With an average transaction cost of 0.43\$ and four caused payouts by each sponsor, payout fees of 1.72\$ are generated, which corresponds to 17.2% of the sponsor amount. Adding the payment fees charged by PSPs, 23.1% of the 10\$ sponsor amount are consumed for payment/payout fees. This shows that single transactions on the Ethereum blockchain are not usable combined with the fanning out of sponsor amounts.

Table 5.4: Ethereum direct transaction costs

Direct transaction	Gas consumption	Cost Gwei	Cost \$
Cost per transaction	21000	1'275'226	0,43

5.4.3 Tezos

The Tezos cost estimation is based on theoretical values from the Tezos documentation [56]. Because no smart contract for Tezos has been written in this thesis, the cost estimation focuses on direct transactions. In general, the transaction fees in the Tezos network are lower than in the Ethereum network. This is because Tezos uses a PoS consensus algorithm where the miners' main income stems from staking coins and not from the transaction fees. The cost estimations for Tezos are also based on the average price over the year 2020 [57]. To calculate the transaction fees, an average XTZ price of 2.52\$ is assumed.

Direct transaction

The minimum transaction fees for a Tezos transaction are shown in Equation 2.1. It has to be noted that this calculation only provides a lower bound and assumes that the bakers use the Tezos default settings. Compared to Ethereum, the Tezos cost per transaction is only a fraction. The four caused payouts by a sponsor in the scenario only accumulate to a total of 0.013\$ in transaction costs. Even if this lower bound is generously doubled, to ensure each transaction is included in a block, the total transaction costs are still only about 0.3% of the monthly sponsor amount. In total, all 4000 monthly payouts generate transaction costs of only 12.83\$, which is about the same as the price of 30 Ethereum transactions.

Table 5.5: Tezos direct transaction costs

Direct transaction	Min. fee XTZ	Min. fee \$
Cost per transaction	0.001 273tezos	0,003207\$

5.5 Discussion

The cost evaluation shows that Ethereum has relatively high transaction fees, and a regular payout (*e.g.*, monthly) is not possible to the majority of the contributors. This is not only caused by the fee structure of Ethereum but also because of the use case, which requires small payments to many different accounts. The use of another blockchain, such as Tezos, could be beneficial in terms of fees. Still, there exists a trade-off between the popularity and transaction costs of a cryptocurrency. While Ethereum is very popular and serves as the basis of many decentralized applications, Tezos is not primarily known as a payment currency and has not gained the same acceptance. In other words, lower transaction fees are not helpful if contributors do not register on the platform because they are paid in an unknown cryptocurrency. With the upgrade to Ethereum 2.0 and the transition to PoS, fees in the Ethereum network could potentially develop more towards the low fees of Tezos, which could make Ethereum a solid choice in the future.

Moreover, the cost evaluation shows that the use of a smart contract is not as cheap as expected. Transferring the balance of an account to the smart contract is even more expensive than sending a direct transaction. Only for subsequent transfers, the use of the smart contract generates fewer fees than a direct transaction. However, this is only valid from a donation platform perspective, as each contributor has to create an additional transaction to claim his funds from the contract. In total, the use of the smart contract causes more fees than direct transactions. Still, the smart contract increases transparency and security from a contributor perspective. Even though the pull payment behavior could also be modeled in the backend of the donation platform, a contributor would not have any guarantee that the donation platform even has the liquidity to pay him at some point. Again, there exists a trade-off between low fees and the trust that has to be put into a donation platform.

Besides the increased transparency, the use of a smart contract also has some disadvantages. In order to claim the funds, a contributor first needs funds to pay the transaction costs. From a contributor perspective, this significantly increases the entry barrier to receive donations. An interesting solution to this problem is the use of the Ethereum Gas Station Network (GSN) [58]. Based on the GSN, a so-called *Paymaster* contract implements the business logic to decide which transactions to pay for. This enables, for example, the use case where the transaction cost of the first claim are paid by the donation platform. However, the use of a smart contract has already been shown to “burn” a significant amount of a sponsor’s donation amount. With the fanning donation architecture, covering contributors’ first transaction fee would thus result in an even larger portion of the donation amount being used for transaction fees.

In general, the cost evaluation of Ethereum has revealed that additional logic is needed to shift all of the payout transaction fees to the recipient of the funds. This means that the transaction costs to populate the smart contract must also be paid by the contributors to avoid a large percentage of the donation amount being consumed by transaction costs. Instead of paying the transaction fees with the sponsor amounts, the fees need to be deducted from each contributor's balance before the fill method of the contract is invoked. Consequently, for small contributor balances, this increases the aggregation duration before the balance in the contract can be updated with reasonable fees. This slightly decreases transparency in a way that some of the balances are only reflected in the smart contracts after multiple months. However, because this only concerns the small balances, it can be neglected.

Without taking into account the smaller popularity, the low transaction fees of the Tezos network enable a good alternative to the Ethereum smart contract. The cost evaluation has shown that direct transactions to all contributors only cause fees of 0.13% of all sponsor amounts. In this regard, there is no need for a smart contract to provide more transparency, as all the balances can regularly be paid out to the contributors. With its low fees, Tezos enables a fanning donation architecture without the need for complicated logic to shift transaction fees to the recipient side. Still, it has to be observed how the fees behave in the future if the Tezos network gains more popularity.

Compared to existing solutions (*cf.* Section 3.1), the transaction fees of both blockchains are still significantly lower than automated bank transfers. For example, Flatlr [39] charges 3\$ per payout and the Stripe connect platform [59] charges fees within the same range (2\$/month per user + (0.25% + 25c) per payout). Additionally, crypto payments omit the need for a complicated onboarding with identity verification, and there is no need to handle refused or returned payouts as all transactions are final. On the payment side, the use of cryptocurrencies is limited because there exists no way to create repeating payments in the form of a subscription.

Chapter 6

Summary and Future Work

In this thesis, the design and implementation of a payment flow for an open source donation platform were presented. The payment flow supports a fanning out donation architecture where each sponsor amount is divided among multiple projects, and the balance of each project is further distributed among the active contributors. With the integration of an already implemented commit analysis engine [6] and the optimistic sponsor approach, where projects without registered contributors can be sponsored, this thesis presents a new approach for OSS donation platforms.

An architecture, which supports the described use case, has been presented together with a fully working prototype. The implementation provides a frontend where users can sponsor projects and contributors can manage their payouts. The payment of sponsors relies on credit card payments. Different payment service providers have been discussed regarding their fee structure, and it has been shown that the chosen payment interval has the greatest effect on payment fees in the case of low-priced subscriptions. The payout to contributors relies on crypto payments.

Further, in this thesis, a pull payment approach using an Ethereum smart contract was presented. The cost evaluation has shown that even when a smart contract is used, the transaction fees must still be fully shifted towards the contributors in order to keep sponsor fees low and enable the fanning out donation architecture. In this regard, the use of Tezos has also been evaluated and shown to be a viable alternative due to its very low fees. However, the combination of fiat payment and crypto payout has shown to be hardly automatable in the scope of this thesis. Therefore, a manual approach to managing the exchange of fiat to crypto has been presented, which in turn provides more flexibility in trading highly volatile cryptocurrencies.

This thesis lays the foundation for a new optimistic donation approach where sponsoring and receiving donations for open source projects can be achieved with minimal actions and a straightforward setup and administration from the contributor side. Future work should address the fee optimization of payouts also for other blockchains (*e.g.*, Tezos), because with the fanning out donation architecture, the ratio of payouts to sponsors is significantly higher than on existing platforms. Additionally, the integration with git hosting platforms could omit some of the required authentication steps and make it even easier to receive

donations. To conclude, it is still to be studied how developers can be incentivized to use such a platform even before a critical mass of sponsors has been achieved.

Bibliography

- [1] M. Stürmer and C. Gauch, “Open Source Studie Schweiz 2018,” 2018.
- [2] Marak, “No more free work from Marak - Pay Me or Fork This.” <https://github.com/Marak/faker.js/issues/1046>.
- [3] Vuejs, “vuejs/vue.” <https://github.com/vuejs/vue>, (Last accessed January 2021).
- [4] Patreon, “Evan is creating Vue.js.” <https://www.patreon.com/evanyou>, (Last accessed January 2021).
- [5] Open Collective, “Vue - Open Collective.” <https://opencollective.com/vuejs>, (Last accessed January 2021).
- [6] C. Falter, “Design and implementation of a commit evaluation engine for an open source donation platform.” B.S. Thesis, Department of Informatics - Communication Systems Group, UZH, Zurich, 2020.
- [7] Visa, “VisaNet.” https://www.visaeurope.ch/de_CH/uber-visa/visanet.html, (Last accessed January 2021).
- [8] Stripe, “Introduction to online payments.” <https://stripe.com/de-ch/guides/introduction-to-online-payments>, (Last accessed December 2020).
- [9] PayPal editorial staff, “Payment Processing 101: Learn how your money gets to you.” 2020. <https://www.paypal.com/us/brc/article/how-online-payments-processing-works>, (Last accessed December 2020).
- [10] Midigator, “The Year in Chargebacks,” 2020. <https://midigator.com/chargeback-report-statistics/>, (Last accessed December 2020).
- [11] Stripe Docs, “Disputes and fraud.” <https://stripe.com/docs/disputes>, (Last accessed December 2020).
- [12] Stripe Docs, “Card authentication and 3D Secure.” <https://stripe.com/docs/payments/3d-secure>, (Last accessed December 2020).
- [13] Stripe, “Online payment processing for internet businesses.” <https://stripe.com/>, (Last accessed January 2021).
- [14] Payrex, “Die einfachste Payment Plattform der Welt.” <https://www.payrex.com/>, (Last accessed December 2020).

- [15] Datatrans, “Advanced Payment Solutions for companies who demand the maximum from their payment solution..” <https://www.datatrans.ch/>, (Last accessed December 2020).
- [16] Liberapay, “FAQ.” <https://liberapay.com/about/faq#maximum-amount>, (Last accessed December 2020).
- [17] Patreon, “Pricing.” <https://www.patreon.com/product/pricing>, (Last accessed January 2021).
- [18] I.-C. Lin and T.-C. Liao, “A survey of blockchain security issues and challenges,” *IJ Network Security*, vol. 19, no. 5, pp. 653–659, 2017.
- [19] I. Bauer, L. Zavolokina, F. Leisibach, and G. Schwabe, “Exploring blockchain value creation: the case of the car ecosystem,” in *Proceedings of the 52nd Hawaii International Conference on System Sciences*, 2019.
- [20] S. Huh, S. Cho, and S. Kim, “Managing IoT devices using blockchain platform,” in *2017 19th international conference on advanced communication technology (ICACT)*, pp. 464–467, IEEE, 2017.
- [21] J. Truby, “Decarbonizing Bitcoin: Law and policy choices for reducing the energy consumption of Blockchain technologies and digital currencies,” *Energy research & social science*, vol. 44, pp. 399–410, 2018.
- [22] V. Buterin, “A Next-Generation Smart Contract and Decentralized Application Platform,” 2013. <https://ethereum.org/en/whitepaper/>, (Last accessed December 2020).
- [23] “Upgrading Ethereum to radical new heights.” <https://ethereum.org/en/eth2/>, (Last accessed December 2020).
- [24] L. Goodman, “Tezos – a self-amending crypto-ledger,” 2014. https://tezos.com/static/white_paper-2dc8c02267a8fb86bd67a108199441bf.pdf, (Last accessed December 2020).
- [25] CoinMarketCap, “Cryptocurrency Prices, Charts And Market Capitalizations.” <https://coinmarketcap.com/>, (Last accessed January 2021).
- [26] Tezos Foundation, “Tezos ecosystem approves Delphi upgrade,” 2020. <https://tezos.foundation/delphi-upgrade/>, (Last accessed December 2020).
- [27] GitHub, “Where the world builds software.” <https://github.com>, (Last accessed November 2020).
- [28] C. Wanstrath, “We Launched - The Github Blog,” 2008. <https://github.blog/2008-04-10-we-launched/>, (Last accessed November 2020).
- [29] A. Bonaccorsi and C. Rossi Lamastra, “Altruistic individuals, selfish firms? The structure of motivation in Open Source software,” *The Structure of Motivation in Open Source Software*, 2003.

- [30] GitHub, “Repository search for public repositories.” <https://github.com/search?q=is:public>, (Last accessed November 2020).
- [31] S. Krishnamurthy, S. Ou, and A. K. Tripathi, “Acceptance of monetary rewards in open source software development,” *Research Policy*, vol. 43, no. 4, pp. 632–644, 2014.
- [32] C. Overney, “Hanging by the thread: an empirical study of donations in open source,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pp. 131–133, 2020.
- [33] PayPal, “Send Money, Pay Online or Set Up a Merchant Account.” <https://paypal.com>, (Last accessed November 2020).
- [34] Open Collective, “Make your community sustainable.” <https://opencollective.com/>, (Last accessed January 2021).
- [35] Issuehunt, “Where open source grows.” <https://issuehunt.io/>, (Last accessed January 2021).
- [36] Tidelift, “A managed open source subscription backed by creators and maintainers.” <https://tidelift.com/>, (Last accessed January 2021).
- [37] Patreon, “Change the way art is valued.” <https://www.patreon.com/>, (Last accessed January 2021).
- [38] Patreon, “Taylor Otwell is creating Laravel.” <https://www.patreon.com/taylorotwell>, (Last accessed January 2021).
- [39] Flattr, “Support creators with one easy subscription or with one-time contributions.” <https://flattr.com/>, (Last accessed December 2020).
- [40] Liberapay, “Liberapay.” <https://liberapay.com/>, (Last accessed January 2021).
- [41] Ko-fi, “Make an Income Doing What You Love.” <https://ko-fi.com/>, (Last accessed January 2021).
- [42] GitHub, “GitHub Sponsors.” <https://github.com/sponsors>, (Last accessed January 2021).
- [43] Flattr, “Account verification.” <https://flattr.com/help/accountverification>, (Last accessed January 2021).
- [44] Flatfeestack, “Flatfeestack/flatfeestack.” <https://github.com/flatfeestack/flatfeestack/tree/demo>, (Last accessed January 2021).
- [45] OpenZeppelin, “SafeMath.sol.” <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol>, (Last accessed January 2021).
- [46] S. Falkon, “The Story of the DAO – Its History and Consequences,” 2017. <https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee>, (Last accessed January 2021).

- [47] Go Co Op, “go-co-op/gocron.” <https://github.com/go-co-op/gocron>, (Last accessed January 2021).
- [48] Gorilla Web Toolkit, “gorilla/mux.” <https://github.com/gorilla/mux>, (Last accessed January 2021).
- [49] Svelte, “Cybernetically enhanced web apps.” <https://svelte.dev/>, (Last accessed January 2021).
- [50] ChainSafe, “ChainSafe/web3.js.” <https://github.com/ChainSafe/web3.js>, (Last accessed January 2021).
- [51] MetaMask, “A crypto wallet gateway to blockchain apps.” <https://metamask.io>, (Last accessed January 2021).
- [52] Caddy, “Caddy 2.” <https://caddyserver.com/v2>, (Last accessed January 2021).
- [53] CoinDesk, “Ethereum Price | ETH Price Index and Live Chart – CoinDesk 20.” <https://www.coindesk.com/price/ethereum>, (Last accessed January 2021).
- [54] Truffle Blockchain Group, “Ganache | Truffle Suite.” <https://www.trufflesuite.com/ganache>, (Last accessed January 2021).
- [55] Etherscan, “Ethereum Average Gas Limit Chart.” <https://etherscan.io/chart/gaslimit>, (Last accessed January 2021).
- [56] Tezos Docs, “Protocol 003_PsddFKi3.” http://doc.tzalpha.net/protocols/003_PsddFKi3.html, (Last accessed January 2021).
- [57] CoinDesk, “Tezos Price | XTZ Price Index and Live Chart – CoinDesk 20.” <https://www.coindesk.com/price/tezos>, (Last accessed January 2021).
- [58] OpenGSN, “Gas Station Network.” <https://opengsn.org/>, (Last accessed January 2021).
- [59] Stripe, “Stripe Connect: Online Marketplace Payments Platform.” <https://stripe.com/connect>, (Last accessed January 2021).

Abbreviations

API	Application Programming Interface
CLI	Command-line Interface
DeFi	Decentralized Finance
GSN	Ethereum Gas Station Network
KYC	Know Your Customer
OSDP	Open Source Donation Platform
OSS	Open Source Software
PSP	Payment Service Provider
PoS	Proof-of-Stake
PoW	Proof-of-Work
SDK	Software Development Kit

List of Figures

2.1	Online payment flow according to [8]	4
3.1	Payment flow comparison of existing platforms	12
4.1	Payment Flow	14
4.2	Example schedule of payment-related tasks	17
4.3	Payout flow with reallocation of expired balances	18
4.4	Architecture	20
4.5	ER-Diagram	22
4.6	Implementation of the architecture	24
4.7	Donation Platform Frontend	36
5.1	Donation Platform Login	38
5.2	Credit card payment	38
5.3	Subscription activation success screen	39
5.4	Selection of repositories to sponsor	39
5.5	Unsponsoring repositories	40
5.6	Adding git emails	42
5.7	Configuring payout address	42
5.8	Smart contract release function call	43
5.9	New balance after release function has been called	43
5.10	Exchange requests	44

List of Tables

2.1	Payment Scenarios	5
2.2	Payment service provider comparison. Values marked with “*” symbol are estimated.	6
4.1	Fees for different payment intervals	16
5.1	Average ETH prices for 2020 [53]	45
5.2	Smart contract - costs per fill invocation	46
5.3	Smart contract - costs per release invocation	47
5.4	Ethereum direct transaction costs	47
5.5	Tezos direct transaction costs	48

Appendix A

Installation Guidelines

A.1 Installation

A.1.1 Prerequisites

The application requires Docker. If you do not have Docker installed on your system, please install it first from here: <https://www.docker.com/get-started>

In order to run the application and additional environment file with sensitive data is needed. Create a `.env` file in the root directory of the application. The form of the `.env` file is shown below. Please add your own secrets to the values denoted with `***`.

You need a stripe account in order to retrieve the stripe secret and the stripe webhook secret.

The smart contract can be deployed using the payout engine. If no `ETH_PK` environment is set, following command will generate a new account and store the private key in `payout/secrets/.key`.

The same command can be used to deploy the smart contract, once the `ETH_PK` is set. After deployment, the contract address is stored in `payout/secrets/.contract` and can be copied to the `.env` file.

```
1 cd payout && make && ENV=local ./payout
```

Listing A.1: Generate ETH account and deploy smart contract

```
1 POSTGRES_PASSWORD=***
2 POSTGRES_USER=***
3 POSTGRES_DB=***
4 STRIPE_SECRET=***
5 STRIPE_WHSEC=***
6 WEBHOOK_CALLBACK_URL=http://scheduler:8080/hooks/analysis-engine
7 ETH_PK=***
8 ETH_CONTRACT=***
```

Listing A.2: Structure of the `.env` file

A.1.2 Starting the project

The application can be built using docker-compose:

```
docker-compose up --build
```

After the build process, the frontend is accessible under `http://localhost:80`

If you want to test credit card payments locally, you need to forward the stripe webhook requests to your local backend. Please install the Stripe CLI and run

```
stripe listen --forward-to localhost/api/hooks/stripe
```

Appendix B

Contents of the CD

- ZIP-file containing the project repository
- ZIP-file containing the source code of the thesis
- PDF of the thesis
- Presentation slides