Aplikacija za poticanje sudionika u akademskom okruženju zasnovana na tehnologiji ulančanih blokova i kriptovalutama

Bunić, Marko

Master's thesis / Diplomski rad

2025

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:116267

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-14**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 711

APLIKACIJA ZA POTICANJE SUDIONIKA U AKADEMSKOM OKRUŽENJU ZASNOVANA NA TEHNOLOGIJI ULANČANIH BLOKOVA I KRIPTOVALUTAMA

Marko Bunić

Zagreb, veljača 2025.

SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 711

APLIKACIJA ZA POTICANJE SUDIONIKA U AKADEMSKOM OKRUŽENJU ZASNOVANA NA TEHNOLOGIJI ULANČANIH BLOKOVA I KRIPTOVALUTAMA

Marko Bunić

Zagreb, veljača 2025.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 30. rujna 2024.

DIPLOMSKI ZADATAK br. 711

Pristupnik:	**Marko Bunić (0036526242)**
Studij:	Računarstvo
Profil:	Računalno inženjerstvo
Mentor:	prof. dr. sc. Josip Knezović

Zadatak: **Aplikacija za poticanje sudionika u akademskom okruženju zasnovana na tehnologiji ulančanih blokova i kriptovalutama**

Opis zadatka:

Potrebno je razviti idejno rješenje i prototip sustava za poticanje (nagrađivanje) sudionika u akademskom okruženju Fakulteta elektrotehnike i računarstva koristeći tehnologiju ulančanih blokova. Analizirati prednosti primjene tehnologije ulančanih blokova u odnosu na klasične modele te implementirati zadani prototip sustava koji će osigurati transparentnost, sigurnost i dostupnost, potičući time motivaciju i dodatno zalaganje sudionika. Istražiti mogućnosti implementacije mehanizma nagrađivanja u obliku kriptovalute kako bi se olakšala razmjena nagrada i interakcija sa stvarnim svijetom. Diplomski rad treba detaljno opisati arhitekturu sustava, istaknuti prednosti korištenja tehnologije ulančanih blokova, te objasniti funkcionalnosti sustava i načine evaluacije njegove uspješnosti (eng. tokenomics). U svrhu potencijalne nadogradnje sustava, potrebno je istražiti ne samo tehničke aspekte implementacije, već i njegovu primjenu te potencijalne koristi za poticanje motivacije i dodatnog zalaganja među svim sudionicima u akademskom okruženju.

Rok za predaju rada: 14. veljače 2025.

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING
MASTER THESIS COMMITTEE

Zagreb, 30 September 2024

MASTER THESIS ASSIGNMENT No. 711

Student:	**Marko Bunić (0036526242)**
Study:	Computing
Profile:	Computer Engineering

Title: **Application for encouraging engagement in the academic environment based on blockchain technology and cryptocurrencies**

Description:

Develop a concept and prototype system for incentivizing participants in the academic environment of the Faculty of Electrical Engineering and Computing using blockchain technology. Analyze the advantages of implementing blockchain technology compared to classical models and implement the specified prototype system that will ensure transparency, security, and availability, thereby fostering motivation and additional engagement among participants. Explore the possibilities of implementing a reward mechanism in the form of cryptocurrency to facilitate the exchange of rewards and interaction with the real world. The thesis should provide a detailed description of the system architecture, highlight the benefits of using blockchain technology, and explain the system's functionalities and methods for evaluating its success (tokenomics). For the purpose of potential system upgrades, it is necessary to explore not only the technical aspects of implementation but also its application and potential benefits for encouraging motivation and additional engagement among all participants in the academic environment.

Issue date:	30 September 2024
Submission date:	14 February 2025

Mentor: Committee Chair:

Professor Josip Knezović, PhD

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 711

Application for encouraging engagement in the academic environment based on blockchain technology and cryptocurrencies

Marko Bunić

Zagreb, February 2025.

I would like to thank my mentor, Prof. Dr. Sc. Josip Knezović, for his guidance and for providing me with the freedom to creatively explore and develop my interests in blockchain technology

I would also like to thank my colleagues for their help in coming up with ideas and brainstorming solutions to challenges encountered during the design and development of the system, Krešimir and Benjamin.

Finally, I would like to thank Matea, my family, and my friends for their support throughout this process.

CONTENTS

1. Motivation

1.1. Blockchain Technology

Blockchain technology is a novel and unique field that introduces a fundamentally new approach to openness and decentralization. The possibilities and opportunities offered by this ecosystem are vast and continually expanding.

To keep pace with technological advancements and prepare for a possible future where blockchain becomes more widely used and integral to everyday systems, it is essential to research how blockchain technology can be implemented to improve existing systems. This involves exploring a variety of innovative ideas.

The core principles of blockchain—openness and decentralization—not only inspire new ideas and possibilities but also demand a shift in traditional system development, requiring additional technical knowledge and specialized security measures.

1.2. FERcoin System

The FERcoin system will be developed to explore the idea of integrating blockchain technology into an academic environment, with the purpose of incentivizing students for academic development, as well as incentivizing students to take an interest in blockchain technology and develop ideas that will help the FERcoin system itself to develop and grow.

The initial project scope is to develop a system that rewards and records students' extra efforts and achievements, thereby motivating them to acquire new knowledge and skills. Additionally, it can serve as a foundation for further development and implementations of blockchain-based innovations.

The first iteration of the project is based on the creation of a cryptocurrency, "FERcoin," which would be issued alongside the allocation of a certain amount of tangible value that the cryptocurrency could be exchanged for. The cryptocurrency itself would not have intrinsic value but rather an indirect value derived from its exchangeability for the allocated tangible assets. The distribution of FERcoin in the project's initial iteration would be tied to students' additional work and achievements in specific courses at FER (Faculty of Electrical

Engineering and Computing). Once the courses or skills to be incentivized with the FERcoin system are selected, a predetermined amount of tangible value or goods would be allocated for distribution to students meeting the established criteria. The FERcoin system would then generate an appropriate amount of FERcoin, backed by the designated tangible assets and goods. The distribution of the allocated FERcoin supply would be entrusted to the instructor of the selected course or skill. The instructor would define the conditions students must meet to be rewarded with FERcoin and determine the amount of FERcoin assigned to fulfilling each condition.

The tangible goods to be used in the first real-world test of the FERcoin system will be coffees in partnership with a selected cafe. For the initial test, a cafe near FER will be chosen, and an agreement will be made where one FERcoin can be exchanged for one cup of coffee. This exchangeability establishes the value of FERcoin itself. The coffees will be bought in advance by the agreement with the cafe, in the same amount as the allocated FERcoin. The already bought coffees can then be received by the students who have earned FERcoin.

The functionalities and uses of FERcoin can be further expanded in many ways, which the technological implementation of the project must make simple and practical. The set of tangible goods and values that FERcoin can be exchanged for can also be broadened. One example of such an expansion would be an agreement with FER's script repository (Cro. "Ferova Skriptarnica"). Upon reaching an agreement, a designated portion of the repository's offerings would be allocated, and an appropriate amount of FERcoin would be created for exchange with the specified products. It is crucial to select an appropriate exchange rate between the amount of FERcoin and a given product to ensure that the exchange rates remain consistent relative to the actual products for which the FERcoin can be exchanged. In addition to potential expansions of FERcoin's functionality, other blockchain capabilities can also be utilized to enhance the system's features and applications. These do not necessarily have to involve the use of cryptocurrency but can include other existing mechanisms provided by blockchain technology, as well as those that may be developed in the future.

2. Introduction to System Development and Infrastructure

The first iteration of the system defines the basic division of FERcoin system users and implements their requirements and functionalities. The emphasis of this iteration is on developing a system that is secure and easy to expand and upgrade while remaining practical and user-friendly for participants and users. The system must be easily scalable, with components designed to be portable and easy to maintain. Additionally, the system's infrastructure is designed to offload the web application and reduce the number of operations and transactions the server must handle on the server side. Instead, most blockchain communication occurs on the client-side, or frontend, following a Fat Client approach. The first iteration consists of a smart contract on a blockchain network, a web application, and a simple database.

The system must be modular and follow the programming principles that make future development and maintenance efficient and straightforward. Depending on the success of the first test in a real-world environment, the system can be scaled further to accommodate a larger user base, introduce new participants and user types, and expand existing functionalities. Additionally the system can be migrated to alternative web application frameworks, databases technologies, or even a different blockchain network for hosting the smart contract. To support these possibilities, system components should be structured for easy expansion, modification, reuse, and migration across different technologies and environments.

When using FERcoin, users retrieve information from the blockchain and submit blockchain transactions signed with their blockchain wallets when interacting with the deployed smart contract. During an exchange of FERcoin for a tangible good, multiple users participate through their respective interfaces. For example, the person purchasing a tangible good submits a blockchain transaction through their interface, while the person delivering the purchased good verifies the blockchain transaction through their own interface. Since the FERcoin system involves multiple interacting components, one of which is the deployed smart contract, it is inherently more complex than a traditional system. To ensure practical usability, all FERcoin users are provided with interfaces that enable simple and efficient usage of their functionalities. Additionally, guides are provided to help users navigate their interfaces and the FERcoin system.

The system's complexity and the number of interacting components increase its potential attack surface, necessitating stricter security measures than those found in conventional web applications, such as systems integrating e-banking access. While systems with integrated e-banking rely on well-established and tested mechanisms, the FERcoin system implements its own security measures for using blockchain to execute transactions. Security is particularly significant due to the financial nature of the system. Beyond the usual risks of data leaks or other vulnerabilities, users manage cryptocurrency with real monetary value. This financial aspect can serve as additional motivation for attackers, amplifying the potential consequences of a breach. The system's infrastructure, as well as the definition of user roles and functionalities, are motivated by ensuring the system's security in both this and subsequent iterations.

The system's web application is implemented to offload the server and reduce the number of actions and transactions it must process. The set of blockchain interactions performed server-side is minimal and limited to interactions involving a smaller group of users, only the professors. The broader group of users, students, as well as cafe workers and administrators, interact with the blockchain by reading from and writing to it via their interface that is fully implemented on the client-side. The student's public interface does not require login, meaning the server is additionally offloaded and only needs to deliver the files needed for the client side usage of the public interface.

3. Selection of Blockchain Technology

In conventional systems using web applications integrated with e-banking, usually a specific set of user data is created and stored for each user. However, FERcoin's use of blockchain technology significantly reduces the amount of stored user data. The largest group of FERcoin system users, students, would typically need to create user accounts under the traditional approach, with accompanying information stored in a database. Blockchain usage eliminates this need by storing all information associated with students directly on the blockchain in a pseudonymous form. Usernames and other details are replaced by blockchain wallet addresses, which cannot be easily correlated to real-world identities. Transactions that would normally be stored in a database are recorded on the blockchain network, eliminating the need for database storage of transactions.

Students use the web application only to access the public students interface which facilitates interaction with the FERcoin smart contract deployed on the blockchain network. All data required for using the public interface is the blockchain wallet with its key pair, the public key (used to derive the blockchain address) and the private key (for signing transactions). Each student stores their blockchain wallet separate from the web application. The recommended storage solution is MetaMask, a software wallet storage, that is used both for storing the blockchain wallets and using them by connecting them to the web application interfaces. Users with sufficient knowledge can review and manage their transactions independently by directly communicating and reading from the blockchain, without relying on the FERcoin web application interface. The student interface is designed to be used with MetaMask and provides a simple and convenient way to interact with the system without requiring additional technical expertise.

3.1. Advantages and Challenges of Blockchain Technology

One challenge of adopting blockchain technology is that every transaction or blockchain data-writing operation has a cost in the native currency of the chosen blockchain network. Thus, it is crucial to select a blockchain network with low transaction fees. Existing modern blockchain networks meet this requirement, offering transaction costs where over a hundred transactions can be covered with the amount of native currency equivalent to one Euro. Networks with such characteristics are ideal for the FERcoin system.

The advantage of using blockchain technology for executing and storing transactions is the significant reduction in database storage requirements and backend operations. The web application server is offloaded, as most interactions are performed between the client side and the blockchain network.

The FERcoin system's performance is inherently tied to the blockchain network hosting its smart contract. While blockchain networks are generally secure and a likelihood of a security incident like a breach is generally low, external factors can influence their usability. One such factor is network congestion.

During periods of high congestion, when a large number of transactions are being processed simultaneously, transaction fees (gas fees) can increase significantly. Although the FERcoin system provides reimbursement for transaction costs with a built-in buffer to handle fluctuations, extreme congestion could still result in fees exceeding the reimbursement amount. In such cases, users would either have to wait until fees decrease or cover the additional transaction costs themselves. This limitation could temporarily restrict FERcoin transactions during peak congestion periods.

The FERcoin system allows users to maintain anonymity by keeping their blockchain addresses private, preventing transactions from being directly linked to their identities. However, users who wish to disclose their identity could have the option to generate a proof of ownership for their blockchain address. While this feature is not a primary focus in the initial implementation of FERcoin, it could become more relevant in future iterations, particularly with the addition of some functionalities discussed in Chapter 11.

3.2. Error Handling and Future Considerations

A limitation of using blockchain for transaction records is the inability to correct some user errors, such as sending FERcoin to the wrong address. While this functionality could be introduced through modifications to the FERcoin smart contract, it may create additional complications. Future system iterations could address such scenarios, but careful consideration must be given to the implications, responsibilities, and requirements this would place on system administrators.

4. Selection of Blockchain Platform

The smart contract will be developed using *Solidity*, a programming language designed for smart contract development, introduced as part of the Ethereum project [12].

Solidity is the most widely used language for writing smart contracts, featuring the best documentation and predefined standards suitable for implementing the functionalities required in this project. Although originally designed for the Ethereum blockchain, many other blockchain platforms also support Solidity for the development and execution of smart contracts.

On Ethereum, after the smart contract is compiled into bytecode, that bytecode is then stored on the blockchain, which is referred to as deploying the smart contract. Each deployed smart contract has its respective *Application Binary Interface* (ABI), which describes the interface for the interaction with the smart contract. When a transaction wants to interact with the smart contract, by calling a function, the bytecode stored on the blockchain is run by the *Ethereum Virtual Machine* (EVM).

Blockchain platforms that support the deployment and execution of smart contracts using the *Ethereum Virtual Machine* are referred to as EVM-compatible platforms. This means that smart contracts developed in Solidity can generally be deployed and executed across these platforms, taking into account platform-specific considerations.

The primary requirements for the blockchain platform on which the system's smart contract will be deployed are as follows:

- It must support smart contracts.
- It must be EVM-compatible.
- It must have low transaction costs.
- It must be a well-known platform with proven stability and security through significant usage.

The potential blockchain platforms that meet these requirements include Binance Smart Chain (BSC), Polygon (Matic), Avalanche (AVAX C-Chain), Fantom, Harmony (ONE), Cronos (CRO), Arbitrum (ARB), Optimism (OP), EVM-compatible Layer 2 solutions on Solana (SOL), and others.

Among these platforms, preference is given to those that:

- Are not developed as part of a centralized cryptocurrency exchange project.
- Feature a native token.
- Are EVM-compatible by default, rather than through additional functionality.
- Are either Layer 1 or Layer 2 platforms, depending on the project iteration.

4.1. Layer 1 vs. Layer 2 Platforms

Layer 1 platforms independently store, process, and validate transactions, offering high security, decentralization, and proven stability through broad adoption. However, they are often limited by scalability challenges and higher transaction costs. Layer 2 solutions, on the other hand, provide faster and cheaper transactions with increased scalability, but rely on Layer 1 platforms for key functionalities. In practice, they are not as secure as Layer 1 platforms due to potential additional vulnerabilities, though this difference is not significant for the needs of this project. Although Layer 2 platforms have advantages, they may also face risks of centralization and liquidity fragmentation, though the latter is not a concern for this project.

Given the initial project iteration requirements, a Layer 2 platform is preferred for its superior scalability, speed, and lower transaction costs. However, in later iterations of the system, a Layer 1 platform might become a more appropriate choice.

4.2. Chosen Blockchain Platform

The chosen platform for this system is **Polygon PoS**, which meets all the defined criteria. Polygon offers extremely low transaction costs and high transaction speeds. While the average time between two blocks on Ethereum is about twelve seconds, it is only 2 seconds on Polygon. The average transaction execution time is below one minute, with "fast transactions" taking approximately 15 seconds, still with a considerably low cost.

Polygon features a native token, MATIC, and is a highly reputable blockchain platform widely used in numerous successful projects. It is a Layer 2 platform that leverages Ethereum as its Layer 1. Polygon uses a "Proof of Stake" (PoS) mechanism for transaction validation, unlike most Layer 1 platforms that utilize "Proof of Work" (PoW). This contributes to Polygons lower transaction costs and scalability.

5. User Roles and System Interactions

The FERcoin system involves multiple user roles, each with specific functionalities and requirements. The system is designed to facilitate seamless user interactions with the system while maintaining security, scalability, and efficiency. This chapter defines the key user roles and their interactions with the system.

5.1. Participants

Participants of the system include:

- **Students**
- **Professors**
- **Cafeteria Staff (Cafe)**
- **Administrator (Admin)**

Each of these users interacts with the system in distinct ways, performing specific actions in regards to their use case in the FERcoin system. Their respective interfaces are developed to provide a simple and convenient way of interacting with the system.

5.2. Functionalities

5.2.1. Students

Students interact with the system through a publicly accessible web interface that requires MetaMask connection for usage and allows them to:

- View the current connected MetaMask account and network
- View the amount of FERcoin that they own.
- View the amount of native currency they own on that blockchain network
- View live network gas price and congestion estimates from blockchain data sources.
- Exchange FERcoin for coffee.

– Send FERcoin to other students.

– View their transaction history of coffees bought with FERcoin in the last 10 hours.

Students do not need to log into the system to access their interface. The student interface is publicly available and designed solely to facilitate the management and usage of FERcoin to all the FERcoin holders. No personal data is stored on the server or within the application, and all student actions are performed client-side. Additionally, a form must be provided for selected students who have earned FERcoin through achievements. This form allows them to anonymously submit their blockchain address to receive their earned FERcoin.

5.2.2. Professors

Professors need to log in to access the professor interface.

Upon login, professors will have the initial page showing all the wallets stored in the system (wallets are created by the admin) associated with that user. After selecting the wallet, another interface will be shown with the display of the amount of the FERcoin allowance that the professor can distribute. On that interface, a form for initiating the action of distribution of FERcoin to selected addresses is also present. This form consists of:

1. A list of student blockchain addresses.
2. The amount of FERcoin each recipient should receive.
3. The password for the selected wallet to decrypt its private key.

Thus, this interface enables the professors to distribute FERcoin to all the students who have acquired the same amount of FERcoin in one action. For example, to distribute all the FERcoin tokens to students who have earned 2 tokens, all of their addresses will be entered in the form and the amount of FERcoin token sent to each participant is set to 2. The professor does not need to do the action of distributing FERcoin for each address individually, but instead, he needs to do the distribution the same number of times as the unique amounts of FERcoin that have been earned by students. (e.g. some students acquired 1 FERcoin, other students have acquired 2 FERcoin and the remaining students have acquired 3 FERcoin, in this case, the professor will need to do only 3 actions of distribution).

To distribute the FERcoin, the professor needs to input the list of addresses that have earned the same amount of FERcoin, set the amount of FERcoin that each participant in that group has earned (e.g 2) and enter the password of the selected wallet for decrypting the private key of the wallet. On the clientside, validation logic is present, which uses regex to check that all the submitted addresses are valid Ethereum addresses and points the invalid Ethereum addresses if there are such. The validation logic also checks if the amount of FERcoin that is being sent is less than or equal to the allowance of FERcoin for the distribution.

When all the data is validated and present, the professor can initialize the transaction. After the action of distributing FERcoin is submitted by the professor, backend logic builds the transactions with all the given addresses and their respective amount of token. Before the transaction is sent, additional validation and a transaction test run is performed by the logic to prevent errors such as insufficient balance, invalid addresses, or exceeding the allowance. If the validation and the test run have passed successfully, the selected wallets' private key is decrypted temporarily with the password and used to sign the transaction which is then sent to the blockchain using the provider in the system. The rest of the process of distribution is handled by the smart contract logic.

The smart contract logic then does the following:

– Distribute FERcoin to the inputted addresses.

– Add a sufficient amount of native currency to cover future transaction costs of spending the FERcoin on buying a coffee or sending the FERcoin to another user for each FERcoin received.

All the logic of processing the transaction and sending it to the blockchain is implemented on the server side. This ensures that the professor does not need to use MetaMask or other form of storing the wallet's key pair associated with his account. Instead the professor only needs to know the password of his account and the password for decrypting the wallet that will be used to execute the transaction.

5.2.3. Administrator

The administrator must log in to access the administrator interface. The administrative interfaces provide tools for:

– Managing the smart contract deployed on the blockchain network.

– Managing the applications configurations.

– Managing user accounts for the web application, including professor and cafeteria accounts, as well as other admin accounts.

– Generating and managing system stored blockchain wallets.

– Retrieving, viewing, and analyzing the transactions of the smart contract deployed on the blockchain.

All of the functionalities associated with managing the deployed smart contract are on the client side and require a connection with MetaMask to use the blockchain interfaces. It is the administrator's responsibility to securely store the key pair of the account that is the owner of the deployed smart contract. This key pair is needed to access the owner-only functions of the deployed smart contract. The administrator interfaces for managing the smart contract, are only meant to ease the usage of viewing the smart contract state and generating the necessary transactions, but the administrator can use other tools and means of analyzing the blockchain data and interacting with the smart contract, independent of the provided interfaces, as the contracts owner accounts key pair is not stored in the system, but rather the administrator stores is securely at his own accord, and can use it to manage the contract in whichever way he finds most practical.

The owner-only function of the smart contract that the administrator has access to are used for:

- Minting new FERcoin tokens which will be distributed to the students
- Approving the addresses that can distribute the FERcoin token(these are professor's wallets, that are generated in the application by the administrator and stored on the server)
- Increasing or decreasing the allowances of the approved addresses
- Adding new Cafes on the blockchain, if a new Cafe joins the project and has the option of purchasing a coffee with FERcoin
- Deleting Cafes from the blockchain
- Changing the amount of native currency that will be sent as reimbursement during the distribution of FERcoin tokens to students
- Withdrawing the native currency from the deployed smart contract balance
- Shutting down the smart contract in the case of master key breach (see chapter Incident response)

Also, it is administrators responsibility to send the needed amount of native currency to the deployed smart contracts balance, which will then be used to send reimbursements along with each distributed FERcoin token, so that the students can cover the gas fees of purchasing a coffee with the FERcoin or sending the FERcoin to another address.

The administrator also has to send the native currency needed to cover the gas fees of the distribution of FERcoin tokens to the wallets that handle the distribution. These wallets are the professor's wallets, which are usually generated and stored on the server.

The rest of the administrators functionalities, revolving around managing the deployed application are:

- Creating and deleting professors, cafes and other administrator accounts
- Generating, deleting, decrypting and re-encrypting the system stored wallets
- Managing the configuration of the blockchain connection and display of blockchain related data on the interfaces, as well as blockchain providers API keys used for blockchain data reading and sending signed transactions from the backend

5.2.4. Cafeteria Staff

Cafeteria staff require a login form to access the cafeteria interface. The cafeteria interface allows for viewing the coffee purchase transactions made with FERcoin. The interface shows only the transactions completed in the last hour. Each transaction has a timestamp and a unique bill identification (Cro. JIR - Jedinstveni identifikator računa) Cafeteria staff do not perform any blockchain transactions; they only review completed transactions.

5.3. Permissions & Access Levels

Each user role has different authentication requirements, data access privileges, and blockchain interaction capabilities, as shown in the Table 5.1 below.

Table 5.1: Access control matrix

Role	**Authentication**	**Data Access**	**Blockchain Write**
Student	None	Ephemeral	Client-side
Professor	Credentials	Institutional	Server-side
Admin	Credentials	Full system	Client-side
Cafeteria	Credentials	Read-only	None

Data access explanation:

– **Ephemeral Data Access**: Student interactions are client-side only, and no data is stored on the server.

– **Institutional Data Access**: Professor interacts only with relevant part of the backend logic for distributing the FERcoin and do not have access to the rest of the system.

– **Full System Access**: Administrators manage the entire system and use the interfaces for managing the smart contract if they have the smart contract owners accounts key pair.

5.3.1. Each user's security considerations

– **Students**: The only part of the system that the students can access is the public interface for FERcoin holders. Since all the operations of this interface are client-side, the students do not have any interaction with the backend logic, rather the server only sends the .HTML file together with the client-side JavaScript, and the rest of the student's actions are done on the client-side, with no connection to backend. The students store their wallet key pair used for holding and using FERcoin on their own accord, independent of the FERcoin application. The recommended way of storing the wallet's key pair and using it for the FERcoin functionalities is by using the MetaMask, but they can store their wallet's key pair differently and communicate with the deployed FERcoin smart contract without using the FERcoin applications' student's interface.

- **Professors**: Each professor's account has one or more associated wallets with it, that are stored in the system and their private keys are encrypted. Private keys are decrypted only temporarily on the server side for transaction signing and are never stored in plaintext on the server. Professors do not even have access to the private key in the encrypted form, upon which the malicious actor with the professor's access could try to brute force.

  If the malicious attacker were to compromise the professor's account, he first has to compromise the account's password. After that, he can view all the wallets associated with that account. Since the professor never has access to any wallet's private key in either encrypted or decrypted form, the wallets can only be used by entering the correct password for decryption of the wallet's private key on the server side. This is another password that has to be compromised as well. If the malicious actor had compromised both the account password and the password of all the associated wallets, then the wallets can be used to send FERcoin to an arbitrary address only if the administrator has already approved the wallet and given it an allowance of distributing the set amount of FERcoin and if that FERcoin is at that moment minted and present in the balance of the master wallet.

- **Cafeteria Staff**: The Cafeteria Staff has an interface that is protected behind a log-in, that only displays the otherwise public data of the blockchain transaction, in a simplified format to ease the reading of the necessary transactions. All of their logic is on the client side. The only sensitive data that the Cafeteria Staff interface has access to is the URL used as a provider for fetching the data from blockchain. This URL often contains an API key, except if some public provider is used instead. If the malicious actor compromised the cafe interface and had access to the API key, he could only use it to spend the limit of the available data traffic on that endpoint.

- **Administrator**: The administrator securely stores the key pair of the account that owns the deployed FERcoin smart contract, ensuring it is independent of and never stored within the FERcoin web application. This key pair is typically the same one used for deploying the FERcoin smart contract, though ownership can be transferred using the same credentials. The deployment of the FERcoin smart contract is a separate process from the deployment of the web application and takes place beforehand.

  This key pair will be referred to as the "master wallet" or "master key pair" throughout this section.

The administrator has full access to system functionalities, including creating and deleting users, generating, decrypting, re-encrypting, and deleting system-stored wallets, modifying blockchain configurations, and managing the deployed smart contract through its interface. However, despite having access to system-stored wallets and professors' wallets, the administrator cannot retrieve their private keys without the decryption password. Additionally, all smart contract management interfaces require a connection to the master wallet, which is never stored within the system. It remains the administrator's responsibility to keep it secure. Without this key pair, the interfaces become unusable, as no blockchain interaction initiated through them can be signed.

This design ensures that even if an attacker gains access to the administrator's account, they cannot control the deployed smart contract's owner-only functions. Similarly, the professor's wallets stored on the system cannot be used without knowing their respective decryption passwords. The administrator cannot access private keys in their encrypted state, preventing any attempt to brute-force decryption. Even if brute-forcing were attempted, the time required would be proportional to the password's strength. If an incident is detected, the administrator can follow the incident response protocol described in Chapter 10 to revoke the wallet's allowance on the blockchain using the master key, rendering the decrypted private key useless.

A decrypted private key from one of the system-stored wallets can only be misused if the administrator has previously approved that wallet for transactions using their key pair. In that case, any FERcoin allocated to the wallet and already minted in the master wallet's balance can be used. However, the administrator can mitigate this risk by generating a new blockchain transaction with the master key to revoke all system-stored wallet approvals as soon as an attack is detected, effectively neutralizing their ability to perform transactions, as outlined in Chapter 10.

6. Selection of Technologies for the Implementation of the First System Iteration

For the development of the web system, technologies were selected to ensure flexibility and simplicity for future developments of the project. These technologies are designed to keep the system scalable, portable, and maintainable.

6.1. Initial Considerations

The initial frameworks considered for web application development included:

- **Node.js**,
- **Django**,
- **Flask**,

Since both Python and JavaScript offer robust blockchain libraries (`web3.py` and `web3.js`), frameworks in these languages were prioritized for better integration.

6.2. Python as the Backend Choice

Between Python and JavaScript for the server-side application, Python was chosen for its extensive libraries, flexibility, and suitability for developing additional functionalities. Python's broader scope and availability of well-documented libraries make it less restrictive for various extensions and integrations with other systems.

6.3. Framework Choice: Django vs. Flask

The final choice was between two popular Python web application frameworks, **Django** and **Flask**. **Flask** is simpler, more flexible, and better suited for smaller projects but lacks built-in solutions and security features that **Django** provides out of the box. Using **Flask** increases the risk of errors, as developers need to manually implement essential features like CSRF protection, which is included by default in **Django**. **Django**, on the other hand, is more complex and imposes predefined decisions about system components. However, **Flask** was chosen for its flexibility and openness, aligning with the system's design principles for simple expansions and potential technology changes. Moreover, implementing features in **Flask**, which are present by default in **Django**, highlights their importance to future developers.

6.4. Frontend Technologies

Some frontend interfaces require more complex client-side functionalities, which are implemented using JavaScript. To facilitate development and improve maintainability, the `Vue.js` framework is used. Vue.js is integrated as a static script file for each interface, providing better structure and organization to the JavaScript code while simplifying the development process. Unlike the usual Vue.js approach, which is used for single-page applications and involves other components like Vue Router, Webpack, or Vite, this system uses Vue only as a lightweight enhancement for specific interfaces. This approach ensures that each interface remains modular, maintainable, and independent without introducing unnecessary dependencies.

The more complex interfaces have all of their respective JavaScript code structured using Vue.js as a static script file, each interface having its own Vue script file. The Vue framework is not used in a traditional approach, for building single page applications and using Vue router or Webpack or Vite or other dependencies, but rather as a static script for one interface to ease code development and structuring of the code.

6.5. Blockchain Libraries

Blockchain libraries are used for implementing blockchain related functionalities. They simplify complex processes like connecting to a blockchain node, querying data, signing transactions, and deploying or interacting with smart contracts.

Used Blockchain Libraries:

– **Client-Side (Frontend)**: The library used to implement the functionalities for communication and interaction with the blockchain on the client side is `web3.js`. This library is widely used, well-documented, and provides all the necessary features for implementing user interactions with the system. It is particularly suitable for execution on the client side (frontend) as it is written in the JavaScript programming language, which is straightforward to use and execute in web browsers.

- **Server-Side (Backend)**: The library used to implement the functionalities for communication and interaction with the blockchain on the server side is `web3.py`. This library is also well-documented, widely used, and compatible with the Python programming language, which implements the server side of the web application. The library used for handling server side generation, management and storage of wallets is Python library `eth_account`.

6.6. Database Technology

The system is designed to minimize backend load by shifting most user functionalities and data storage to the client-side (frontend) and the blockchain. This leverages the selected technologies and simplifies web hosting. Given the simplicity of the system's database needs and the non-taxing nature of **SQLite** for storage and execution, it was selected. **SQLite** is lightweight, does not require a dedicated database server, and is highly compatible with Python, which further supports its use.

6.7. Containerization

Docker is used to containerize and isolate the application during runtime, enhancing:

- Security,
- Portability,
- Ease of deployment.

Docker ensures consistent performance across different environments, simplifies scaling, and eliminates compatibility issues caused by software version mismatches.

7. Development of the Smart Contract

7.1. Smart Contract Design

The FERcoin smart contract is designed to reliably, efficiently, and securely manage FERcoin token usage, capabilities, and administration.

The smart contract is developed using the `Solidity` programming language.

The **ERC20** (Ethereum Request for Comments) Ethereum standard [18] is used for the core FERcoin token logic. This standard defines the logic for fungible tokens, their creation, transferring and some aspects of their usage. Fungibility in this context means that each token is completely equal to other tokens, all tokens have the same functionality, and they are interchangeable.

ERC20 implementation is sourced from OpenZeppelin [15], a widely-used library for secure smart contract development. Along with ERC20, additional libraries used from OpenZeppelin are:

- Ownable.sol – Implements access control, contracts owner definition and allowing only the contract owner to perform certain actions.
- Pausable.sol – Used to implement the shut down function which the contract owner can use in case of certain security breaches.

7.2. Users and Contract Functions

All the functions of the smart contract belong to one of the three groups by access:

- Public functions
- Owner only functions
- Approved addresses functions

Each of these groups is relevant to one of the defined FERcoin users.

The contract security and access control is described in the chapter 9.2.

7.2.1. Public functions

The public functions are the publicly accessible functions for managing FERcoin tokens by FERcoin token holders. These are the functions relevant to the students, which are the intended token holders.

The available functions are `buyCoffee`, used for purchasing a coffee with a FERcoin token and `transfer` function used for transferring a FERcoin token to another address.

7.2.2. Owner-Only Functions

The FERcoin smart contract has an owner, defined as a blockchain address that did the initial deployment of the contract. Ownership can be transferred to another address if needed. The FERcoin administrator is responsible for securely storing the contract owner's key pair.

The FERcoin administrator will manage the FERcoin smart contract using the owner key pair to access owner-only functionalities.

The owner-only functions for administrator management of the contract are used for:

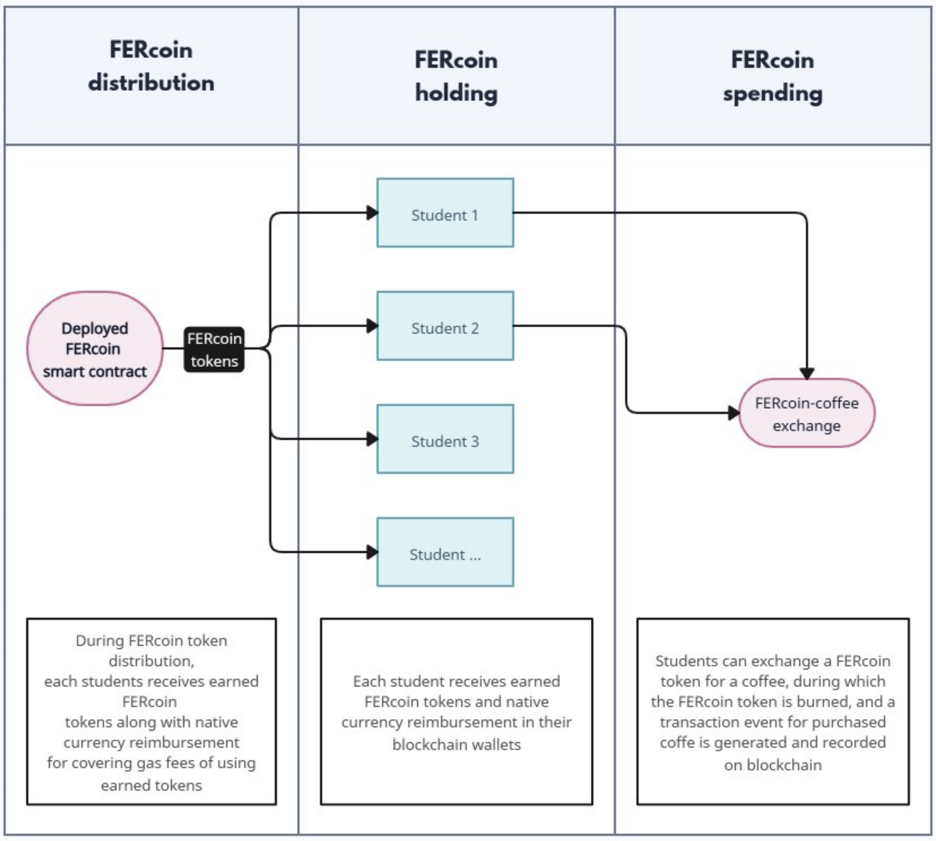
- FERcoin token creation
- Approving addresses and defining setting allowances
- Managing native currency of the contract
- Creating and removing available cafes
- Setting the native currency reimbursement amounts
- Shutting down the deployed smart contract if necessary.
- Transferring ownership of the contract

7.2.3. Approved addresses functions

The contract owner can approve addresses and define their FERcoin allowances for using the functions available only for the approved addresses. These functions are `transferFrom` and `batchTransferFrom`, which are used for FERcoin distribution. The approved addresses are the addresses of the professors' wallets since the professors handle the FERcoin distribution. These wallets are usually stored on the FERcoin application server. The contract owner defines the FERcoin allowance, which is the number of tokens that can be distributed by a certain professor to the students who have earned the FERcoin.

7.3. Token flow

When FERcoin is designated for use in a specific course and the required resource allocation is determined, accompanying FERcoin tokens are created by the contract owner. These tokens remain in the owner's balance until they are distributed to the students. The contract owner approves the professor's wallet and sets an allowance, defining how many FERcoin tokens can be distributed to eligible students. When the blockchain addresses of the students who have earned the FERcoin are collected, the professor distributes the FERcoin tokens from allowance to the student's addresses. Each student receives FERcoin tokens along with a native currency reimbursement to cover gas fees for the transaction of purchasing a coffee or transferring tokens. When a student purchases a coffee with a FERcoin token, that token is burned then, it can no longer be used for any action afterward. The described token flow is shown in Figure 7.1 below.



FERcoin distribution
FERcoin holding
FERcoin spending
Deployed FERcoin smart contract
FERcoin tokens
Student 1
Student 2
Student 3
Student ...
FERcoin-coffee exchange
During FERcoin token distribution, each students receives earned FERcoin tokens along with native currency reimbursement for covering gas fees of using earned tokens
Each student receives earned FERcoin tokens and native currency reimbursement in their blockchain wallets
Students can exchange a FERcoin token for a coffee, during which the FERcoin token is burned, and a transaction event for purchased coffe is generated and recorded on blockchain

Figure 7.1: FERcoin Token Flow

7.4. Reimbursements

Every blockchain transaction incurs gas fees, paid in native currency to miners so that the miners would include the transaction in the new block which will be added to the blockchain.

After the transaction is added to the blockchain it is considered successfully executed. To cover the expenses and ease the usage of FERcoin tokens for students, native currency is distributed alongside earned FERcoin tokens to cover transaction fees of using earned tokens. Native currency used for reimbursements is held by the smart contract. The contract owner funds the smart contract's native currency balance and configures the parameters that are used to calculate the amount of native currency that is used as a reimbursement for using the FERcoin tokens.

The configuration of the reimbursement parameters depends on the blockchain network that hosts the FERcoin smart contract and the gas price of that network during a longer recent time period, which can be a three-month period. These parameters change as the gas price of the network changes to adapt to the changes and remain optimal.

The amount of native currency used as a reimbursement needs to be balanced between two requirements:

- Adequate redundancy of native currency for resistance against the fluctuations in network gas price
- Reducing the amount of native currency given out as reimbursements, to cut the costs and spending of the system

The goal is to use the minimum amount of native currency while ensuring that transaction costs are covered by reimbursement, except during severe congestion.

7.4.1. Gas Price

The gas price is the amount of native currency paid per one gas unit. The network gas price is determined from the gas price of the transactions that were successfully added to the blockchain. The gas price is usually calculated in *Gwei*, which is one billionth of one Ether. Ether here does not mean Ethereum token, as in the native currency of the Ethereum network, but rather as a unit, meaning one whole token. This means that the network's gas price is usually calculated in billionths of the blockchain network's native currency token.

The gas price of any blockchain network fluctuates over time and is determined by the miners and network users. The goal of the miners is to earn as much as possible for mining a block, so they are incentivized to include the transactions that offer to pay the most native currency per gas unit for inclusion in the limited number of block's transaction slots. Since the pending transaction pool is usually larger than the number of transactions that can fit in a single block, the transaction with the highest offered native currency per gas unit from the pool will be chosen to be included in the mined block.

For a transaction to be successfully included in a new block by a miner, its offered gas price must be competitive with other pending transactions in the transaction pool. If the transaction does not offer an adequate reward, it will remain in the pending transaction pool for a longer time period, until all other transactions with higher offered rewards are mined, or if it is low enough it may never be mined.

The network gas price is defined as the amount of native currency offered per one gas unit, which would result in the transaction being included in a new mined block within a reasonable timeframe. The offered reward can be higher, to increase the transaction priority and secure a sooner inclusion of the transaction in a block, or lower if including the transaction in the new block as soon as possible is not as important. Transactions are typically confirmed within minutes, though this can be longer during high network congestion.

To predict the future gas price of the blockchain platform hosting the FERcoin smart contract, the gas price during the recent timeframe period needs to be analyzed.

The blockchain network hosting the deployed FERcoin smart contract in the first iteration is the Polygon network with the *MATIC* native token.

The chosen reimbursement parameters that define the amount of MATIC token given, which is meant to redundantly cover the transaction costs of using FERcoin tokens is two hundred Gwei. This parameter ensures that the students will be able to use the reimbursement to cover the transaction fee cost of using FERcoin tokens always, except during the most significant network congestion periods.

The figure 7.2 below demonstrates the resistance of using two hundred Gwei per gas unit as reimbursement against the network gas fee fluctuations.

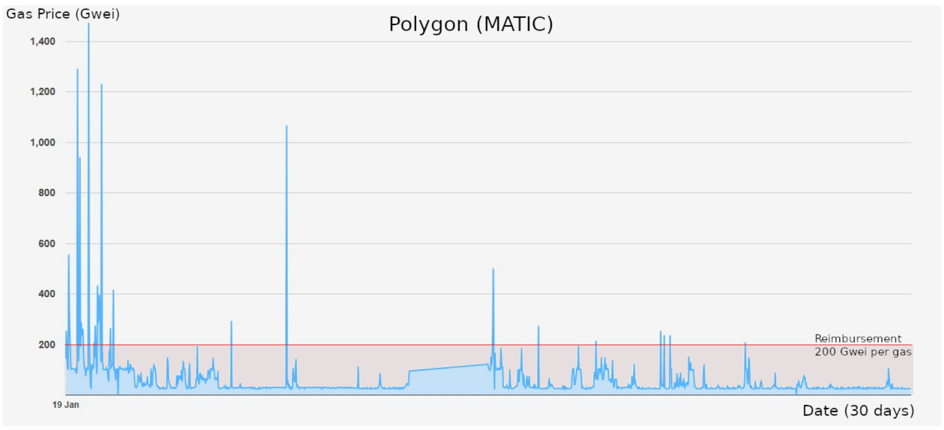


Figure 7.2: Polygon Gas Price Fluctuation

The graph shows gas price fluctuations over one month period of January 2025. The Y-axis represents gas price in Gwei, while the X-axis represents time. The red line at two hundred Gwei per gas represents the reimbursement threshold. Periods below the red line show when the reimbursement would fully cover transaction fees, while periods above the line are the periods of the significant network congestion, during which the reimbursement could not successfully cover the necessary cost of executing the transaction at that moment.

7.4.2. Gas Fee

Transaction gas fee is calculated by multiplying the number of gas units used in the transaction with the gas price of one gas unit:

$$\text{Gas Fee} = \text{Gas Units} \times \text{Gas Price} \tag{7.1}$$

Each executed transaction uses a specific number of gas units determined by the transaction logic. Each transaction's logic consists of opcodes run inside the Ethereum Virtual Machine (EVM). Each opcode has a predefined number of gas units depending on the complexity of the operation. The more opcodes a transaction uses, and the more complex they are, the bigger will the number of used gas units be. Bigger and more complex transactions require more gas units. Each gas unit must be paid to miners for executing the transaction. Thus, the gas fee of the transaction depends on the transaction complexity and the network gas price at that time.

When a transaction interacts with a smart contract, the number of gas units depends on the complexity of the executed function. More complex functions consume more gas.

To minimize gas costs, the FERcoin smart contract optimizes frequently used and computationally intensive functions to reduce the number of gas units used.

The optimization of the most complex smart contract function `batchTransferFrom` is discussed in the section 7.5.3.

The student's reimbursement per FERcoin token is calculated with the number of gas units required to complete a coffee purchase using the `buyCoffee` function.

Testing the `buyCoffee` function execution determined that, with an added redundancy factor, the reimbursement for this function can be calculated with 45,000 gas units.

Together with the chosen native currency per gas for reimbursement which is 200 Gwei, the reimbursement is determined as:

$$\text{Reimbursement} = 45000 \text{ Gas Units} \times 200 \text{ Gwei per Gas Unit} \times 1 \text{ MATIC} \tag{7.2}$$

Since MATIC is the native currency of the Polygon network on which the FERcoin smart contract is deployed, reimbursement is:

$$\text{Reimbursement} = 9,000,000 \text{ MATIC Gwei} = 0.009 \text{ MATIC} \tag{7.3}$$

With the average MATIC price during the year 2024 of 0.61 Euro, the Reimbursement cost given for each FERcoin token in Euros is calculated as:

$$0.009 \text{ MATIC} \times 0.61 \text{ EUR per MATIC} = 0.00549 \text{ EUR } \approx \text{half a Eurocent} \tag{7.4}$$

This means that one Euro would be enough to cover the given reimbursements for 200 FERcoin tokens.

An increase in MATIC's price does not necessarily lead to a proportional increase in reimbursement costs. Often, gas prices (in Gwei) decrease slightly when the native token's value rises, as network participants adjust their transaction fee bidding behavior. This means that the fiat-denominated cost of gas does not always scale directly with MATIC's price. In such cases, reimbursement parameters can be adjusted to maintain cost-effectiveness in response to changes in network condition.

7.5. Smart Contract Implementation

The implementation and code of the smart contract won't be discussed in detail, rather only the chosen parts, the most important and complex ones, will be discussed. The full contract code with `NatSpec` comments documentation is available if needed.

7.5.1. Cafe Management

Each cafe offering coffee purchases with FERcoin is defined in the smart contract by a unique ID and a name.

The contract owner manages the cafes, creating new cafes or removing them.

The `buyCoffee` function takes the unique cafe ID as an argument and stores the information at which cafe was the coffee purchased.

Cafe objects are defined with two variables in the contract, a mapping and an array:

```
/// @dev Mapping of cafe IDs to their names
mapping(uint256 => string) public cafes;

/// @dev Array of registered cafe IDs for enumeration, needed for
    getAllCafes function
uint256[] private cafeIds;
```

The mapping defines the mapping of `uint256` ID of the cafe to the `string` name of the cafe.

The `uint256` array contains the IDs of all available cafes. Since Solidity mappings do not support key enumeration [13], a separate array stores the existing cafe IDs which are used as mapping keys, for efficient retrieval [6].

This way, when some client-side interface must fetch all the available defined cafes, it can first read the array of all defined cafe IDs, and then use the mapping to retrieve the cafe name for each ID, which is the mapping key.

This is implemented in the `getAllCafes` function:

```solidity
/**
 * @notice Get list of all registered cafes
 * @return ids Array of cafe IDs
 * @return names Array of cafe names
 */
function getAllCafes() external view returns (uint[] memory ids, string[]
    memory names) {
    uint256 cafeCount = cafeIds.length;
    string[] memory cafeNames = new string[](cafeCount);

    for (uint256 i = 0; i < cafeCount; i++) {
        cafeNames[i] = cafes[cafeIds[i]];
    }

    return (cafeIds, cafeNames);
}
```

The contract owner manages the defined cafes with the `addCafe` and `removeCafe` functions.

To add a new cafe, the owner specifies an ID and name. The ID is stored in the array and an ID to cafe name mapping pair is then defined in the mapping.

When removing a cafe, the owner provides its ID, which is then removed from the array, and the mapping pair is unset, which reduces the used memory expenditure. The function optimizes gas costs by swapping the removed cafe ID with the last element before popping it, ensuring minimal restructuring overhead.

These two functions are as shown:

```solidity
/**
 * @notice Add new cafe location (owner only)
 * @param spendLocationId Unique identifier for the cafe
 * @param spendLocationName Display name of the cafe
 */
function addCafe(uint256 spendLocationId, string memory spendLocationName
   )
external onlyOwner whenNotPaused {
    require(bytes(spendLocationName).length > 0, "Spend location name
        must not be empty");
    require(bytes(cafes[spendLocationId]).length == 0, "Cafe ID already
        exists");

    _addCafe(spendLocationId, spendLocationName);
}

/// @dev Internal implementation of cafe addition
```

```
function _addCafe(uint256 cafeId, string memory cafeName) private {
    cafes[cafeId] = cafeName;
    cafeIds.push(cafeId);
}

/**
 * @notice Remove cafe location (owner only)
 * @param spendLocationId ID of cafe to remove
 */
function removeCafe(uint256 spendLocationId) external onlyOwner
    whenNotPaused {
    require(bytes(cafes[spendLocationId]).length > 0, "Cafe does not
        exist");

    // Swap cafe ID being deleted with the last array element and pop
        last element from array for deletion
    for (uint256 i = 0; i < cafeIds.length; i++) {
        if (cafeIds[i] == spendLocationId) {
            cafeIds[i] = cafeIds[cafeIds.length - 1];
            cafeIds.pop();
            break;
        }
    }

    delete cafes[spendLocationId];
}
```

7.5.2. Gas Reimbursement Configuration

Previously discussed gas reimbursement configuration is defined in the smart contract using two variables:

```
/// @notice Base gas units estimated for buyCoffee transaction
uint256 public buyCoffeeGasUnits = 45000; // Base transaction gas + logic
    + buffer

/// @notice Gas price in gwei used for reimbursement calculations
uint256 public gasPriceInGweiForReimbursement = 200;
```

the `buyCoffeeGasUnits` variable defines the gas units for `buyCoffee` function used in the reimbursement calculation and the `gasPriceInGweiForReimbursement` defines the Gwei per gas given in reimbursement, as discussed in the section 7.4.

The gas units used in the calculation remain constant, provided the `buyCoffee` function logic remains unchanged. The `gasPriceInGweiForReimbursement` variable is adjusted by the contract owner based on the network gas prices fluctuations to ensure optimal reimbursement configuration.

The contract owner uses the following function to change the value of Gwei par gas in reimbursement:

```
/**
 * @notice Update gas price for reimbursements (owner only)
 * @param newGasPriceInGweiForReimbursement New gas price in gwei
 */
function setGasPriceInGweiForReimbursement(uint256
    newGasPriceInGweiForReimbursement)
external onlyOwner whenNotPaused {
    require(newGasPriceInGweiForReimbursement > 0, "Gas price must be
        greater than zero");
    gasPriceInGweiForReimbursement = newGasPriceInGweiForReimbursement;
}
```

The following public view function retrieves the total reimbursement in Gwei:

```
/**
 * @notice Calculate gas reimbursement cost in gwei sent together with
     each FERcoin
 * @return Reimbursement cost in gwei
 */
function gasReimbursementCostBuyCoffeeGWei() public view  returns (
    uint256) {
    return buyCoffeeGasUnits * gasPriceInGweiForReimbursement;
}
```

Application interfaces and backend logic can use this function to perform various calculations and analysis.

7.5.3. Key functions Implementation

This section focuses on two key functions: `buyCoffee` and `batchTransferFrom`. These functions implement custom logic specific to the FERcoin smart contract, differing from modified ERC20 standard functions.

Buy Coffee Function

This function implements the logic of exchanging a FERcoin token for a coffee.

It accepts two arguments:

– The ID of the cafe where the coffee is purchased

– The unique bill number (Cro. JIR - Jedinstveni identifikator računa)

Before executing the transaction, the function performs a series of checks, that revert the transaction upon fail, to ensure data integrity and prevent invalid transactions:

- Bill number validation - avoids empty records
- Cafe check - prevents purchases at non-existent locations
- Balance check - Implicitly handled by the internal `_burn` function, ensuring that the caller possesses sufficient FERcoin for the transaction

If all checks pass, the function burns the spent FERcoin token, permanently removing it from circulation and emits an event containing the following transaction details:

- the blockchain address of the wallet that has purchased the coffee
- ID of the cafe where coffee was purchased at
- Unique bill number of the bill issued by the cafe

The transaction event timestamp can be retrieved from the blockchain by converting the transaction's block number into a timestamp.

The function code is shown below:

```
/**
 * @notice Purchase coffee using FERcoin
 * @dev Burns COFFEE_PRICE tokens and emits purchase event
 * @param spendLocationId ID of the cafe where coffee is purchased
 * @param billNumber Unique bill identifier from POS system (JIR on
    Croatian)
 */
function buyCoffee(uint256 spendLocationId, string calldata billNumber)
external whenNotPaused  returns (bool) {

    require(bytes(billNumber).length > 0, "Bill number must not be empty"
       );

    string memory spendLocation = cafes[spendLocationId];
    require(bytes(spendLocation).length > 0, "Invalid spend location");

    // internal burn function implements the balance check require
    _burn(msg.sender, COFFEE_PRICE);
    emit CoffeeBought(msg.sender, spendLocation, billNumber);

    return true;
}
```

Batch Transfer FERcoin Distribution

`batchTransferFrom` is the most complex smart contract function. It is used to handle the distribution of multiple FERcoin tokens and native currency reimbursements to students in a single transaction, reducing transaction costs and improving efficiency.

Batch Limit

The function has a strict upper limit of 200 recipients per transaction, defined in the following constant:

```
/// @notice Maximum number of recipients in a single batch transfer
uint256 public constant MAX_BATCH_SIZE = 200;
```

The Ethereum network enforces the maximum gas limit per block of 30 million units, while the target block size is 15 million units [20, 21].

While a single block can include transactions up to the 30 million gas units, the network aims to keep block sizes around the 15 million gas target. If blocks regularly exceed this target, the base fee increases, making transactions more expensive and discouraging excessive gas usage.

When processing the maximum batch size of 200 addresses, the `batchTransferFrom` function consumes just under 9 million gas units.

By keeping gas consumption below the block target, the transaction is more likely to be prioritized by miners, as it allows them to efficiently structure blocks and maximize fee earnings.

If the transaction were closer to the 15 million gas target or beyond, miners might hesitate to include it in a block, preferring to construct blocks with multiple smaller transactions. Keeping the function's gas usage below the block target increases the likelihood of timely inclusion in a block while maintaining network efficiency.

Gas Optimization

Since this is the most complex function in the smart contract as it handles the distribution of both FERcoin tokens and native currency for up to two hundred students in a single transaction, it is the most costly function of the smart contract.

The distribution of FERcoin tokens and reimbursements could be performed without this function completely, which would consist of one transaction of sending FERcoin tokens and one transaction of sending reimbursement for each student, which would in the case of 200 students be 400 transactions. These 400 transactions would have a significant cost of execution because all the transaction gas fees would accumulate.

Instead, the `batchTransferFrom` function is used, which handles both the FERcoin and reimbursement distribution to multiple students in a single transaction to optimize gas consumption and reduce costs. Batch transfers are a well-established gas optimization technique used in various smart contracts [17].

When multiple transfers are performed with one transaction in a single function, there is only one transaction overhead, instead of multiple overheads of all separate transactions which accumulates savings.

Along with this technique for gas-saving optimizations, the following techniques are used as well:

– All the addresses to which the FERcoin is being distributed are used as calldata, meaning they are not saved to memory which leads to gas savings

– External variables are assigned to local variables to minimize redundant storage reads

– The total transaction predicted spendings and allowance check is performed only once outside the loop, instead of separately in each iteration

– All calculations are performed outside the loop, ensuring the loop operates with precomputed values.

– No helper functions used inside the loop, to omit the function call overhead

– Rather than using `transferFrom` inside the loop—which would perform the tokens transfer with `_transfer` and allowance update with `_spendAllowance` in each iteration—only the token transfer is executed inside the loop with internal `_transfer` function, while the allowance is updated only once after exiting the loop, based on the total successfully transferred token amount.

Require checks are not used inside the loop, as a require failure would revert the entire batch transaction, still wasting gas without successful completion of any transfers.

On a failed batch transfer with a particularly large number of distribution addresses, this would be costly.

Instead of using require statements that revert the transaction, a pre-check is performed to identify potential failures. If a failure is detected, the affected logic segment is skipped, and an event is emitted detailing the omitted logic that would otherwise revert the whole transaction.

The administrator can later review all the emitted events, and manually perform the necessary actions to address any skipped transfers.

This approach handles the errors with significantly less gas usage, as a single iteration that would fail can not revert the whole transactions in which a large number of other iterations were successful, and the FERcoin administrator can manually address all the errors that might have occurred.

The loop logic and consequent allowance update of the `batchTransferFrom` function is as follows:

```solidity
/**
* @dev Loop logic
*      require inside the loop would revert the whole transaction
*      which would be costly if it were to happen later on in the loop
*      with 100+ recipients
*      instead the part of the transaction that would fail is omitted
*      and an event signaling the fail is emitted, so the owner can later
    analyze
*      the incident and take the necessary actions
*
*      either the part of the transaction of transfering FERcoin can be
   omitted with an event
*      or the part of the transaction of transfering the native currency
   for reimbursement can be omitted
*/
for (uint256 i = 0; i < recipients.length; i++) {

    if (recipients[i] == address(0) || !isEOA(recipients[i])) {
        emit TransferFailed(recipients[i], "Invalid recipient address -
            is either 0 or not EOA");
        continue;
    }

    _transfer(owner, recipients[i], amount);

    (bool success, ) = recipients[i].call{value: reimbursementWei, gas:
        EOA_TRANSFER_GAS_LIMIT }("");

    if (!success) {
        emit ReimbursementFailed(recipients[i], "Reimbursement failed -
            recipient is an EOA and contract balance was not the issue");
        continue;
    }

    emit FERcoinReceived(recipients[i], amount);
}


uint256 totalTransferred = initialOwnerBalance - balanceOf(owner);
_spendAllowance(owner, msgSender, totalTransferred);
```

Analysis was performed to determine the gas unit savings by using the single transaction `batchTransferFrom` function instead of two transactions per student: one FERccoin token transfer and one reimbursement transfer transaction. The gas units used in the transactions were obtained by running the transactions in the EVM and noting the used gas units.

The results for 1, 10, 50, 100 and 200 distributions are shown in Figure 7.3 below.

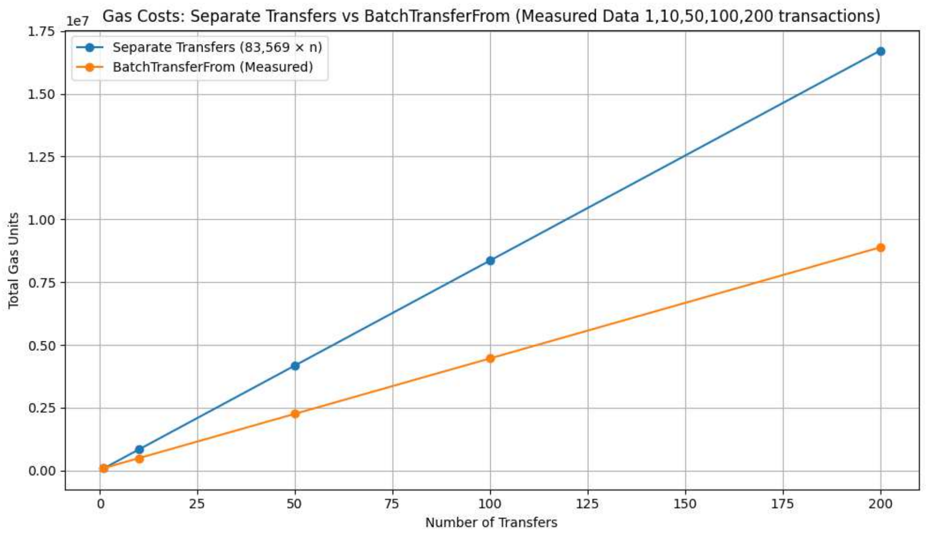


Figure 7.3: Gas Units Usage: Separate Transfers vs Batch transfer

The following Figure 7.4 shows the gas unit savings of using optimized batch transfer, while Table 7.1 below shows the savings in percentages.

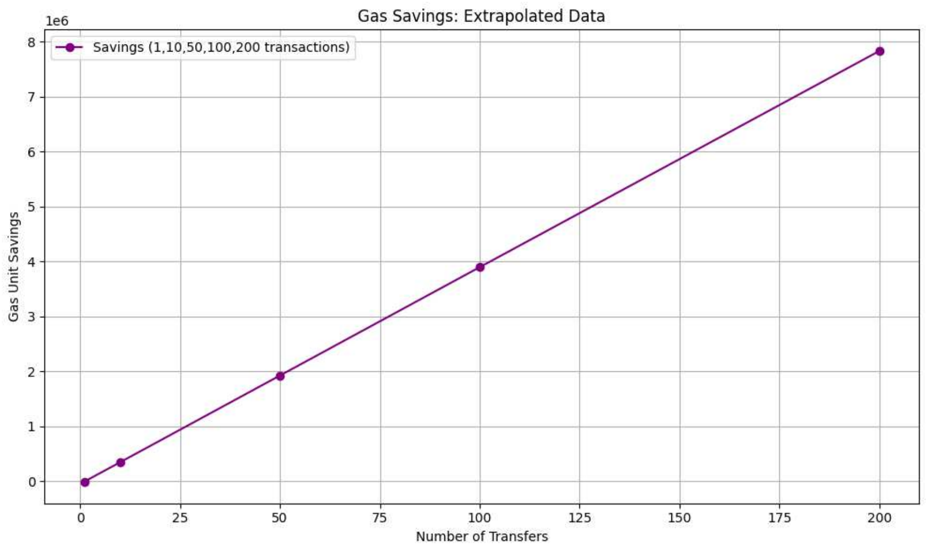


Figure 7.4: Gas Units Savings: Separate Transfers vs Batch transfer

Table 7.1: Gas Units Savings in Percentages

Number of Distribution Addresses	**Gas Units Savings (%)**
1	-12.64
10	41.24
50	46.02
100	46.62
200	46.85

The average Polygon gas price during the year 2024 was 120 Gwei per gas, as calculated with Dune's DuneSQL tool for blockchain analytics.

The gas fee for a maximum batch transfer to 200 recipients is calculated as follows:

$$\text{Max Distribution Gas Fee} = 8,882,632 \text{ Gas Units} * 120 \text{ Gwei per Gas Unit } = 1,066 \text{ MATIC token} \tag{7.5}$$

One MATIC token should be enough to cover the gas fee for distributing FERcoin and reimbursements to 200 students in a single batch transaction.

8. Web application development

This chapter discusses key aspects of the FERcoin web application implementation. For more detailed information, the server-side code is documented using PyDoc, generated from docstrings, and is available alongside the source code. Similarly, client-side JavaScript interface components are documented using JavaScript docstrings.

8.1. Modules and Versions

The server-side logic is implemented in Python using Flask 3.0.3. The Python version used for the project is 3.10, as specified in the Dockerfile:

```
# Lightweight Python image with latest security patches
FROM python:3.10-slim-bullseye
```

All additional modules and their versions are listed in Table 8.1.

Table 8.1: Modules and Versions

Module	**Version**
Flask	3.0.3
flask-session	0.8.0
Flask-Limiter	3.9.2
flask-talisman	1.1.0
Flask-WTF	1.2.2
Werkzeug	3.0.3
SQLAlchemy	2.0.36
Web3	7.4.0
eth-account	0.13.4
pycryptodome	3.21.0
cryptography	39.0.1
Gunicorn	23.0.0
SQLite3	Standard Python 3.10 Package
Secrets	Standard Python 3.10 Package

8.2. Configurations

The application follows a centralized configuration approach, ensuring that all settings related to security, blockchain, session management, and database access are defined in a single place.

Configurations are categorized into different classes, each handling a specific aspect of the application, such as Flask settings, security policies, blockchain interaction, wallet storage, and database management.

These settings are loaded from environment variables, as well as JSON configuration files.

Key configuration components are described in table 8.2:

Table 8.2: Overview of Application Configurations

Configuration	**Description**
AppConfig	Manages core application settings such as debug mode, session lifetime, and base directory paths.
SecurityConfig	Defines security policies, including Content Security Policy (CSP), rate limiting, and endpoint access control.
DatabaseConfig	Specifies the database storage path and manages SQLite configurations.
BlockchainConfig	Handles blockchain network configuration, Web3 provider setup, contract interaction, and gas management.
WalletsConfig	Manages encrypted wallet storage.

8.3. Users Database

The database requirements for the application are minimal due to the adoption of a fat-client architecture. This approach shifts most of the computational logic to the client-side while leveraging blockchain capabilities to offload server-side processing and storage. Consequently, the only requirement for the database is to store authenticated application users, specifically professors, cafes, and administrators.

For this purpose, an SQLite database is sufficient. However, as the project scales and evolves, a more robust database system may be adopted.

8.3.1. Table Definition

The users' data is stored in a SQLite database, initialized using a dedicated script. The schema ensures data integrity and security, with constraints on unique identifiers, secure password storage, and strict role validation.

The table schema is shown in Table 8.3.

Table 8.3: Schema for the Users Table

Column	Type	Explanation
id	INTEGER	Randomly generated 9-digit user ID (Primary Key, Unique). Collision check is implemented.
username	TEXT	Unique username for user authentication.
password	TEXT	Password hashed using bcrypt (uses salt) for secure storage.
role	TEXT	User role; must be one of ADMIN, PROFESSOR, or CAFE.

To ensure proper system setup, an initial administrator account is automatically created during database initialization using environment variables set at container startup. This initial administrator user is intended for bootstrapping access and is expected to create a permanent administrator account before being removed.

8.3.2. SQLAlchemy ORM

The application utilizes SQLAlchemy as an Object Relational Mapper (ORM) to simplify interaction with the SQLite database. SQLAlchemy provides structured table definitions and efficient query execution while ensuring that ORM-level constraints align with database-level constraints.

The primary database model is the `User` model, which represents users in the system.

This model enforces unique usernames, hashed password storage, and strict role validation at the ORM level, in alignment with the database constraints.

The database interfaces—responsible for authentication, user retrieval, and user management—utilize this ORM-based User model as an interface for interacting with the database.

8.3.3. Database Interfaces

To facilitate interaction with the database, three primary interfaces are implemented:

Table 8.4: Overview of Database Interfaces

Interface	**Description**	**Used By**
AuthInterfaceDbUsers	Handles authentication logic, verifying user credentials.	Authentication logic (login)
DbUsersViewInterface	Provides read access to user data, such as retrieving all users or specific users by ID or role.	Authentication logic (access control), Admin panel, Wallet management logic
DbUsersWriteInterface	Manages user creation and deletion in the database.	Admin panel

Database View Interface

The `DbUsersViewInterface` allows retrieval of user data through multiple query methods:

Table 8.5: DbUsersViewInterface Methods

Method	**Description**
`get_all_users()`	Retrieves all users from the database.
`get_all_users_by_role(role)`	Retrieves all users that have a specific role.
`get_user_by_id(user_id)`	Retrieves a user by their unique ID.

Database Write Interface

The `DbUsersWriteInterface` provides methods to create and delete users:

Table 8.6: DbUsersWriteInterface Methods

Method	**Description**
`create_user (username, password, role)`	Creates a new user with a randomly generated 9-digit ID, ensuring uniqueness.
`delete_user (user_id)`	Deletes a user from the database by their ID.

Authentication Interface

The `AuthInterfaceDbUsers` is responsible for user authentication:

Table 8.7: AuthInterfaceDbUsers Methods

Method	**Description**
`authenticate_user (username, password)`	Validates user credentials by checking username and password hash.

These database interfaces enable structured and efficient interaction with the database, ensuring clear separation of concerns for authentication, data retrieval, and user management.

8.4. Server Stored Wallets management

In the FERcoin system, wallets are securely stored on the server as JSON files. Each wallet file contains the wallet's Ethereum address, the encrypted private key, the associated user ID, and the user's username. Importantly, the private key remains encrypted until a transaction requires signing, at which point it is decrypted temporarily. This approach ensures that sensitive key material is protected at rest. The decryption password is not stored on the server; rather, the users store it separately.

To manage these wallets, the system provides interfaces described in Table 8.8.

Table 8.8: Wallet Interfaces

Interface Name	**Description**
`WalletViewInterface`	Enables listing and viewing wallet data without decrypting private keys. Private key is never viewed or listed on any users interface in either encrypted or decrypted format.
`WalletDecryptInterface`	Allows authorized users to load a wallet and decrypt its private key temporarily on the server side for transaction signing.
`WalletWriteInterface`	Manages the generation, re-encryption, and deletion of wallet files. Used by the administrator.
`WalletFetchDataInterface`	Supports interactions with wallet data from blockchain, like fetching allowances.
`WalletTransactionsInterface`	Enables execution of transactions which are signed by the server stored wallets. Currently only used for FERcoin token distribution to students by the professors.

All of these interfaces are used for the server side management and usage of wallets.

WalletViewInterface

Table 8.9: WalletViewInterface Functions

Function	**Description**
`list_all_wallets()`	Retrieves a list of all wallets stored on the server, including Ethereum addresses, usernames, and user IDs.
`list_all_wallets_for_user (user_id)`	Retrieves a list of wallets for a specific user based on their user ID.
`load_wallet_undecrypted (wallet_address)`	Loads wallet data from storage without decrypting the wallet or loading the encrypted private key.

WalletDecryptInterface

Table 8.10: WalletDecryptInterface Functions

Function	**Description**
`load_wallet_for_user (address, password)`	Loads a wallet and decrypts its private key on the server side for authorized users (wallet owner or admin) for further usage.

WalletWriteInterface

Table 8.11: WalletWriteInterface Functions

Function	**Description**
`generate_wallet_for_user (user_id, username, password)`	Generates a new Ethereum wallet, encrypts its private key, and stores it securely.
`reencrypt_wallet (wallet_address, old_password, new_password)`	Re-encrypts a wallet's private key with a new password while maintaining access control.
`delete_wallet_storage (address)`	Deletes a stored wallet file from the server (admin-only operation).

WalletFetchDataInterface

Table 8.12: WalletFetchDataInterface Functions

Function	**Description**
`get_wallet_allowance (wallet_address)`	Fetches the allowance for a given wallet address from the blockchain.

WalletTransactionsInterface

The `WalletTransactionsInterface` includes two different functions for transferring FERcoin tokens:

Table 8.13: WalletTransactionsInterface Functions

Function	**Description**
`batch_send_tokens_from_allowance(transaction_initiator_account, recipient_list, token_amount)`	Sends FERcoin tokens in a batch to multiple recipients, signing the transaction with the initiator's private key. This function is optimized for bulk transfers to minimize gas costs.
`single_send_token_from_allowance(transaction_initiator_account, recipient, token_amount)`	Sends FERcoin tokens to a single recipient, signing the transaction with the initiator's private key. This function is optimized for individual transfers, ensuring lower gas usage compared to batch transfers when sending to a single recipient.

The `single_send_token_from_allowance` function is used when transferring tokens to a single recipient. This function uses a direct `transferFrom` call, ensuring minimal gas usage when only one transaction is required

The `batch_send_tokens_from_allowance` function is used when transferring tokens to multiple recipients. This function calls the optimized smart contract's `batchTransferFrom` function, which consumes significantly less gas compared to executing multiple individual `transferFrom` calls.

8.5. Interface Controllers

Interface controllers are responsible for generating the user interfaces via Flask's Jinja2 templating engine. Their key roles include:

- Rendering dynamic templates with data supplied from the server side.
- Bridging the connection between backend logic and the user interfaces.
- Implementing access controls to ensure that only authenticated users with the proper roles can access each interface.

Table 8.14 summarizes the various interfaces along with their controllers, descriptions, and the types of data and connections they use.

Table 8.14: Interface Controllers and Their Associated Data

Interface	**Description**	**Data Included**
Student Controller	Public interface for student interactions with the deployed FERcoin smart contract. All actions are performed on the client side. There is no interaction with the server.	Blockchain configuration data (e.g., contract address, network details, gas management parameters).
Cafe Controller	Authenticated interface for cafe users. It retrieves deployed smart contract transactions, with all of the logic handled on the client side. Interface does not interact with the server.	Contract address and the URL of the web3 provider (including the API keys used for transaction retrieval).
Professor Controller	Authenticated interface designed for professors to distribute FERcoin tokens to students. Backend logic manages token distribution, while the interface displays users' wallet public keys and blockchain-related wallet information.	Associated users' wallets' public keys from the server-side wallet storage; submission of data for token distribution is handled via backend logic that generates and sends the blockchain transaction.
Admin Controller	Authenticated interface for administrators with multiple components for managing both the application and the deployed smart contracts.	Access to the users database, wallet storage, and application configuration. Also includes smart contract management interfaces that require an owner key pair (which is stored securely and separately from the server).

Figure 8.1 visually represents the interaction between different user interfaces and other system components; the deployed smart contract and the web application server. While the diagram focuses on the interfaces themselves, each of these interfaces is managed by its corresponding controller, as detailed in Table 8.14.

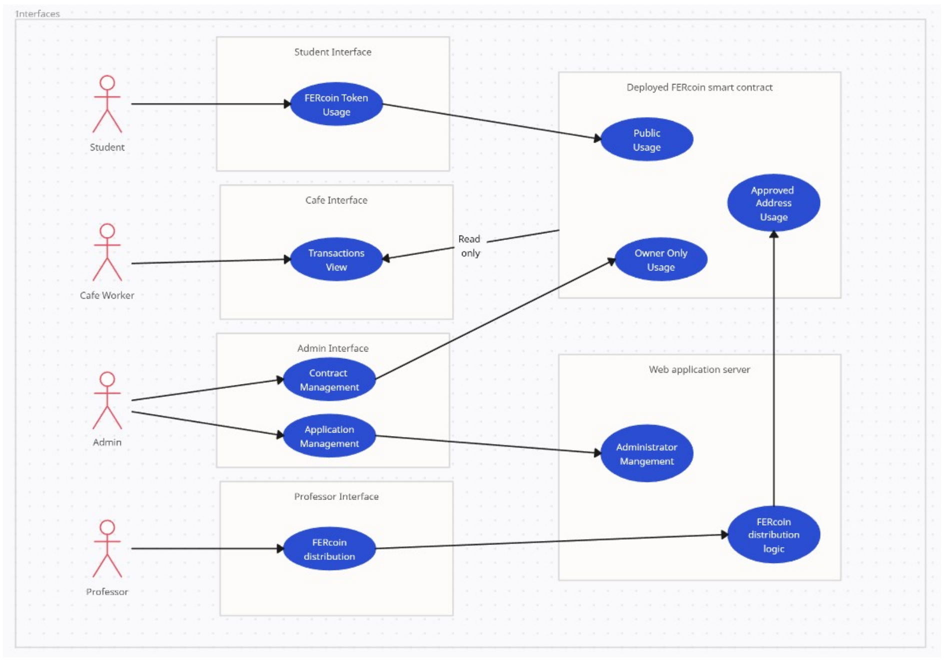


Figure 8.1: Interfaces And System Components Integration

Security implementations and access control of the interface controllers are discussed in Chapter 9.

8.5.1. Forms

All interfaces that interact with the backend logic use provided interface forms submitted to server with POST requests. These forms are validated on the server side and include Cross-Site Request Forgery (CSRF) protection to ensure secure data submissions.

The WT-Forms module is used to implement data validation [10]. CSRF protection is discussed in greater detail in Chapter 9.1.3. For each form that communicates with the backend, a corresponding `FlaskForm` class is defined. This class specifies the required fields, data types, and available choices.

For example, the administrator form for creating a new user is defined as follows:

```
class UserCreationForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired(), Length(min=8)])
    password = PasswordField('Password', validators=[DataRequired(), Length(min=8)])
    role = SelectField(
        'Role',
        choices=[(Role.ADMIN.value, 'Admin'), (Role.PROFESSOR.value, '
            Professor'), (Role.CAFE.value, 'Cafe')],
        validators=[DataRequired()]
    )
    submit = SubmitField('Add User')
```

In this form:

– `username` and `password` must be at least 8 characters long.

– The `role` chosen must be one from the fixed list of roles.

For forms that require dynamically determined options, different approach is used. The following example for selecting a web3 provider in the `BlockchainConnectionForm` illustratares the implemented logic:

```
class BlockchainConnectionForm(FlaskForm):
    provider = SelectField('Provider', choices=[], validators=[
        DataRequired()])
    contract_address = StringField('Contract Address', validators=[
        DataRequired()])
```

In this case, the list of available providers is initially empty. The actual choices are assigned dynamically in the administrator controller when handling the POST request submitting the form:

```
    form = BlockchainConnectionForm()
    form.provider.choices = [(name, name) for name in BlockchainConfig.
        PROVIDERS.keys()]
```

The existing defined providers are displayed on the interface using a separate logic, then the logic for submitted form validation. The choices are determined at the time the form is submitted, ensuring that the chosen provider is one of the existing defined providers on the server. This approach ensures that the data validation is not only implemented on the client side but on the server side as well.

8.5.2. Administrator Interface Controller

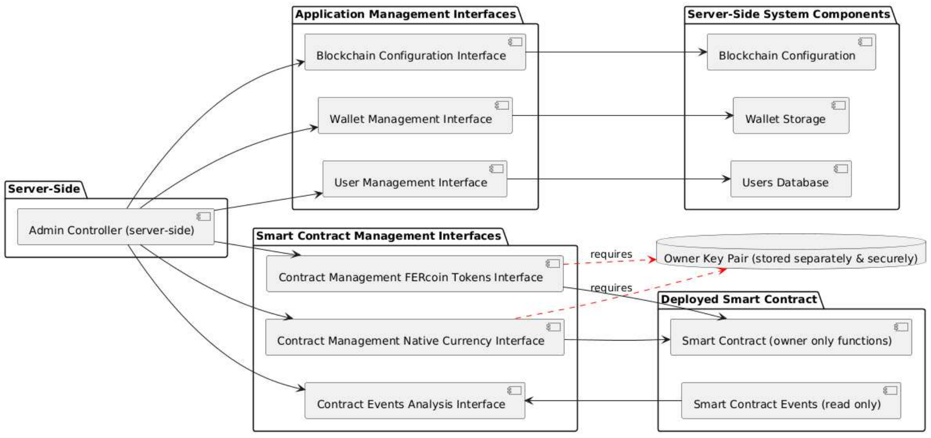
The administrator controller is the most complex controller, as it manages six different interfaces that manage both the application and the deployed smart contract.

Table 8.15 describes all the administrator interfaces and their interactions.

Table 8.15: Administrator Interfaces: Description and Components & Interactions

Interface	**Description**	**Components & Interactions**
Blockchain Configuration	Interface for viewing and updating blockchain connection parameters (provider, contract address, chain ID, gas multipliers, ABI uploads).	Displays and updates the blockchain configuration.
Users Management	Interface for creating, deleting, and listing user accounts with input validation.	Retrieves and manages users from user database using database interfaces.
Wallet Management	Interface for handling wallet operations such as creation, deletion, download (with decryption), and password changes.	Displays and manages wallets from wallet storage using wallet management interfaces.
Smart Contracts Events Analysis	Interface for displaying blockchain transactions and events; requires MetaMask for a Web3 connection.	Fetches deployed smart contracts transactions and events data and details from the blockchain.
FERcoin Token Management	Interface for managing deployed smart contract's FERcoin tokens via smart contract owner functions; requires MetaMask with the owner key pair connection.	Displays FERcoin token-related data from the blockchain and sends owner-level transactions to the deployed smart contract.
Native Currency Management	Interface for managing contract's native currency balance and reimbursement configuration via smart contract owner functions; requires MetaMask with the owner key pair connection.	Displays contracts native currency data from the blockchain and sends owner-level transactions to the deployed smart contract.

The following Figure 8.2 illustrates all the administrator interfaces and their interactions with other system components.



Application Management Interfaces
Blockchain Configuration Interface
Wallet Management Interface
User Management Interface
Server-Side System Components
Blockchain Configuration
Wallet Storage
Users Database
Server-Side
Admin Controller (server-side)
Smart Contract Management Interfaces
Contract Management FERcoin Tokens Interface
Contract Management Native Currency Interface
Contract Events Analysis Interface
requires
requires
Owner Key Pair (stored separately & securely)
Deployed Smart Contract
Smart Contract (owner only functions)
Smart Contract Events (read only)

Figure 8.2: Administrator Interfaces and Interactions

8.6. Interfaces

The following section describes each user's interfaces for interacting with the system.

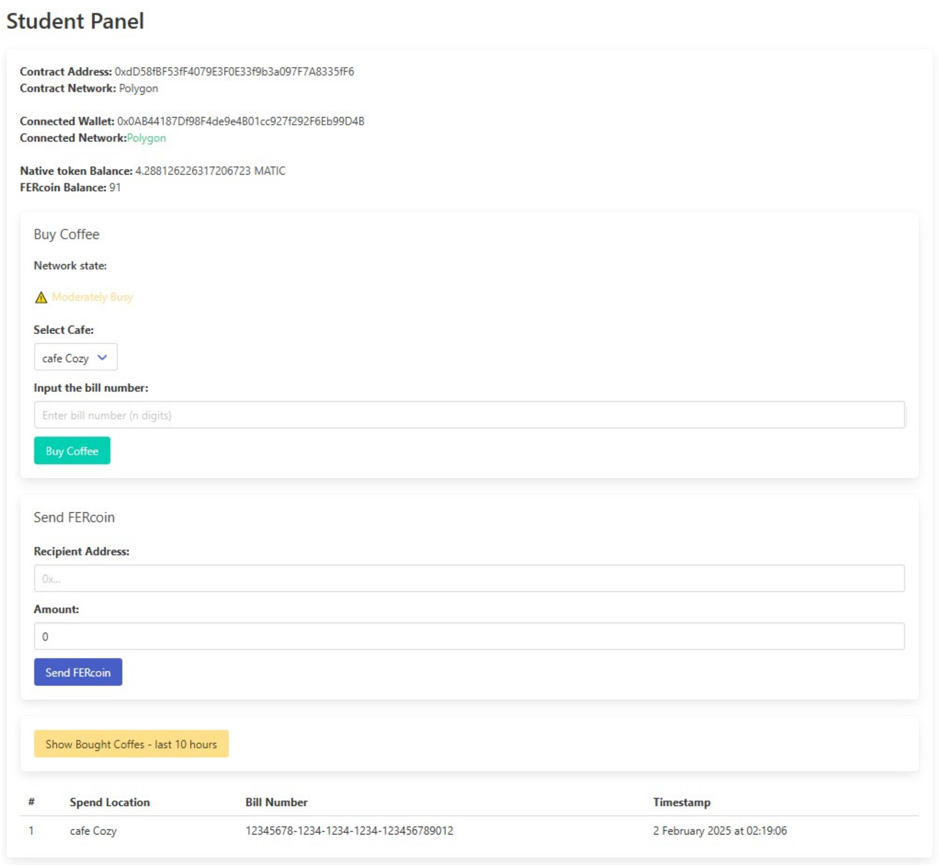
8.6.1. Student Interface

The student interface does not require authentication to access, it is completely public and accessible by anyone. The logic of the interface is completely implemented on the client side. Once the interface is loaded in the browser, it operates without further server interaction.

The interface has the following functionalities:

- Smart contract address and network display
- Connected wallet and network display
- FERcoin and native currency balance display
- Current network state
- Purchase of coffee with FERcoin token
- Sending FERcoin to another address
- Transaction history overview

The interface is shown in Figure 8.3



Student Panel

Contract Address: 0xdD58fBF53fF4079E3F0E33f9b3a097F7A8335fF6
Contract Network: Polygon

Connected Wallet: 0x0AB44187Df98F4de9e4B01cc927f292F6Eb99D4B
Connected Network: Polygon

Native token Balance: 4.288126226317206723 MATIC
FERcoin Balance: 91

Buy Coffee

Network state:

⚠ Moderately Busy

Select Cafe: cafe Cozy

Input the bill number: Enter bill number (n digits)

Send FERcoin

Recipient Address: 0x...

Amount: 0

#	Spend Location	Bill Number	Timestamp
1	cafe Cozy	12345678-1234-1234-1234-123456789012	2 February 2025 at 02:19:06

Figure 8.3: Student Interface

Network State

The network state is calculated to display to students the likelihood and the price of the successful transaction at that moment. To calculate the network state score, three parameters are determined in real time from the gas price API and latest mined block data. These parameters are then normalized, weighted and added together into a single 0-1 value, referred to as the network state score.

The network state is determined by calculating network congestion using three key parameters:

– **Gas Price:** The priority fee for fast transactions relative to a benchmark (130 Gwei).

– **Block Utilization:** The ratio of used gas in the latest block to the block's gas limit.

– **Gas Tier Ratio:** The ratio between fast and standard gas prices, indicating urgency premiums.

Each metric is normalized to a 0-1 scale and assigned a weight to compute an overall congestion score.

Table 8.16: Network Congestion Calculation Parameters

Parameter	Description	Normalization Formula
`Gas Price`	Priority fee for fast transactions, relative to a 130 Gwei reference.	$\frac{\text{Gas Price (fast)}}{130}$
`Block Utilization`	Ratio of used gas to block gas limit in the latest block.	$\frac{\text{Gas Used}}{\text{Gas Limit}}$
`Gas Tier Ratio`	Ratio of fast to standard gas priority fees, capped at 2.5.	$\min\left(\frac{\text{Gas Price (fast)}}{\text{Gas Price (standard)}}, 2.5\right)$

The overall congestion score is computed using weighted metrics:

$$S = 0.6 \times \text{Gas Price} + 0.3 \times \text{Block Utilization} + 0.1 \times \text{Gas Tier Ratio} \tag{8.1}$$

Table 8.17: Network Congestion Score Calculation

Metric	Weight	Final Contribution
`Gas Price`	0.6	$0.6 \times \left(\frac{\text{Gas Price (fast)}}{130}\right)$
`Block Utilization`	0.3	$0.3 \times \left(\frac{\text{Gas Used}}{\text{Gas Limit}}\right)$
`Gas Tier Ratio`	0.1	$0.1 \times \left(\min\left(\frac{\text{Gas Price (fast)}}{\text{Gas Price (standard)}}, 2.5\right)\right)$

The congestion score S determines the network state, mapped into four categories:

Table 8.18: Network Congestion Status Levels

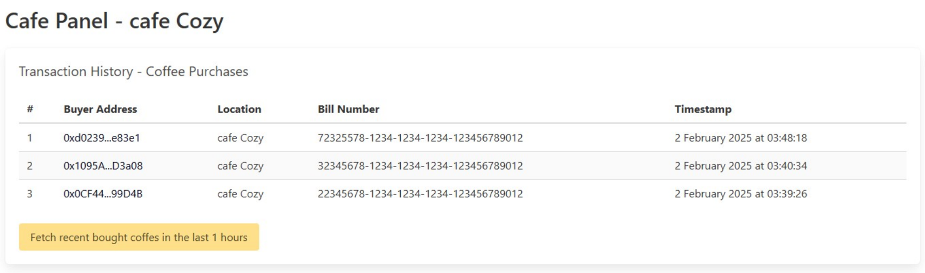
Congestion Level	Score Range	Message
`Good`	$S < 0.4$	Fast, Good Gas Price
`Busy`	$0.4 \leq S < 0.6$	Moderately Busy
`Unstable`	$0.6 \leq S < 0.8$	High Congestion
`Congested`	$S \geq 0.8$	Extremely Congested

The network state dynamically updates based on real-time gas data and latest block statistics, ensuring an accurate representation of congestion.

8.6.2. Cafe Interface

The cafe interface does require authentication to access, as it contains the URL with API key used to retrieve the transaction records from the blockchain. The logic of the interface is completely implemented on the client side, without any connection to the server, after the interface is loaded in the browser. The only function of the interface is fetching the records of the bought coffee transactions. It fetches blockchain transactions using the provided URL with an API key as the web3 provider, without the need to use MetaMask or similar software. Cafe staff use this interface to verify coffee purchases by verifying the unique bill number in the fetched transactions.

The interface is displayed in Figure 8.4.



Cafe Panel - cafe Cozy

Transaction History - Coffee Purchases

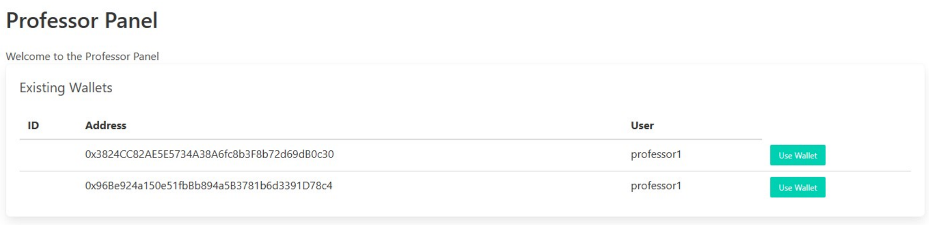
#	Buyer Address	Location	Bill Number	Timestamp
1	0xd0239...e83e1	cafe Cozy	72325578-1234-1234-1234-123456789012	2 February 2025 at 03:48:18
2	0x1095A...D3a08	cafe Cozy	32345678-1234-1234-1234-123456789012	2 February 2025 at 03:40:34
3	0x0CF44...99D4B	cafe Cozy	22345678-1234-1234-1234-123456789012	2 February 2025 at 03:39:26

Figure 8.4: Cafe Interface

8.6.3. Professors interface

The professor interface is made up of two parts, the initial interface for choosing one of the associated user's wallet, and the second interface for using the chosen wallet for FERcoin token distribution.

The initial interface for viewing and choosing the wallet is shown in Figure 8.5



Professor Panel

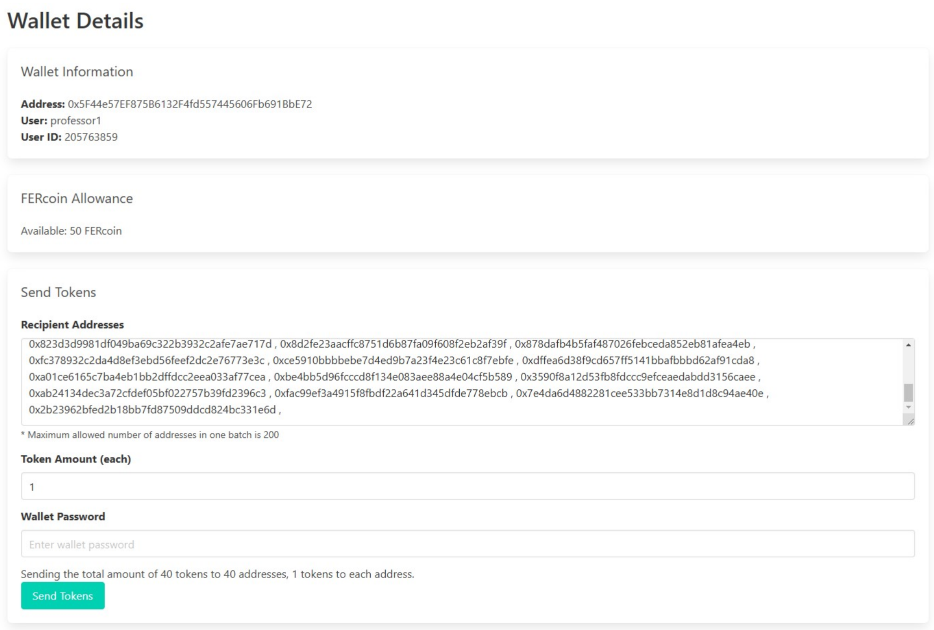
Welcome to the Professor Panel

Existing Wallets

ID	Address	User	
	0x3824CC82AE5E5734A38A6fc8b3F8b72d69dB0c30	professor1	Use Wallet
	0x96Be924a150e51fbBb894a5B3781b6d3391D78c4	professor1	Use Wallet

Figure 8.5: Initial Professor Interface

Once a wallet is selected, the interface for token distribution is displayed, as shown in Figure Figure 8.6



Wallet Details

Wallet Information

Address: 0x5F44e57EF875B6132F4fd557445606Fb691BbE72
User: professor1
User ID: 205763859

FERcoin Allowance

Available: 50 FERcoin

Send Tokens

Recipient Addresses

0x823d3d9981df049ba69c322b3932c2afe7ae717d , 0x8d2fe23aacffc8751d6b87fa09f608f2eb2af39f , 0x878dafb4b5faf487026febceda852eb81afea4eb ,
0xfc378932c2da4d8ef3ebd56feef2dc2e76773e3c , 0xce5910bbbbebe7d4ed9b7a23f4e23c61c8f7ebfe , 0xdffea6d38f9cd657ff5141bbafbbbd62af91cda8 ,
0xa01ce6165c7ba4eb1bb2dffdcc2eea033af77cea , 0xbe4bb5d96fcccd8f134e083aee88a4e04cf5b589 , 0x3590f8a12d53fb8fdccc9efceaedabdd3156caee ,
0xab24134dec3a72cfdef05bf022757b39fd2396c3 , 0xfac99ef3a4915f8fbdf22a641d345dfde778ebcb , 0x7e4da6d4882281cee533bb7314e8d1d8c94ae40e ,
0x2b23962bfed2b18bb7fd87509ddcd824bc331e6d ,

* Maximum allowed number of addresses in one batch is 200

Token Amount (each)

1

Wallet Password

Enter wallet password

Sending the total amount of 40 tokens to 40 addresses, 1 tokens to each address.

Figure 8.6: Professor Distribution Interface

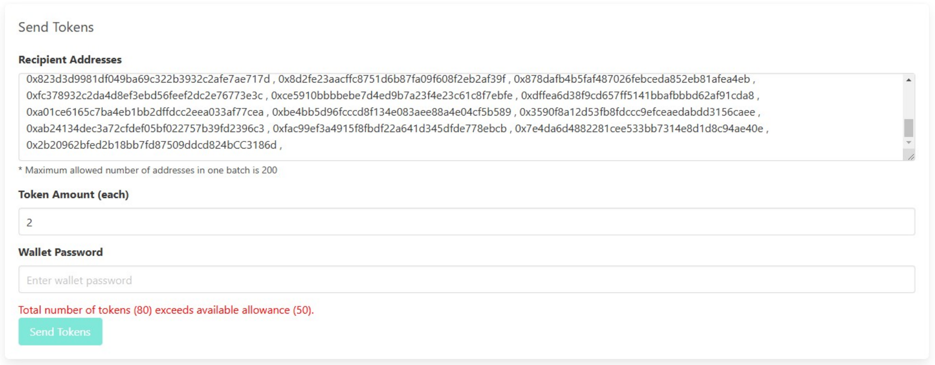
This interface has the following functionalities:

– Display of the wallet address and current user data

– Display of the FERcoin allowance available for token distribution

– A distribution form with input fields for the list of student addresses receiving FERcoin tokens, token amount, and wallet decryption password

Additionally, client-side validation is implemented on this interface to prevent invalid data inputs. The client side ensures that the number of FERcoin tokens and addresses entered does not exceed the available allowance. The error message display of this logic is shown in Figure 8.7



Send Tokens

Recipient Addresses

0x823d3d9981df049ba69c322b3932c2afe7ae717d , 0x8d2fe23aacffc8751d6b87fa09f608f2eb2af39f , 0x878dafb4b5faf487026febceda852eb81afea4eb ,
0xfc378932c2da4d8ef3ebd56feef2dc2e76773e3c , 0xce5910bbbbebe7d4ed9b7a23f4e23c61c8f7ebfe , 0xdffea6d38f9cd657ff5141bbafbbbd62af91cda8 ,
0xa01ce6165c7ba4eb1bb2dffdcc2eea033af77cea , 0xbe4bb5d96fcccd8f134e083aee88a4e04cf5b589 , 0x3590f8a12d53fb8fdccc9efceaedabdd3156caee ,
0xab24134dec3a72cfdef05bf022757b39fd2396c3 , 0xfac99ef3a4915f8fbdf22a641d345dfde778ebcb , 0x7e4da6d4882281cee533bb7314e8d1d8c94ae40e ,
0x2b20962bfed2b18bb7fd87509ddcd824bCC3186d ,

* Maximum allowed number of addresses in one batch is 200

Token Amount (each)

2

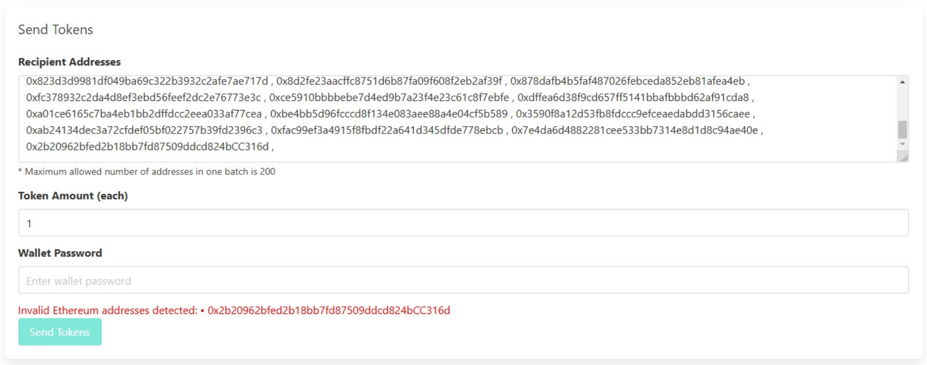
Wallet Password

Enter wallet password

Total number of tokens (80) exceeds available allowance (50).

Figure 8.7: Distribution Input Allowance Validation

The other validation confirms that all the addresses entered in the student address list are in the valid Ethereum address format. The error message display of this validation is shown in Figure 8.8.


Send Tokens

Recipient Addresses

```
0x823d3d9981df049ba69c322b3932c2afe7ae717d , 0x8d2fe23aacffc8751d6b87fa09f608f2eb2af39f , 0x878dafb4b5faf487026febceda852eb81afea4eb ,
0xfc378932c2da4d8ef3ebd56feef2dc2e76773e3c , 0xce5910bbbbebe7d4ed9b7a23f4e23c61c8f7ebfe , 0xdffea6d38f9cd657ff5141bbafbbbd62af91cda8 ,
0xa01ce6165c7ba4eb1bb2dffdcc2eea033af77cea , 0xbe4bb5d96fcccd8f134e083aee88a4e04cf5b589 , 0x3590f8a12d53fb8fdccc9efceaedabdd3156caee ,
0xab24134dec3a72cfdef05bf022757b39fd2396c3 , 0xfac99ef3a4915f8fbdf22a641d345dfde778ebcb , 0x7e4da6d4882281cee533bb7314e8d1d8c94ae40e ,
0x2b20962bfed2b18bb7fd87509ddcd824bCC316d ,
```

* Maximum allowed number of addresses in one batch is 200

Token Amount (each)

1

Wallet Password

Enter wallet password

Invalid Ethereum addresses detected: • 0x2b20962bfed2b18bb7fd87509ddcd824bCC316d

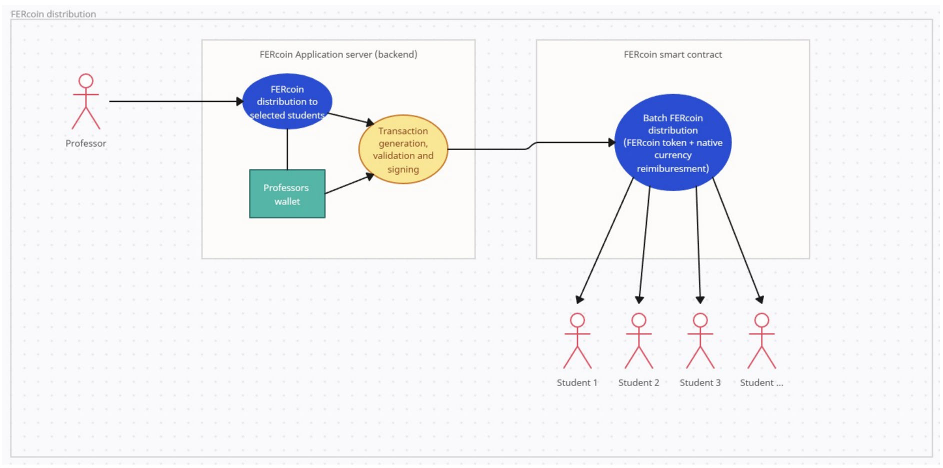
Send Tokens

Figure 8.8: Distribution Address Validation

Further and more robust validation is present on the server-side as well to ensure correct execution.

Token Distribution Process Integration

Figure 8.9 illustrates how the FERcoin distribution process is handled and executed within the broader system architecture.



FERcoin distribution
Professor
FERcoin Application server (backend)
FERcoin distribution to selected students
Transaction generation, validation and signing
Professors wallet
FERcoin smart contract
Batch FERcoin distribution (FERcoin token + native currency reimiburesment)
Student 1
Student 2
Student 3
Student ...

Figure 8.9: FERcoin Distribution Workflow and System Interaction

After the professor submits the student blockchain addresses, the token amount and the wallet decryption password, the professors interface sends the data to the server side. The backend logic validates the data, generates the transaction and performs the test run of the transaction. If all validations pass, the professor's stored wallet is temporarily decrypted using the provided decryption password and used to sign the generated and validated transaction. If FERcoin tokens are distributed to multiple addresses, gas optimized `batchTransferFrom` smart contract function is used, otherwise, if only one address is receiving FERcoin, the `transferFrom` smart contract function is used instead, which uses less gas for single transfers. The transaction is sent by the server to the deployed smart contract, which handles the remaining logic of FERcoin token distribution.

8.6.4. Administrator's Interfaces

The administrator manages the FERcoin system through six different interfaces, three for managing the application and other three for viewing and managing the deployed smart contract. Upon login, the administrator is presented with the initial page (Figure 8.10) to choose one of the six interfaces.

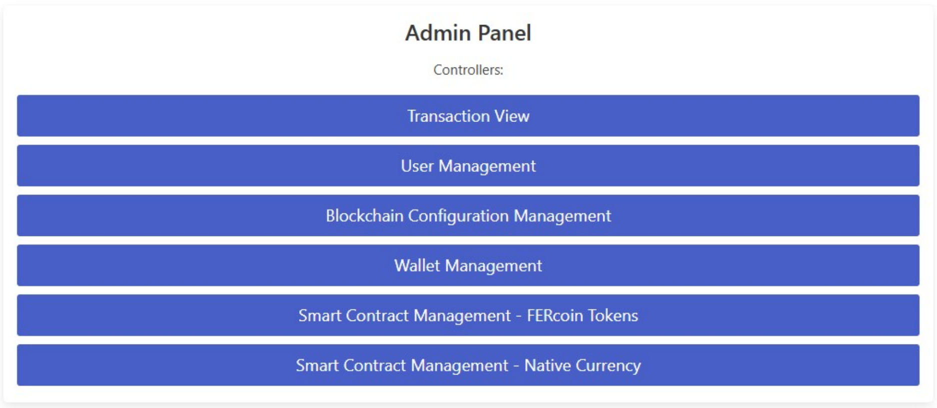
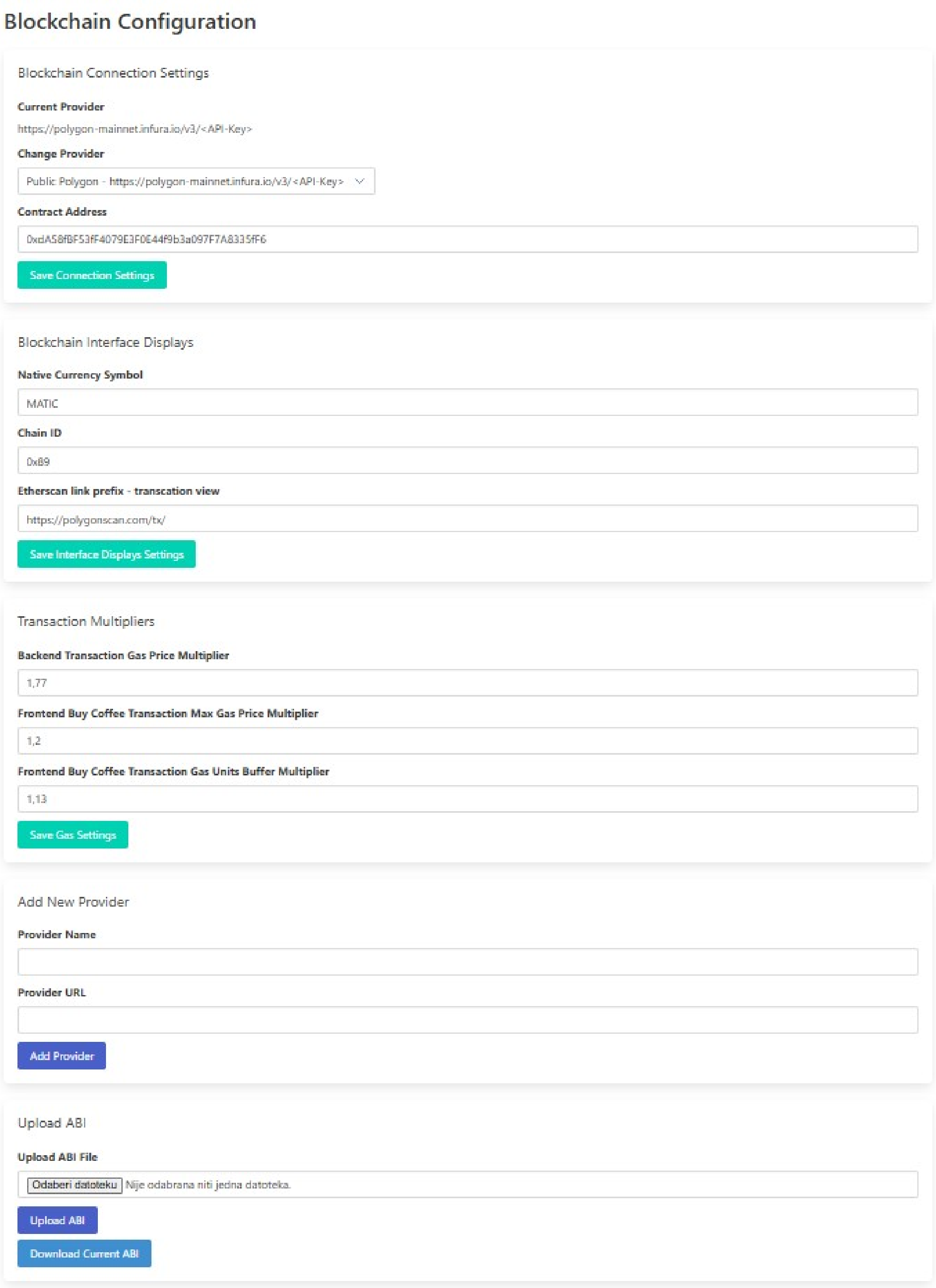

Admin Panel
Controllers:
Transaction View
User Management
Blockchain Configuration Management
Wallet Management
Smart Contract Management - FERcoin Tokens
Smart Contract Management - Native Currency

Figure 8.10: Initial Administrator Interface

Blockchain connection and display configuration

The administrator can configure the application's blockchain settings using the interface shown in Figure 8.11.


Blockchain Configuration

Blockchain Connection Settings

Current Provider

https://polygon-mainnet.infura.io/v3/<API-Key>

Change Provider

Public Polygon - https://polygon-mainnet.infura.io/v3/<API-Key>

Contract Address

0xdA58fBF53fF4079E3F0E44f9b3a097F7A8335fF6

Save Connection Settings

Blockchain Interface Displays

Native Currency Symbol

MATIC

Chain ID

0x89

Etherscan link prefix - transcation view

https://polygonscan.com/tx/

Save Interface Displays Settings

Transaction Multipliers

Backend Transaction Gas Price Multiplier

1,77

Frontend Buy Coffee Transaction Max Gas Price Multiplier

1,2

Frontend Buy Coffee Transaction Gas Units Buffer Multiplier

1,13

Save Gas Settings

Add New Provider

Provider Name

Provider URL

Add Provider

Upload ABI

Upload ABI File

Odaberi datoteku Nije odabrana niti jedna datoteka.

Upload ABI

Download Current ABI

Figure 8.11: Administrator Blockchain Configuration Interface

This interface has the following functionalities:

– Configuring the Web3 provider URL

– Management of the contract address

– Setting the deployed contracts network chain id and native currency symbol

– Management of the contract's blockchain network's block explorer URL prefix

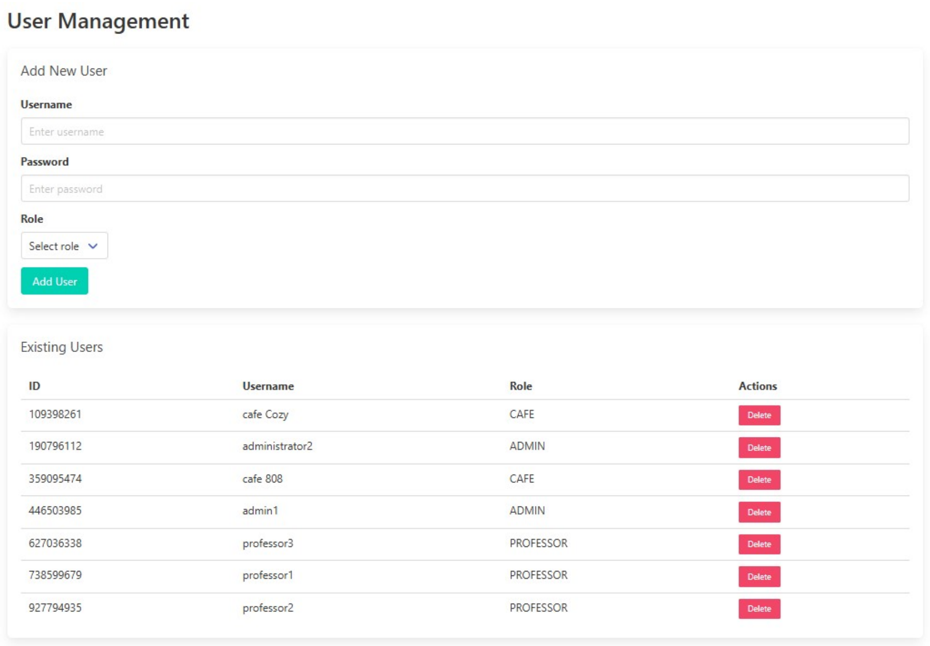
– Adjusting the transactions gas limit and gas price multipliers

– Adding new Web3 provider URLs

– Management of the server stored contract ABI

User Management

The administrator can manage the application users which require authentication to use the system. This interface is shown in Figure 8.12.



User Management

Add New User

Username

Enter username

Password

Enter password

Role

Select role

Add User

Existing Users

ID	Username	Role	Actions
109398261	cafe Cozy	CAFE	Delete
190796112	administrator2	ADMIN	Delete
359095474	cafe 808	CAFE	Delete
446503985	admin1	ADMIN	Delete
627036338	professor3	PROFESSOR	Delete
738599679	professor1	PROFESSOR	Delete
927794935	professor2	PROFESSOR	Delete

Figure 8.12: Administrator User Management Interface

This interface has the following functionalities:

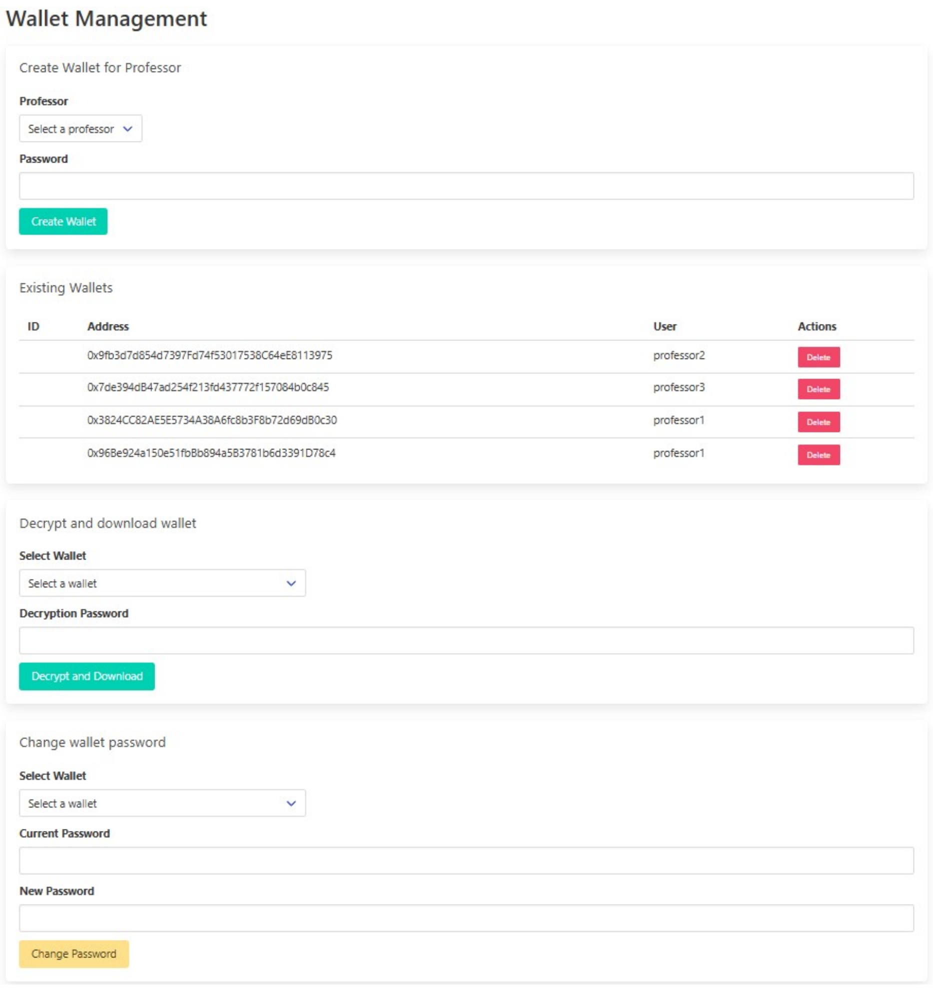
– View of all the system users

– Creating a new user by specifying a username, password and role. The username and password must each be at least 8 characters long.

– Deletion of system users

System Stored Wallets Management

The administrator manages the server stored wallets with the wallet management interface shown in Figure 8.13.


Wallet Management

Create Wallet for Professor

Professor

Select a professor

Password

Existing Wallets

ID	Address	User	Actions
	0x9fb3d7d854d7397Fd74f53017538C64eE8113975	professor2	Delete
	0x7de394dB47ad254f213fd437772f157084b0c845	professor3	Delete
	0x3824CC82AE5E5734A38A6fc8b3F8b72d69dB0c30	professor1	Delete
	0x96Be924a150e51fbBb894a5B3781b6d3391D78c4	professor1	Delete

Decrypt and download wallet

Select Wallet

Select a wallet

Decryption Password

Change wallet password

Select Wallet

Select a wallet

Current Password

New Password

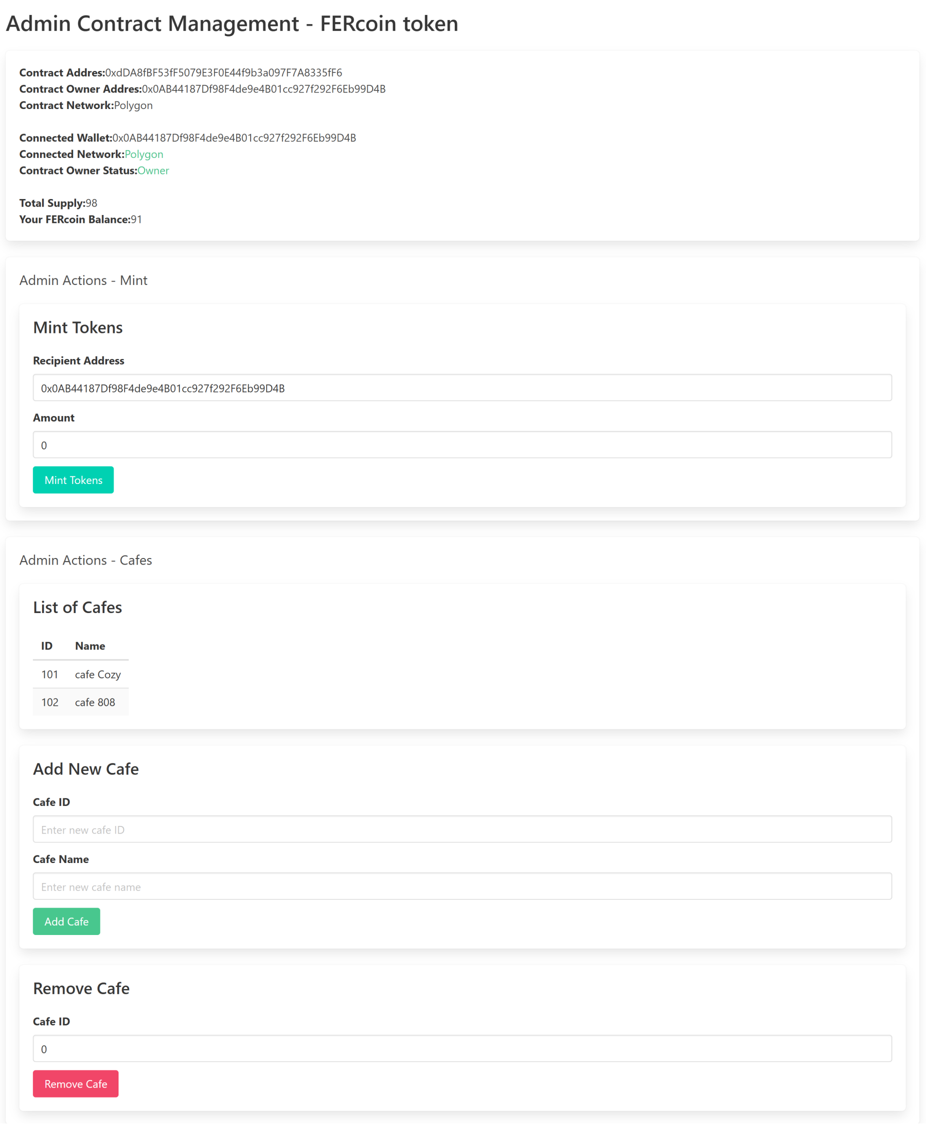
Figure 8.13: Administrator Wallet Management Interface

This interface has the following functionalities:

- Creation of a new wallet for a professor user, with a password for encrypting and decrypting the wallet's private key
- View of all the stored wallets
- Deletion of stored wallets
- Decryption and download of chosen wallet
- Changing the encryption password of a selected wallet

Smart Contract Management Interface for FERcoin Tokens

The administrator manages the deployed smart contracts FERcoin tokens using the interface shown in Figures 8.14 and 8.15.



Admin Contract Management - FERcoin token

Contract Addres:0xdDA8fBF53fF5079E3F0E44f9b3a097F7A8335fF6
Contract Owner Addres:0x0AB44187Df98F4de9e4B01cc927f292F6Eb99D4B
Contract Network: Polygon

Connected Wallet:0x0AB44187Df98F4de9e4B01cc927f292F6Eb99D4B
Connected Network: Polygon
Contract Owner Status: Owner

Total Supply:98
Your FERcoin Balance:91

Admin Actions - Mint

Mint Tokens

Recipient Address
0x0AB44187Df98F4de9e4B01cc927f292F6Eb99D4B
Amount
0

Admin Actions - Cafes

List of Cafes

ID	Name
101	cafe Cozy
102	cafe 808

Add New Cafe

Cafe ID
Enter new cafe ID
Cafe Name
Enter new cafe name

Remove Cafe

Cafe ID
0

Figure 8.14: Administrator Smart Contract FERcoin Token Management Interface

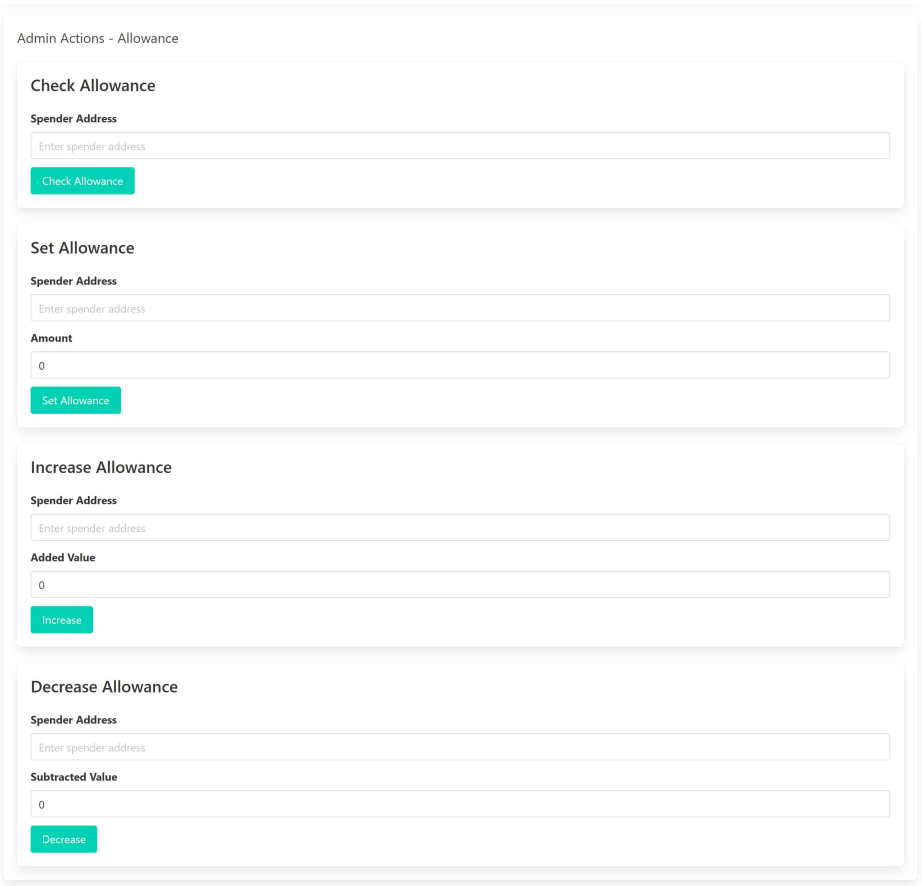


Figure 8.15: Administrator Smart Contract FERcoin Token Management Interface

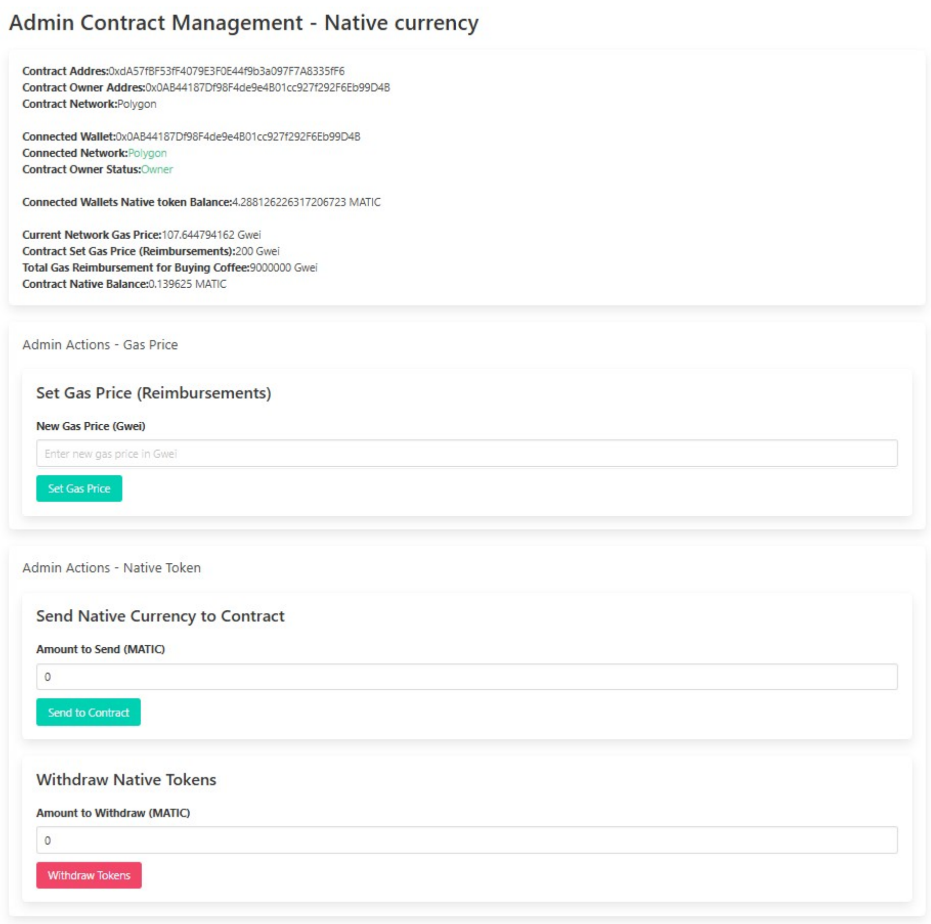
This interface has the following functionalities:

- View of the contract address, network and owner
- View of the connected wallet, network and whether is the connected wallet the contract owner
- View the total FERcoin token supply and the connected wallet's FERcoin balance
- Minting new FERcoin tokens to a specified address (typically the contract owner's address)
- Viewing the contract's defined cafes, creating new ones or deleting existing ones
- Checking and managing the FERcoin token allowances of other wallets

This interface can only be used with the contract owner's wallet, as all its functionalities correspond to owner-only functions of the contract. If a transaction is not signed with the contract owner's key pair, the smart contract will reject the execution of any owner-only function.

Smart Contract Management Interface for Native Currency

The administrator manages the native currency balance of the deployed contract using the interface shown in Figure 8.16.



Admin Contract Management - Native currency

Contract Addres:0xdA57fBF53fF4079E3F0E44f9b3a097F7A8335fF6
Contract Owner Addres:0x0AB44187Df98F4de9e4B01cc927f292F6Eb99D4B
Contract Network: Polygon

Connected Wallet:0x0AB44187Df98F4de9e4B01cc927f292F6Eb99D4B
Connected Network: Polygon
Contract Owner Status: Owner

Connected Wallets Native token Balance:4.288126226317206723 MATIC

Current Network Gas Price:107.644794162 Gwei
Contract Set Gas Price (Reimbursements):200 Gwei
Total Gas Reimbursement for Buying Coffee:9000000 Gwei
Contract Native Balance:0.139625 MATIC

Admin Actions - Gas Price
Set Gas Price (Reimbursements)
New Gas Price (Gwei)
Enter new gas price in Gwei
Set Gas Price
Admin Actions - Native Token
Send Native Currency to Contract
Amount to Send (MATIC)
0
Send to Contract
Withdraw Native Tokens
Amount to Withdraw (MATIC)
0
Withdraw Tokens

Figure 8.16: Administrator Smart Contract Native Currency Management Interface

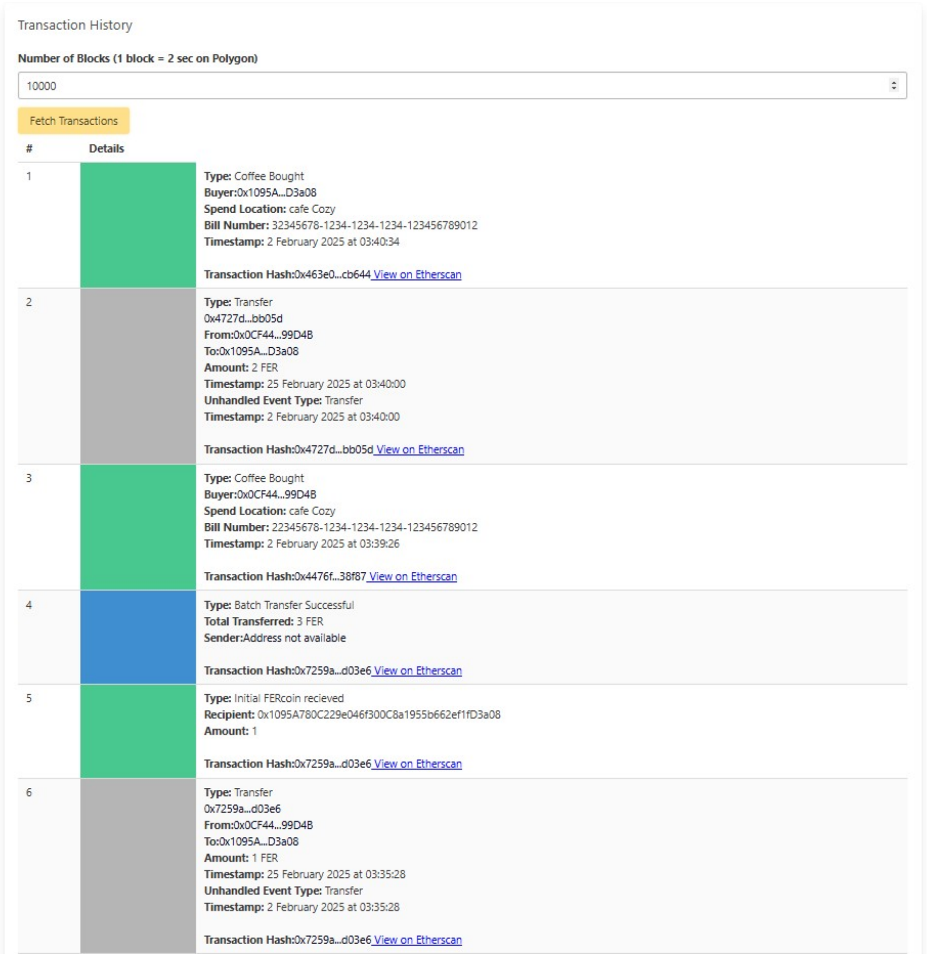
This interface has the following functionalities:

- View of the connected wallet's address, network, native currency balance, and whether the connected wallet is the contract owner
- View of the contract's native currency balance
- View of the current network gas price and reimbursement configuration
- Modifying the reimbursement configuration
- Sending and withdrawing native currency from the contract's native currency balance

This interface can only be used with the contract owner's wallet, as all its functionalities correspond to owner-only functions of the contract. If a transaction is not signed with the contract owner's key pair, the smart contract will reject the execution of any owner-only function.

Deployed Contracts Transactions and Events View

The administrator can view events generated by the smart contract and executed transactions to analyze blockchain activity and detect errors. This interface displays executed transactions and events, as shown with example data in Figure 8.17.



Transaction History

Number of Blocks (1 block = 2 sec on Polygon)

10000

Fetch Transactions

#	Details	
1	🟩	**Type:** Coffee Bought **Buyer:**0x1095A...D3a08 **Spend Location:** cafe Cozy **Bill Number:** 32345678-1234-1234-1234-123456789012 **Timestamp:** 2 February 2025 at 03:40:34 **Transaction Hash:**0x463e0...cb644 View on Etherscan
2	⬜	**Type:** Transfer 0x4727d...bb05d **From:**0x0CF44...99D4B **To:**0x1095A...D3a08 **Amount:** 2 FER **Timestamp:** 25 February 2025 at 03:40:00 **Unhandled Event Type:** Transfer **Timestamp:** 2 February 2025 at 03:40:00 **Transaction Hash:**0x4727d...bb05d View on Etherscan
3	🟩	**Type:** Coffee Bought **Buyer:**0x0CF44...99D4B **Spend Location:** cafe Cozy **Bill Number:** 22345678-1234-1234-1234-123456789012 **Timestamp:** 2 February 2025 at 03:39:26 **Transaction Hash:**0x4476f...38f87 View on Etherscan
4	🟦	**Type:** Batch Transfer Successful **Total Transferred:** 3 FER **Sender:**Address not available **Transaction Hash:**0x7259a...d03e6 View on Etherscan
5	🟩	**Type:** Initial FERcoin recieved **Recipient:** 0x1095A780C229e046f300C8a1955b662ef1fD3a08 **Amount:** 1 **Transaction Hash:**0x7259a...d03e6 View on Etherscan
6	⬜	**Type:** Transfer 0x7259a...d03e6 **From:**0x0CF44...99D4B **To:**0x1095A...D3a08 **Amount:** 1 FER **Timestamp:** 25 February 2025 at 03:35:28 **Unhandled Event Type:** Transfer **Timestamp:** 2 February 2025 at 03:35:28 **Transaction Hash:**0x7259a...d03e6 View on Etherscan

Figure 8.17: Administrator Smart Contract Transactions and Events View

On the Polygon network, blocks are mined approximately every two seconds. This information is displayed on the interface to help the administrator select the appropriate number of recent blocks, determining the time frame for analysis.

Administrator Process for Professor Onboarding

This section outlines the required steps for administrators to onboard a new professor and authorize them to distribute FERcoin tokens.

The workflow of this process is as following:

– Creating a new professor account using the **user management interface**

– Generating a system-stored wallet for the professor using the **wallet management interface**

- Transferring native currency to cover FERcoin distribution transaction fees, typically using **MetaMask**
- Configuring the FERcoin token allowance for the professor's wallet through the **smart contract FERcoin token management interface**
- Transferring native currency to the deployed smart contract's balance for reimbursement, which will be distributed along with FERcoin to students. This is done using the **smart contract native currency management interface**
- roviding the professor with the wallet decryption password and minting new FERcoin tokens for distribution via the **smart contract FERcoin token management interface**

Administrators interaction with system components during this process is illustrated in Figure 8.18.

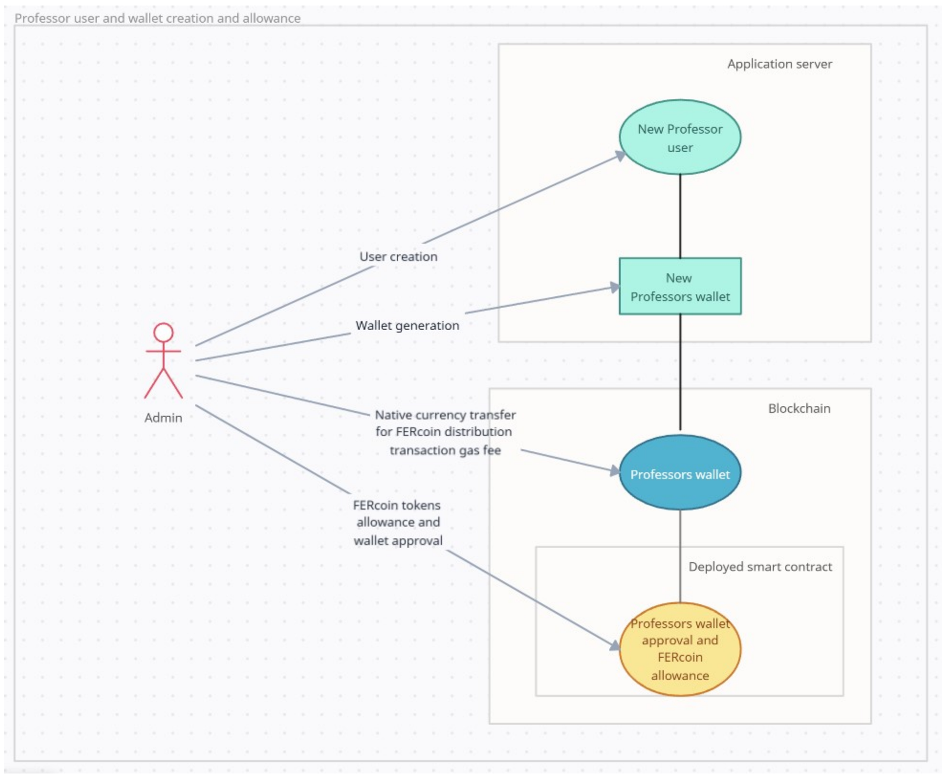

Professor user and wallet creation and allowance
Application server
New Professor user
New Professors wallet
User creation
Wallet generation
Admin
Native currency transfer for FERcoin distribution transaction gas fee
Blockchain
Professors wallet
FERcoin tokens allowance and wallet approval
Deployed smart contract
Professors wallet approval and FERcoin allowance

Figure 8.18: Administrator Interactions for Professor Onboarding, Wallet Creation and Authorization

8.6.5. Web3Utils Module

A lot of the interfaces use the same blockchain related logic. To avoid code duplication and to standardize blockchain operations across various interfaces, common functionalities are encapsulated into exported functions and implemented in `web3Utils.js` module. Client interfaces import and use these utility functions.

Table 8.19 provides a summary of the exported functions and constants in this module.

Table 8.19: Summary of Web3Utils Functions and Constants

Function / Constant	**Description**	**Notes**
`ETH_ADDRESS_REGEX`	Regular expression for basic Ethereum address validation.	Constant.
`isValidEthereum Address(address)`	Validates Ethereum address format using regex and Web3's utility functions.	Returns a boolean.
`isValidChecksum Address(address)`	Validates Ethereum address in checksum format using regex and Web3's utility functions.	Returns a boolean.
`getNetworkName (chainId)`	Returns a human-readable network name based on the provided chain ID.	Examples: 'Ethereum Mainnet', 'Polygon'.
`initContract (web3Provider, contractAddress)`	Initializes a smart contract instance using the ABI fetched from the server.	Asynchronous; returns the contract instance.
`fetchCafes(contract)`	Retrieves and formats defined cafes from the deployed smart contract.	Returns an array of objects with `id` and `name`.
`fetchBalance (web3Provider, walletAddress)`	Fetches the native cryptocurrency balance for given address in ether units.	Returns a decimal number in string format.
`fetchBalanceWei (web3Provider, walletAddress)`	Fetches the native cryptocurrency balance for given address in wei units.	Returns a unsigned integer number in string format.
`fetchTokenBalance (contract, walletAddress)`	Retrieves the ERC20 token balance (e.g., FERcoin) for a given wallet address and ERC20 contracts address.	Returns the token balance as a string.
`fetchRecent CoffeeBoughtEvents (web3Provider, contract, numberOfPreviousBlocks)`	Fetches recent `CoffeeBought` events from the blockchain.	Returns an array of event objects.
`formatBlock Timestamp (blockTimestamp)`	Formats a block timestamp into a human-readable date string.	Returns a formatted date string.
`NumberOfBlocksToHour TimeframePolygon (numberOfBlocks)`	Converts a number of blocks into an approximate timeframe in hours (specific to Polygon).	Returns a numerical value in hours.

8.7. Containerization and Initialization

The application is containerized using Docker. The Dockerfile sets up a lightweight Python environment with latest Python security patches, installs the required packages, creates a dedicated application user, and configures various environment variables and build-time arguments. The container includes an `entrypoint.sh` script used as the container entrypoint that performs dynamic initialization tasks such as generating a secret key, initializing the users database (if needed), and starting the Gunicorn server.

The Dockerfile and the `entrypoint.sh` define several environment variables and options. Table 8.20 summarizes these key settings.

Table 8.20: Environment Variables and Configuration Options

Variable / Option	**Description**	**Default / Note**
`SESSION_TYPE`	Determines session storage method. Options are `securecookie` (client-side encrypted sessions) or `filesystem` (server-side storage).	`securecookie` (set via `SESSION_TYPE_ARG`)
`PERMANENT_SESSION _LIFETIME_MINUTES`	Duration (in minutes) for which a session remains active.	480
`DEBUG`	Enables or disables debug mode for the application.	`False`
`PORT`	Port on which the application listens for incoming connections.	3000
`CONFIG_PATH_WEB3`	Path to the blockchain configuration file.	`./configs/ PolygonConfig.json`
`ABI_PATH`	Path to the smart contract ABI file.	`./blockchain/ contractABI.json`
`USERS_DATABASE_PATH`	Path to the SQLite database file used for storing user information.	`./databases/ sqlite/users.db`
`PYTHONPATH`	Configures the Python module resolution path.	`/application/..`
`SECRET_KEY`	Secret key for cryptographic operations (session cookies, CSRF tokens etc.). If not provided, it is dynamically generated at runtime.	Randomly auto-generated if not provided (should be provided if multiple workers are used)
`INITIAL_ADMIN _USERNAME`	Initial administrator username for the system.	Must be set at first-time initialization
`INITIAL_ADMIN _PASSWORD`	Initial administrator password for login.	Must be set at first-time initialization

The application user inside the container runs as `application_user` with the appropriate permissions to manage the application. Sensitive variables such as `INITIAL_ADMIN_USERNAME` and `INITIAL_ADMIN_PASSWORD` are unset after database initialization to prevent exposure.

The `entrypoint.sh` script is used as the container entrypoint and launches Gunicorn with a single worker. Since some application components rely on in-memory context, multiple workers are not yet supported without additional architectural changes.

8.7.1. Makefile

The Makefile simplifies common Docker operations by defining several targets that manage the lifecycle of the Docker container.

The Makefile defines the following variables:

- `IMAGE_NAME`: Name of the Docker image (default: `fercoin-flask`).
- `OUTER_PORT`: Port on the host machine, 3000 by default.
- `INNER_PORT`: Port inside the container, 3000 by default.
- `DOCKERFILE_PATH`: Path to the Dockerfile.

The following Table 8.21 lists the key make targets along with a brief description of each:

Table 8.21: Makefile Targets and Their Descriptions

Target	**Description**
`all`	Default target that builds the Docker image and runs the container in interactive mode.
`build`	Builds the Docker image using the specified Dockerfile.
`run-interactive`	Runs the Docker container interactively (prompts for initial admin credentials).
`run-detached`	Runs the Docker container in detached mode (useful when admin credentials are preset).
`stop`	Stops any running container created from the image.
`clean`	Removes the FERcoin Docker image.

Typically, the application is started by running `make` (which triggers the default target), building and running the container in interactive mode. For production environments where the initial admin credentials are preconfigured, `make run-detached` can be used instead, with defining initial administrator username and password environment variables upon running the container.

To stop running containers, `make stop` is used, and to remove the Docker image, `make clean`.

9. Security Solutions and Implementations

FERcoin tokens can be exchanged for real-world goods, giving them an indirect monetary value. The blockchain wallets of users interacting with the FERcoin system can hold various currencies, although this is strongly discouraged, as stated in the user guide. This makes security a critical concern. Malicious actors have an incentive to illegally acquire FERcoin tokens or compromise the FERcoin system in multiple ways using a variety of malicious techniques. Certain attack scenarios could also compromise the currencies held in the blockchain wallets of the FERcoin system users, specifically, students. The following sections discuss various security solutions and system-hardening techniques that are used and implemented in the FERcoin system.

9.1. Application security solutions and implementations

The application security solutions and implementations section discusses the solutions and techniques used in the FERcoin systems' application component, both the server-side and the client-side implementations. Web applications and interfaces that communicate and work with developed contracts, in most cases, implement their blockchain-related functionalities for users on the client-side.

That's why the attacks that target the users of this project most commonly focus on the client-side, specifically on the front-end components that users rely on for building and executing transactions. High-profile cases, such as those affecting BadgerDAO, Curve Finance, Radiant Capital and Solana web3.js packet Supply Chain Attack [5], illustrate the risks and ramifications of these exploits. These types of attacks usually involve compromising the CDN, DNS, or some other projects component which enables the attackers to manipulate the content served to all the project users and target the projects users and their crypto wallets.

That is why client-side security is a crucial aspect of developing interfaces that implement web3 blockchain-related capabilities, such as generating transactions for communication with deployed smart contracts and connecting software wallets with interfaces, which store cryptocurrency wallet keys. All the server-side components managing front-end interfaces and other client-side functionalities must also be secured. An example of such a component is the blockchain configuration panel available on the administrator interface, where the address and blockchain network of the deployed FERcoin contract are stored and can be changed. These values are forwarded to the users' interfaces and if compromised, could alter the address of the deployed FERcoin smart contract to a malicious one and target the users this way. If such server-side components were compromised, the malicious actor could manipulate the front-end components that the application delivers to users.

9.1.1. Subresource Integrity and Version Lock

Three primary CDN (Content Delivery Network) resources are used to support the front-end user interfaces and all client-side functionalities. The resources used are Web3.js, Vue.js and Bulma css. These resources are retrieved from cdn.jsdelivr.net CDN and used in the interfaces they are required.

An attack on the client-side component by compromising the CDN-fetched packets could involve direct compromise of the CDN itself, where malicious actors alter the hosted resources. Additionally, attackers might exploit cache poisoning by injecting malicious payloads into intermediary caches (e.g., browsers, proxies, or ISP caches), DNS spoofing/poisoning to redirect users to rogue servers hosting tampered resources as well as a few other attack techniques. If HTTPS is not enforced, attackers could perform Man-in-the-Middle (MITM) attacks, intercepting and modifying scripts before they reach the client. A large-scale example of a CDN compromise has already occurred when a foreign entity took over the JavaScript Polyfill project, embedding malware in its CDN-hosted assets [19].

In such scenarios, malicious actors could alter the contents of the received packets used by this project, injecting malicious code into them.

The most critical resource in terms of security is Web3.js library, such libraries have previously suffered security breaches, an example of this is Solana web3 JavaScript blockchain SDK "@solana/web3.js" [1]. If compromised, malicous code injected in the library packet could steal users cryptocurrency wallet keys or manipulate how the interface builds the transactions. Usually, after the transaction is built, the interface requests from students' MetaMasks' connected account to either sign the transaction and send it to the blockchain network or refuse the signing of the transaction. The compromised CDN packet could manipulate the generated transaction. Instead of executing the expected exchange of FERcoin for coffee on the FERcoin smart contract, the compromised script could "drain" the cryptocurrency holdings of the user's wallet.

The way this security issue is secured is by first "locking" the packet version to the static version that was used in the final build of the project instead of the latest version and then storing the hash of the packet, which will be used in the future for validating the integrity of the CDN fetched packet. This also helps with preventing errors and mismatches between the usage of the functions at the time of development and in the newest packet versions.

After the packet is locked, the SRI (Subresource Integrity) [14] security feature of the browsers is used to generate the SHA-384 hash of the used packet version, which is supposed to stay static, and then store the hash together with the script element that retrieves the packet from the embedded URL. The simple tool used for this is the online SRI Hash Generator [2].

When the client-side loads the interface and fetches the needed scripts from the CDN, the browser generates the SHA-384 hash of the fetched library and compares it to the previously computed hash stored on the server and embedded in the script element. This ensures that the retrieved resource is the same as it was at the time the hash was generated. If there is a mismatch between the hashes, the fetched resource is dropped with an error. This prevents the ramifications of compromises of the packets fetched from the CDN because if any malicious code is injected, the hashes won't be identical anymore and the resource will be dropped.

The following code excerpt demonstrates the implementation of the SRI feature.

```
<script
nonce="{{ csp_nonce() }}"
src="https://cdn.jsdelivr.net/npm/web3@4.16.0/dist/web3.min.js"
integrity="sha384-Ptk2PWqkZWMoP7ivsQOoijqfM5hQC4AgIEmQ+
    WUgk2OY5eNAmVLXunFrfpnZhmkb"
crossorigin="anonymous">
</script>
```

In the src attribute, it can be noticed that the web3 packet is locked to the version 4.16.0 and under the integrity attribute, the hash algorithm used for generating the hash is defined, as well as the generated hash itself, which is used for comparison.

9.1.2. Content Security Policy and Script Nonce

The application employs a **Content Security Policy (CSP)** to mitigate security risks such as **Cross-Site Scripting (XSS)**, **clickjacking**, and **data exfiltration**. CSP enforces strict control over script execution, resource loading, and data transmission, ensuring that only trusted sources are permitted.

CSP Directives and Their Purpose

CSP configuration is shown in the code excerpt below.

```
csp = {
    'default-src': "'self'",
    'script-src': [
        "'self'",
        "https://cdn.jsdelivr.net",  # Allow Web3.js, Vue.js, Bulma CDN
        "https://cdnjs.cloudflare.com",
        "'unsafe-eval'",  # Required for Vue/Web3
    ],
    'style-src': [
        "'self'",
        "https://cdn.jsdelivr.net",
        "https://cdnjs.cloudflare.com",
        "'unsafe-inline'"  # Needed for Bulma
    ],
    'connect-src': "'self' *",
    'frame-src': "'none'",
    'frame-ancestors': "'none'",
    'base-uri': "'self'",
    'form-action': "'self'"
}

FRAME_OPTIONS = 'DENY'
FORCE_HTTPS = False # Set to True in production if using HTTPS
```

The explanation of each directive in the CSP configuration is as follows.

– **script-src**: Restricts scripts to trusted sources, including the application server and the trusted CDNs. Inline scripts require a dynamically generated **nonce** to execute, ensuring that only authorized scripts run. The directive allows `unsafe-eval` (necessary for Vue/Web3 functionality) but the Subresource Integrity (SRI) is enforced, as described in the previous section, to block subsequently modified scripts from CDNs.

– **style-src**: Limits stylesheets to approved domains while permitting inline styles (`unsafe-inline`), required for the Bulma CSS framework.

– **connect-src**: Allows the application to connect to various external APIs. The wildcard ($\star$) is necessary because client-side interfaces need to dynamically fetch data, such as current network gas prices, from different API endpoints that may change over the application's lifetime. Additionally, further security measures can be implemented by explicitly defining a set of allowed endpoints. However, since these endpoints may change during the application's lifetime (e.g., when an admin updates the API settings), the system must properly handle such changes and dynamically update the directive to include newly authorized endpoints.

– **frame-src** & **frame-ancestors** & **FRAME_OPTIONS**: All three directives are set to restrictive values to mitigate **clickjacking attacks**. **frame-src** and **frame-ancestors** are both set to `'none'`, ensuring that this application cannot embed external content within iframes, preventing clickjacking attacks initiated from this site toward other sites. Additionally, **FRAME_OPTIONS** is set to `'DENY'`, blocking any attempts by external websites to embed this application within an iframe, protecting against external clickjacking attacks targeting this site.

- **base-uri**: Restricts the base URL of the application to `'self'`, preventing malicious base tag modifications.
- **form-action**: Ensures that form submissions are confined to the application's domain only, preventing **data exfiltration** to external sites.

Ensuring Secure HTTPS Connections in Production

The configuration includes the `FORCE_HTTPS` flag, which is currently set to `False` for development but must be enabled (set to `True`) in production when HTTPS is configured. Enforcing HTTPS ensures that all traffic is encrypted, preventing **man-in-the-middle (MITM) attacks** and **data interception**.

CSP Implementation in Flask

The CSP is implemented using **Flask-Talisman**, a security middleware for Flask applications. The flask-talisman initialization function is available in the code excerpt below.

```python
def initialize_talisman_csp(app, csp, FRAME_OPTIONS, FORCE_HTTPS):
    """
    Initialize Flask-Talisman with security headers and CSP

    Args:
        app (Flask): Flask application instance
        csp (dict): Content Security Policy configuration
        frame_options (str): X-Frame-Options header value
        force_https (bool): Enforce HTTPS redirection

    Returns:
        Talisman: Configured security middleware instance
    """
    talisman = Talisman(
        app,
        content_security_policy=csp,
        content_security_policy_nonce_in=['script-src', 'style-src'],  # 
            Enable nonce generation
        frame_options=FRAME_OPTIONS,
        force_https=FORCE_HTTPS
    )

    return talisman
```

The Flask-Talisman initialization function defines how a nonce is applied to client-side scripts and styles for permission control. The nonce is generated on the server side and embedded in the corresponding HTML elements before delivering the requested resource to the user. The following code excerpt, previously discussed in the section on SRI implementation, is shown again here—though this time to illustrate nonce embedding rather than integrity verification.

```
<script
nonce="{{ csp_nonce() }}"
src="https://cdn.jsdelivr.net/npm/web3@4.16.0/dist/web3.min.js"
integrity="sha384-Ptk2PWqkZWMoP7ivsQOoijqfM5hQC4AgIEmQ+
   WUgk2OY5eNAmVLXunFrfpnZhmkb"
crossorigin="anonymous">
</script>
```

9.1.3. CSRF Protection

Cross-Site Request Forgery (CSRF) protection in the Flask application is implemented using the `CSRFProtect` class from the `flask_wtf.csrf` module [9].

The `flask_wtf` module extends Flask's capabilities by integrating `WTForms (Web Forms Toolkit)` [10], a form-handling library, with additional security features such as CSRF protection.

In addition to CSRF protection, `flask_wtf` leverages `wtforms` to simplify form validation and processing by providing a structured way to define form fields and enforce validation rules. These validation mechanisms apply to all forms that communicate with the backend, ensuring data integrity and security.

CSRF protection requires a secret key to securely sign the token. By default, this uses the Flask app's `SECRET_KEY`, unless specified otherwise. This `SECRET_KEY` is the same key used for both CSRF token generation and session encryption. It will be discussed further in the next section: *Secret key*.

The function `initialize_csrf(app)` sets up CSRF protection for all form submissions that interact with backend logic. The import of the requied class and the initialization function is shown in the code excerpt below.

```
from flask_wtf.csrf import CSRFProtect

def initialize_csrf(app):
    """
    Set up CSRF protection for form submissions

    Args:
        app: Flask application instance

    Returns:
        CSRFProtect: Initialized CSRF protection handler
    """
    csrf = CSRFProtect(app)
    return csrf
```

CSRF protection ensures that malicious requests from unauthorized sources cannot be executed on behalf of an authenticated user. A hidden input field, `csrf_token`, is included in each form to verify the legitimacy of a request, as shown in the excerpt below.

```
<input type="hidden" name="csrf_token" value="{{ csrf_token() }}">
```

When the form is sent from the client-side and reaches the server-side, the CSRF token from the form is validated. This ensures that the request originates from the application interface itself, preventing malicious actors from tricking users into submitting unintended requests by embedding malicious forms on external sites. The CSRF token validation is handled automatically by the `flask_wtf` module.

9.1.4. Secret Key

The secret key can either be explicitly set when running the container or, if not set, it will be generated randomly.

If a secret key is not explicitly set, it is randomly generated using the `token_urlsafe` function from Python `secrets` module. The length of the key is 64 bytes and is converted into a URL-safe base64-encoded string to avoid errors that may arise with certain generated byte sequences.

The following entrypoint script excerpt checks whether the secret key is defined as an environment variable. If not, it generates a new one as described above:

```
# Generate SECRET_KEY if not provided
if [ -z "$SECRET_KEY" ]; then
    export SECRET_KEY=$(python -c "import secrets; print(secrets.
       token_urlsafe(64))")
    echo "Generated SECRET_KEY: $SECRET_KEY"
fi
```

The secret key is generated in the entrypoint script rather than in the Dockerfile to prevent it from being embedded in the built image during the build phase. If someone had access to the image, they could read and extract the secret key. To mitigate this risk, secret key generation is performed during container initialization rather than the build phase, ensuring that it is absent from the image and not at risk of compromise. It is later on read by flask application from the environment when the container starts running.

This does mean that each time the container is restarted, a new random secret key would be generated, because all the environment variables set via export in the entrypoint script do not persist between container runs. This would invalidate all the existing session cookies.

This is not a major concern, as the number of authenticated users is relatively small (limited to professors, administrators, and cafe workers). These users access the application infrequently, and Docker container restarts are uncommon. The only impact of a restart is that users must log in again, which they usually have to do anyway, as the session lifetime is 8 hours.

When running the container in production, it is recommended that the secret key should be explicitly set as follows:

```
docker run -e SECRET_KEY=<secret key> ...
```

This way the explicitly set secret key will persist between restarts of the container, and will not be embedded in the image itself.

9.1.5. Session Management, Validation and Cookie Forgery Protection

The system provides two session management methods: using the filesystem storage or encrypted session cookies. Encrypted session cookies are the default and recommended approach unless explicitly set to filesystem storage in the Dockerfile. The remainder of this section focuses on encrypted session cookies.

Upon login, the session is initialized and defined with the user's username, ID, and role, as shown below:

```python
# Session initialization
session.update({
    'username': user.username,
    'id': user.id,
    'role': Role.from_str(user.role).value
})
```

The session object is then encrypted with Flask's secret key and sent to the user as an encrypted session cookie.

All authenticated users (those who log in to access protected system functionalities) use sessions. All endpoints requiring authentication validate the session and check user roles before granting access. All authentication-protected endpoints validate the session and check user roles before granting access. These endpoints are defined in each authenticated user's Flask blueprint.

All Flask blueprints for authenticated users implement a role check (which also performs session validation) before every request, as shown in the administrator blueprint example:

```python
@admin_bp.before_request
def verify_admin_access():
    """
    Validate admin role for all blueprint routes before processing
        requests.
    """
    access_check = check_role(Role.ADMIN.value)
    if access_check is not True:
        return access_check
```

The `role_check` function is responsible for session validation and role verification, as shown below:

```python
def check_role(required_role: str):
    """
    ...
    """

    if 'id' in session:

        user_id = session['id']

        # Verify against database
        user = DbUsersViewInterface.get_user_by_id(user_id)

        if not user:
            # No user with the session ID
            # Clear compromised session and redirect
            session.clear()
            return redirect(url_for('auth.login'))

        if user.role != session['role']:
            # Db user role and session role mismatch
            # Clear compromised session and redirect
            session.clear()
            return redirect(url_for('auth.login'))

        user_role = user.role

        # Check if the current session role matches the required role
        if user_role == required_role:

            # Role is correct, request processing can continue
```

```
                return True

            else:
                # Role is not correct, redirect user to his dashboard
                return redirect_role(user_role)

        else:
            # If no role is found, redirect to login page
            return redirect(url_for('auth.login'))
```

The session validation and role verification process follows these steps:

- Verify that a session exists
- Confirm that the session ID corresponds to a valid user in the database
- Ensure that the session role matches the user's role stored in the database
- Ensure that the user's role matches the required role (required role is set as an argument to the function)
- If any check fails, return the redirect for login page

This implementation makes it significantly more difficult for malicious actors to forge cookies, even if they obtain the secret key used for session cookies encryption. To forge a valid cookie using a compromised secret key, an attacker would also need to obtain the user's random 9-digit ID and their role as stored in the database, to successfully forge that user's cookie.

An additional security benefit of this implementation is that when an administrator deletes a user from the database, their session cookie becomes invalid immediately, rather than allowing the user to continue using their session cookie until expiration, if there was no Database-backed role verification.

9.1.6. Restricting Access to Protected Client-Side Resources Based on User Roles

Certain authenticated user interfaces rely on client-side JavaScript for functionality. In conventional setups, static frontend assets (JavaScript, CSS files and similar) are stored in a publicly accessible static directory.

However, this would allow unauthenticated users to retrieve JavaScript files and other frontend components intended for authenticated users interfaces.

These scripts do not contain sensitive data on their own without the appropriate authenticated users controller injecting specific values into them through the template that uses these scripts. The sensitive data injected mostly consists of API endpoints with API keys. However, their exposure could help attackers to gather information about the authenticated users' interfaces and infer some system functionalities. This would allow them to plan further attacks, like social engineering and XSS attempts.

To prevent unauthorized access, the system separates frontend assets into two directories:

– `static` - publicly accessible frontend components and assets

– `static_auth` - protected frontend components and assets requiring authentication

Within `static_auth`, role-based subdirectories (`admin/`, `cafe/`, `professor/`) correspond to the possible user roles and ensure that authenticated users can only access assets relevant to their role. Unlike public resources, access to these protected files is managed by separate authentication logic instead of direct filesystem access.

The authentication logic is implemented in the authentication controller, where the handler for fetching the protected resources requires a session cookie to be present in the request. The validity of the cookie as well as the role required for accessing the requested component is checked by the `check_role` function (previously described in `Session Management, Validation and Cookie Forgery Protection`).

The handler mentioned is implemented as follows:

```python
@auth_bp.route('/static_auth/<role>/<path:filename>', methods=['GET'])
def serve_static_auth(role, filename):
    """
    Securely serve role-specific frontend assets from protected
        directories.

    Parameters:
        role (str): Directory name that must match user's role
                    (e.g., 'admin' for files in static_auth/admin/)
        filename (str): Relative path to asset within role directory

    Security:
        - Validates user's role matches requested directory
        - Returns login redirect for unauthorized access
        - Returns login redirect for missing files
        - Prevents enumeration
    Returns:
        file: Requested asset or redirect if unauthorized
    """

    # Role validation, comparing the role in session
    # and the role needed for the file requested
    redirect_result = check_role(role)
    if redirect_result != True:
        return redirect_result

    # Check if the requested file exists in the role-specific directory
    role_dir = os.path.join(STATIC_AUTH_DIR, role.lower())
    if not os.path.exists(os.path.join(role_dir, filename)):
        return redirect(url_for('auth.login'))

    # Serve the file
    return send_from_directory(role_dir, filename)
```

The path for requesting a protected component is in the format as follows:
`/static_auth/<role>/<path:filename>`

In each request's path, the role corresponding to both the subdirectory in `static_auth` directory and the user role required for accessing the protected component must be present, as well as the filename of the protected component.

Then that role in the path is used first to verify that the user's session is valid and contains the required role for accessing the component, by calling the `check_role` function with the role from the path as an argument. That same role from the request together with the path requested is then used to construct the path to the requested file from the `static_auth` directory.

If either session validation or role verification fails, the request is redirected to the login page instead of returning a "403 Forbidden" or "404 Not Found" response. This prevents attackers from systematically probing and enumerating the system to discover the existence and paths of the protected resources.

This implementation enforces strict role-based access control for protected frontend resources. It prevents unauthorized users from accessing protected assets while ensuring that authenticated users can only retrieve resources relevant to their role.

9.1.7. Enumeration Protection

Enumeration protection prevents unauthorized users from discovering non-public endpoints. It also restricts authenticated users from detecting endpoints that belong to other users.

This is achieved by explicitly defining which paths and path prefixes are publicly accessible, allowing them to be reached without a session cookie.

The following paths are publicly accessible without authentication:

– All resources in the `static` directory

– The student interface at base path / and the student guide at `/guide`

– All paths beginning with the `/auth/` prefix.

– The deployed smart contract ABI JSON file at `/contract-abi`

The list of public resources from the `static` directory is dynamically generated by reading the contents of the directory, while other public paths are explicitly hardcoded. If new public endpoints are introduced, they must be manually added to the lists in the `SecurityConfig`

The middleware enforcing these restrictions is implemented as follows:

```python
@app.before_request
def enforce_auth():
    """
    Authentication gatekeeper middleware

    Allows:
    - Authenticated users (session contains role)
    - Public resources (allowed paths/prefixes)
    """

    # Skip for authenticated users
    if 'role' in session:
        return

    # Allow public endpoints
    if request.path in SecurityConfig.ALLOWED_PATHS:
        return

    # Allow static assets
    if any(request.path.startswith(prefix) for prefix in SecurityConfig.
       ALLOWED_PREFIXES):
        return

    # Redirect unauthenticated access attempts
    if not session.get('authenticated'):
        return redirect(url_for('auth.redirect_user'))
```

This middleware runs before every request and follows this logic:

– If a valid session exists, continue processing the request

– If the requested path is in the list of publicly accessible paths or matches a public prefix, continue processing

– Otherwise, redirect unauthenticated users to the login page or authenticated users to their role-based interface

By enforcing these rules, the system ensures that unauthenticated users receive the exact response for both non-existent endpoints and protected endpoints, effectively preventing both the standard and response-time-based enumeration techniques.

Preventing Enumeration of Other Authenticated Users' Endpoints

The prevention of enumeration of other users' authenticated endpoints by an already authenticated user is handled by the `check_role` function (previously described in `Session Management, Validation and Cookie Forgery Protection`).

`check_role` is applied as a middleware to all authenticated user endpoints before processing requests, as defined in each user's blueprint.

The function returns the same redirect, which for authenticated users redirects to their interface, if the role of the user requesting the access to the endpoint is not the defined role required by the endpoint.

If an authenticated user requests a non-existent endpoint, the following logic is applied:

```python
@app.errorhandler(404)
def page_not_found(e):
    """
    On 404, redirect
    Used for further enumeration protection

    (404 occurrence can still be detected, but only possible
    on publicly available prefixes for unauthenticated actors)
    """
    return redirect(url_for('auth.redirect_user'))
```

Although this logic returns the same redirect as for unauthorized requests, differences in responses can still be detected.

This makes it possible for authenticated users to enumerate the base prefix of other users' endpoints. However, for any requested endpoint that begins with a valid prefix, the redirect response is exactly the same, regardless of whether the full path corresponds to a valid or non-existent endpoint.

This way, they can not enumerate the remainder of any unauthorized endpoint, especially any endpoints that deal with POST requests. They can only discover the existence of the prefix, that is on its own used only to return the initial users frontend interface, working as a pure GET request functioning endpoint.

This prevents a malicious actor that has gained access for one user to successfully enumerate other users endpoints.

9.1.8. Bruteforce Protection

To prevent brute-force attacks, especially on the login page, the Flask application uses the `Flask-Limiter` module [8], which provides rate limiting capabilities. This extension is configured to limit the number of requests per IP address in the request on protected resources.

The limiter is applied to the entire authentication blueprint, to limit both the login attempts and all the unauthorized requests or requests to non-existent endpoints. Since unauthorized or non-existent endpoint requests trigger redirect function from the authentication blueprint, they also get rate-limited, further reducing enumeration capabilities.

The rate limit is set to 10 requests per minute and defined in the `SecurityConfig`.

The following code shows the necessary imports and the limiter initialization function:

```python
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

def initialize_limiter(app):
    """
    Create rate limiter instance for brute-force protection

    Args:
        app: Flask application instance

    Returns:
        Limiter: Configured rate limiting instance
    """
    limiter = Limiter(get_remote_address, app=app)
    return limiter
```

This function is used to initialize the limiter and apply it over authentication blueprint during the application initialization as follows:

```python
# Register blueprints
...
app.register_blueprint(auth_bp, url_prefix='/auth')
...

# Set rate limit on authentication blueprint
rate_limiter.limit(SecurityConfig.RATE_LIMIT_AUTH_BP,
error_message="Too many requests. Please try again later.")(auth_bp)
```

Currently, the limiter uses its default in-memory storage for tracking request counts. While this approach remains viable even in production, using a persistent storage backend should be considered, especially when scaling the application across multiple workers or instances. `Flask-Limiter` supports various storage backends, including Redis, Memcached, and MongoDB.

Additionally, if the application is hosted behind a reverse proxy in production, the `get_remote_addr`
function could return the proxy's IP address instead of the requester's IP. In such cases, the limiter should be configured accordingly, following the guidelines in the `Flask-Limiter`

documentation.

9.1.9. Separation of the Master Wallet from the Application and Encryption of Server-Side Stored Wallets

The most critical component of the system is the deployed smart contract. It faces two primary risks: one, a bug or vulnerability in the smart contract code; and two, the compromise of the master key—the key pair of the account that owns the contract. If the master key were compromised, a malicious actor could invoke all functions restricted to the contract owner, such as minting new FERcoin, shutting down the contract, or withdrawing native currency from the contract's balance.

For this reason, the master key is kept entirely separate from the application. It is the administrator's responsibility to securely store the master key using appropriate methods. The application only provides an interface for contract owner's functions by building transactions. However, the transactions must still be signed by the master key, which remains separate from the application. These interfaces are only usable if the administrator uses MetaMask, but the administrator is free to manage the deployed contract on the blockchain network any way they choose, even completely omitting the application's contract management interfaces. Thus, even if an administrator's account is compromised, the deployed smart contract remains secure, and no native currency or FERcoin will be lost or misused as long as the master key is secure.

As noted, if the deployed smart contract were compromised, recovery would be significantly more difficult and the potential ramifications far greater than those associated with an application breach. Essentially, the application only provides a set of interfaces for reading data and communicating with the deployed smart contract for different users—except for one user group, the professors. For professors, the system provides server-side wallet generation, storage, and usage to simplify their interactions with the FERcoin system and eliminate the need for additional software like MetaMask.

System-generated and stored wallets are managed by administrators, for use by professors. These wallets are always stored with their private keys encrypted and are decrypted only temporarily at the time they need to be used to sign a transaction. The passwords used for decryption are not stored on the server; users must store them separately from the application.

Wallets are stored as JSON files on the server and contain the following values:

- Wallet address (derived from public key)
- Encrypted private key (secured using AES encryption with salting, following the Web3 Secret Storage Definition)

- ID of the user associated with the wallet
- Username of the user associated with the wallet

These wallets are always stored with their private keys encrypted and are only temporarily decrypted on the server side when needed to sign a transaction. The decryption passwords are not stored on the server; instead, each user must keep their password separately from the application.

Administrators have access to all wallets stored on the server, although in the encrypted state. They can only decrypt the wallets if they know the password for decryption.

For the management of the server stored wallets, three interfaces are implemented:

- `WalletViewInterface` used only for viewing the wallets present on the server in the encrypted state
- `WalletDecryptInterface` used for decrypting wallets with the decryption password
- `WalletWriteInterface` used for generating new wallets, re-encrypting existing wallets with a new password (with the old password provided), and deleting stored wallets from the server

Administrators use all of these interfaces.

Professors, on the other hand, use the `WalletViewInterface` to view all of the wallets associated with them and load the data of the chosen wallet. Backend logic verifies that the professor's ID matches the ID stored in the wallet JSON file. This prevents any professor from accessing another professor's wallet and helps to prevent horizontal privilege escalation. Even if an attacker compromises a professor's account, they cannot access wallets belonging to other users. Note that the private key is never loaded into any interface, neither in its encrypted nor decrypted form, to avoid the possibility of brute-force attempts on the encrypted private key. The only way a private key of a system stored wallet can leave the server is if the administrator uses the functionality of decrypting the wallet with its password and downloading the wallets JSON file in the decrypted format. An administrator may use this function if actions outside the intended scope of system-stored wallets—limited to FERcoin distribution—are required.

When the professors initiate the action of distributing FERcoin to selected students, the `WalletDecryptInterface` is used on the server side to temporarily decrypt the chosen wallet with the provided password. The decrypted wallet is then used to sign the transaction of FERcoin distribution, which the backend validates before sending it to the blockchain network.

The only wallets stored on the server are those assigned to professors, and they are decrypted only temporarily for transaction signing. These wallets have limited contract management capabilities, which are defined in the deployed smart contract and cannot be modified without the master wallet. Professors' wallets can only spend the FERcoin allowance allocated to them. These allowances are only usable if sufficient FERcoin tokens have been minted and are available in the master wallet's balance. Both actions, minting and setting allowances, are limited only to the contracts owner, requiring the master wallet.

The only way these wallets could be compromised is if a malicious actor gains access to the server. An attacker could either retrieve the encrypted private keys, in which case they would need to brute-force the encrypted wallet's private keys encryption password or read the server's memory at the moment when a wallet is decrypted for signing a transaction. If successful, the attacker could use the compromised wallet, but only within the constraints of its set allowance and the smart contract's state at that time. Appropriate countermeasures for such attacks are discussed in the next chapter, chapter 10.

9.2. Smart Contract Security

FERcoin's smart contract security primarily focuses on protecting the contract from the most common vulnerabilities. The vulnerabilities considered are based on the `OWASP Smart Contract Top 10` [16] and the Solidity Security Considerations Documentation [7]. This section discusses several implementations designed to mitigate these vulnerabilities.

In addition to inheriting ERC20, the FERcoin contract also inherits Ownable and Pausable from OpenZeppelin, a library for secure smart contract development that implements well-known and used standards. These inheritances ensure that the FERcoin contract functionalities and access control are implemented following the best approaches and standards.

9.2.1. Access Control

Access control is achieved by using the Ownable inheritance and using the ERC20 token approval logic. All the functions that change the state of the smart contract fall into one of the three categories:

- Public functions - available to everyone
- Owner-only functions - available to only the contract owner
- Approved-wallet functions - available only to addresses explicitly approved by the contract owner, allowing them to distribute their assigned allowances

All the functions that manage sensitive logic and should only be modified by an administrator use the `onlyOwner` modifier from the Ownable inheritance. This ensures they are callable only by the contract's owner. The contract owner is typically the address that deployed the smart contract, and its key pair is securely stored by the FERcoin administrator.

Functions available only to approved addresses are used for distributing FERcoin. These functions, `transferFrom` and `batchTransferFrom`, allow professors to distribute FERcoin to students. The way the access control enforcement is implemented for these functions is not by using a function modifier, like for owner's functions, but by implementing a `require` statement that checks if the caller address has the necessary allowance approved by the contract owner to execute the function.

Only the contract owner can approve addresses and set allowances, and he can only set an allowance which gives the right to approved addresses to spend FERcoin from the owner's balance.

Since only addresses approved by the contract owner have a nonzero allowance—and because the distribution functions cannot be called with a zero amount—any calls from unapproved addresses will fail due to `require` checks, as shown:

```
function transferFrom(address spender, address recipient, uint256 amount)
public override whenNotPaused returns (bool) {

    require(amount > 0, "Transfer amount must be greater than zero");

    require(spender == owner(), "Tokens can be transferred only from the
        owners account balance");


    require(isEOA(recipient), "Recipient must be an EOA");

    require(allowance(spender, _msgSender()) >= amount, "Transfer amount
        exceeds allowance or the caller is not allowed to transfer from
        this account");
    ...
```

The last set of functions that are publicly available and change the state of the contract are the `transfer` and `buyCoffee` functions, both of which are used exclusively to manage the caller's FERcoin balance.

9.2.2. Reentrancy Attack Protection

Reentrancy attacks are a serious security threat unique to blockchain smart contracts. Several high-profile reentrancy exploits have been documented [4]. Reentrancy attacks can have severe consequences, often leading to the complete loss of all funds within the targeted smart contract. These attacks involve deploying a malicious smart contract that exploits a vulnerability in another contract.

A contract is vulnerable to this type of attack if it has a function, callable by another smart contract, that has a particular unsafe order of actions as well as the ability to trigger the execution of another contract. Specifically, if a function modifies the contract state (e.g., transferring tokens), before updating that state, where in between those two actions, the logic from another smart contract can be triggered. The state change itself could be the part of the code that triggers the execution of the malicious smart contract, if, for example, the state change is transferring native currency. If the native currency is transferred to an address of another smart contract, this can trigger `receive()` or `fallback()` functions of the malicious smart contract upon native currency receival. The triggered function of the malicious smart contract would then call the same vulnerable function again. This would in turn form a loop of the actions that change the contract state, like transferring tokens, where these actions would be performed multiple times, and the update of the state change would be performed only once, when the loop is exited. This can result in the loss of all the tokens or native currency from the vulnerable smart contract's balance.

In the FERcoin smart contract, the only two functions that could trigger the execution of logic within another smart contract, are the FERcoin distribution functions that also transfer native currency: `transferFrom` and `batchTransferFrom`.

The functions that manage FERcoin tokens, such as the ERC20-inherited `transfer` function, cannot trigger the execution of another smart contract. Even if the custom ERC20 token is sent to another smart contract's address, the transfer of custom ERC20 token does not do anything outside of the scope of changing the original contract state by updating the balances in regards to transfer.

FERcoin distribution functions, on the other hand, not only distribute FERcoin tokens but also send a reimbursement of native currency. This reimbursement is intended to cover the gas fees when spending or transferring the received FERcoin tokens. If the recipient address belongs to another smart contract, this could trigger its `receive()` or `fallback()` function upon the receipt of reimbursement.

Even though FERcoin distribution functions can only be called by approved addresses with an allowance set by the contract owner, a student could still submit a smart contract's address instead of their personal wallet address to receive FERcoin. In such a case, the FERcoin tokens and native currency reimbursement would be sent to the smart contract. However, a smart contract receiving the FERcoin and reimbursement would not be able to call the distribution function again, as its address would not be pre-approved by the contract owner.

The following code sends the reimbursement to the recipient address:

```
(bool success, ) = recipient.call{value: reimbursementWei, gas:
    EOA_TRANSFER_GAS_LIMIT }("");
```

The gas limit for this transaction is defined by the constant `EOA_TRANSFER_GAS_LIMIT`, which is set as follows:

```
/// @notice Gas limit for reimbursement transfers to EOA addresses
uint256 public constant EOA_TRANSFER_GAS_LIMIT = 2300;
```

This constant is set to 2300, which is a sufficient gas limit for sending native currency to an externally owned account (EOA), but the transaction would fail if the address receiving the currency was that of another smart contract [11, 3]. This is because the gas limit defined is insufficient to execute a `receive()` or `fallback()` function of the smart contract that is receiving the native currency.

To further enhance security, the smart contract also implements a function to determine whether an address belongs to an EOA or a smart contract:

```
/**
* @notice Checks if an address is an Externally Owned Account (EOA)
* @dev Uses EXTCODESIZE to detect contract deployment. Note: Contracts
   under
*      construction (in constructor) will return false negative (size=0)
* @param account Address to check
* @return isEOA Boolean indicating EOA status (true = EOA, false =
   contract)
*/
function isEOA(address account) public view returns (bool) {
    uint256 size;
    assembly {
        size := extcodesize(account)
    }
    return size == 0;
}
```

This function uses `extcodesize`, which returns the size (in bytes) of the code stored at a given address. Since EOAs do not have stored code, the function returns zero for EOAs and a value greater than zero for deployed smart contracts.

There are two edge cases where this function could return zero incorrectly: If given the address of a smart contract that is being deployed at that moment, during the time that its constructor is executing. In that period, smart contracts code that is being deployed is not yet stored on chain, but its address is known. The other case is a smart contract that has called `selfdestruct`, removing its code. `extcodesize` function would return zero as codesize because all the code is removed. However, in this case, no contract logic remains that could be used for an exploit.

For this reason, this function is not used as the sole security check. Instead, it is combined with access control and the EOA gas limit.

This function is used inside `require` statements in all functions that need to confirm that the address given in the argument is an EOA address. The require statement is as follows:

```
require(isEOA(recipient), "Recipient must be an EOA");
```

Due to these multiple security measures, reentrancy attacks in the FERcoin smart contract are effectively prevented.

9.2.3. Input Validation

The most critical functions for input validation are public functions that modify the contract's state.

These functions are the `transfer` and `buyCoffee` function. Transfer function only implements the inherited virtual transfer function from ERC20 with added `whenNotPaused` modifier to disable the function if the contract is shut down, making it secure, as its logic is implemented by a known ERC20 standard.

The `buyCoffee` function implements custom logic. It takes two arguments: first is the ID of the cafe that the coffee is being purchased from; second is the unique bill number of the cafe bill for the bought coffee.

```
/**
 * @notice Purchase coffee using FERcoin
 * @dev Burns COFFEE_PRICE tokens and emits purchase event
 * @param spendLocationId ID of the cafe where coffee is purchased
 * @param billNumber Unique bill identifier from POS system (JIR on
    Croatian)
 */
function buyCoffee(uint256 spendLocationId, string calldata billNumber)
external whenNotPaused  returns (bool) {

    require(bytes(billNumber).length > 0, "Bill number must not be empty"
        );

    string memory spendLocation = cafes[spendLocationId];
    require(bytes(spendLocation).length > 0, "Invalid spend location");

    _burn(msg.sender, COFFEE_PRICE);
    emit CoffeeBought(msg.sender, spendLocation, billNumber);

    return true;
}
```

The function then burns one FERcoin token from the balance of the function caller using the internal `_burn` function from inherited ERC20.

The requires implemented in the function verify that the bill number is not undefined, to not spend a FERcoin token on nothing and that the ID of the cafe given in the argument maps to an existing cafe that is defined in the system. The inherited internal burn function already enforces that the caller has the needed FERcoin present in balance, that is being burned, and reverts the function execution if that is not satisfied. This function cannot be misused to harm the FERcoin system or other FERcoin holders. The only potential misuse is by the caller themselves, who could accidentally burn their own token without generating a valid transaction.

9.2.4. Shut Down Function

In cases where the contract owner's key pair is compromised or an approved professor's wallet is breached, leading to an illegal contract state, the contract can be shut down by the contract owner. Other similar severe incidents may also justify executing this function. The appropriate response steps in such scenarios are described in chapter 10.

The shutdown function is as follows:

```
/**
* @notice Emergency shutdown mechanism for the contract
* @dev Withdraws all native currency to specified EOA, pauses all
   pausable functions,
*       and permanently renounces ownership. This action is irreversible.
* @param withdrawTo Address to receive remaining native currency (must be
    EOA)
* @return success Boolean indicating whether shut down was successful
* Requirements:
* - Can only be called by contract owner
* - `withdrawTo` must be an Externally Owned Account (EOA)
*
* After execution:
* - All functions with whenNotPaused modifier will be disabled
* - Ownership is permanently renounced
* - Contract becomes immutable, read only
*
* Safety features:
* - Uses exactly 2300 gas for transfer to ensure compatibility with EOAs
* - Works even with zero balance in contract
* - Fails if recipient does not receive funds
*/
function shutDown(address withdrawTo) external onlyOwner returns (bool) {

    require(isEOA(withdrawTo), "withdrawTo must be EOA");

    uint256 balance = address(this).balance;

    // Send all remaining native balance of the contract to an EOA
       withdrawTo address
    (bool sent, ) = withdrawTo.call{value: balance, gas:
       EOA_TRANSFER_GAS_LIMIT}("");

    require(sent, "Native currency transfer failed");

    _pause();         // Disable all pausable functions
    renounceOwnership(); // Make shutdown irreversible

    emit ShutDown(withdrawTo, balance, sent);

    return true;

}
```

The shutdown function follows a structured process:

- Transfer all remaining native currency to a specified Externally Owned Account (EOA)
- Pause all state-changing contract functions (functions with `whenNotPaused` modifier)
- Renounces ownership permanently, making the contract immutable and read-only

Since only the contract owner can unpause the contract, renouncing ownership ensures that it remains permanently paused and immutable. All state-changing functions are disabled, making the contract effectively read-only. This is done so that, during the recovery, the newly deployed contract can be returned to the same state of the compromised contract as at the moment it was shut down.

Excluding Shutdown Function UI Mapping

The shutdown function is intentionally not accessible through the FERcoin administrator interface. This design choice ensures that:

- Accidental execution (e.g., user error) is prevented
- Attacks targeting the admin UI (such as XSS exploits) cannot trigger the shutdown process
- In the severe breach cases, the application is not relied on, which can also be compromised

Attackers have a strong incentive to target this function, because this function also withdraws all the remaining native currency of the contract to the specified address.

The administrator should only execute the shutdown function manually using a dedicated script that does not store the owner's key pair but takes it as an argument. Alternatively, they can use on-demand contract interaction tools like Remix IDE to generate an interface and execute the function securely using the owner key pair.

10. Incident Response and Attack Scenarios

This chapter discusses potential attack scenarios and the appropriate incident response measures to mitigate their impact.

10.1. Administrator Account or Server Breach

If an attack is detected that suggests a potential breach of the web application or server, either through unauthorized access to the administrator account or direct shell access to the server, the immediate priority is to revoke the allowances of all approved professor wallets on the smart contract using the contract owner's master key.

Once the allowances are revoked, the next step is to shut down the application to prevent further exploitation. The primary sensitive data at risk includes system-stored wallets on the server and the API keys of the integrated service providers. If the wallet allowances are successfully removed in time, the risk of unauthorized transactions is mitigated.

Following the shutdown, all API keys associated with external providers must be deactivated. Since these keys are only used to send signed transactions to the blockchain and retrieve data, they cannot be exploited for unauthorized transactions but could still be misused to exhaust the allocated request limits.

After securing the system, log analysis should be conducted to determine the methods and vulnerabilities exploited by the attacker. Identifying the attack vector is essential for preventing future breaches.

If no server-stored wallet with an active allowance was compromised before the administrator revoked permissions using the master key, no FERcoin or native currency from any wallet would be at risk, and all blockchain data would remain intact.

Once the vulnerability is identified and resolved, the patched application can be redeployed. New user accounts must be created with fresh passwords, and new professor wallets must be generated and approved using the master key. Additionally, the application should be reconfigured with new provider API keys. The newly deployed, patched web application will then reconnect to the existing deployed smart contract, restoring the system to its pre-incident state.

10.1.1. Breach of Server Stored Wallet

If a malicious actor gains access to a server-stored approved wallet, decrypts the private key, and uses it before the administrator revokes its allowance with the master key, the attacker can fully utilize the FERcoin allowance assigned to that compromised wallet. The extent of potential damage depends on whether the FERcoin tokens were minted and available in the master wallet's balance at the time of the attack. Approved wallets can only spend FERcoin from the master wallet's balance if those tokens have been minted and exist at the moment of transaction.

If the stolen FERcoin tokens were used and transferred to another address, the smart contract should be shut down using the owner-only shutdown function described in Chapter 9.2.4. During the shutdown process, a designated address must be provided to receive all remaining native currency from the smart contract.

After the shutdown, a new instance of the smart contract can be deployed, restoring the system to the last recorded valid state before the breach. The recovery process involves redistributing FERcoin tokens to all previous holders on the new contract, ensuring their token balances match those before the shutdown.

As a result, the FERcoin transaction history will be divided into two distinct parts: the valid transactions recorded on the compromised contract up until the moment of shutdown, and the transactions executed on the newly deployed contract after the system's restoration.

If any stolen FERcoin tokens related to the incident were used for the coffee purchase, the result will be a monetary loss, as the coffees were purchased with stolen FERcoin tokens.

10.2. Master Key Breach

A breach of the master key represents the most severe security incident. The only way this could occur is if the administrator responsible for securely storing the master key failed to do so, allowing the attacker to obtain both the private and public keys of the master wallet.

In this scenario, the master key must be used immediately to transfer contract ownership to a new address, provided that the attacker has not already done so. If unauthorized FERcoin tokens were minted or stolen, the smart contract should be shut down. If the shutdown is successful, the recovery process follows the same procedure outlined in Section 10.1.1.

By the time the breach is detected, the attacker has likely withdrawn all available native currency from the deployed contract, as well as any funds from the compromised master wallet, transferring them to another address. These stolen funds cannot be recovered.

If it is not possible to change the contract owner or shut down the contract due to the complete depletion of the master wallet's native currency, any additional funds sent to the wallet for recovery purposes risk being immediately stolen. In such a case, the best course of action is to shut down the application entirely and notify all system participants to refrain from using it until the recovery process is completed.

To restore the system, a new smart contract and application instance must be deployed. The new application should connect to the newly deployed smart contract, and all previous transactions and FERcoin token distributions must be replicated to reflect the last valid state of the compromised contract.

10.3. Changing Smart Contract Address to a Malicious Smart Contract

If a malicious actor gains access to the administrator account, they can modify the blockchain connection settings to redirect the application to a malicious smart contract.

The goal of this attack is to deploy a fraudulent smart contract and alter the application's configuration to connect to it instead of the legitimate FERcoin contract. A malicious smart contract could include functions that compromise the funds in students' wallets when they interact with the FERcoin system.

The extent of the attack depends on whether the attacker has full server access or only access to the administrator account. With unrestricted server access, the attacker can modify the HTML and client-side JavaScript files served to students, making the attack more effective. If the attacker only has administrator access, they can still alter the blockchain configuration, changing the contract address used for FERcoin transactions such as purchasing coffee or transferring tokens.

To execute this attack, the attacker must create a malicious smart contract that closely resembles the FERcoin contract while maintaining the same ABI interface for transaction functions. This ensures that the existing client-side logic continues generating transactions as expected, without any visible discrepancies in the user interface. If the attacker has full server access, disguising the fraudulent transaction becomes even easier.

In both cases, if the attack is successful, the malicious modifications will not be apparent in the user interface. However, before submitting a transaction, the MetaMask extension will display a confirmation dialog showing the transaction details. The attacker cannot modify this step, meaning students will always have the opportunity to review the transaction before signing. Although the attacker can try to make the fraudulent transaction appear legitimate, they cannot conceal the contract address the transaction interacts with.

Because of this, the best defense against such an attack is educating students on the proper use of the FERcoin system, specifically how to verify transactions before signing them.

To mitigate the impact of such an attack, all students should be required to read a guide on using the FERcoin system. This guide should emphasize several security measures:

– Students should use a dedicated blockchain wallet exclusively for FERcoin transactions, separate from any other wallets they own.

– The FERcoin wallet should not hold other cryptocurrencies, except for a small amount of native currency required for gas fees. This native currency will be provided as part of the FERcoin distribution process.

– Before signing any transaction in MetaMask, students should always verify the contract address the transaction interacts with. The official FERcoin contract address should be publicly available and independent of the application to allow for easy verification.

– The transaction details in the MetaMask pop-up should match the intended action. Any request to approve permissions or transfer a significant amount of native currency should be treated with suspicion.

By following these security guidelines, students can better protect themselves from falling victim to this type of attack.

11. Further FERcoin System Development

This chapter discusses some ideas, expansions and functionalities that could be used to expand and better the FERcoin system.

11.1. FERcoin Token Exchange Smart Contract Function

With the growth of the FERcoin system and expansion of goods available to be exchanged for the FERcoin, the current smart contract function `buyCoffee` for exchanging one FERcoin for one coffee, can be reimplemented to a more universal function instead, that will enable the users to exchange their FERcoin for any of the available goods, and also the option to do multiple exchanges in a single transaction can be added. This means the user can spend multiple FERcoin tokens to exchange them for multiple items in a single transaction.

11.2. Student Interface Improvements

Given that students will primarily use FERcoin interface on their phones to exchange FERcoin tokens for coffee, the user interface should be adapted for mobile use. This includes implementing an additional mobile-friendly view, besides the current desktop one.

The transaction of buying a coffee requires the input of the unique bill number (Cro. JIR - Jedinstveni identifikator računa). This number can currently only be manually entered. Since this number is embedded in the QR code on invoice bills, a QR code scanning feature should be implemented. When using the web application on a phone, users could scan the QR code with their camera, while desktop users could upload an image of the bill for automatic extraction of the unique bill number from the QR code. This would enhance user experience by eliminating manual input errors.

Additionally, the student guide should include instructions on installing and using the MetaMask software wallet on a mobile phone for connecting with the student's interface.

11.3. Hosting and Achieving Production Ready State

For real-world deployment, an appropriate hosting solution must be determined. While the backend logic is mostly simple and uses Python, permanent stable storage is recommended, which should be separated from the application itself. Currently, while in development, both the application and permanent storage (SQLite and JSON configurations and wallets) are contained within a single Docker container, with some of the application state stored in memory.

When in production, a good approach would be to have a separate application and permanent server storage, like a database, where all users of the application, as well as the configurations, wallets and all of the application state, can be stored. This approach allows some security features, like Flask-limiter which currently uses application memory and sessions implemented as client side encrypted cookies to also use that permanent storage.

This would allow the application to be more stable, preserving state in cases of restart or some critical errors, while also making the scaling of the application much easier and efficient, through the usage of multiple workers or multiple application instances that all use the same separate permanent storage.

11.3.1. Expanding FERcoin System with Other Blockchain Capabilities

Some other blockchain capabilities could be utilized to expand the uses and functionalities of the FERcoin system. One such example are the NFTs. They currently hold some negative connotations but can be used and implemented in a different way than the use-case they are most known for.

Each student would have an academic blockchain identity, represented by a unique blockchain address. This identity could be used to collect NFTs awarded for completing optional academic activities, such as skill courses, additional lectures, or other extracurricular achievements. These NFTs would serve as immutable proof of achievement rather than as tradable assets.

Since all the student's blockchain identities are pseudonymous, meaning their academic blockchain address cannot be connected to their real-life identity in a simple way, there would be an option for generating a "Proof of ownership". A student could generate proof that they are the owner of their academic blockchain identity, which he would then disclose as he wishes. That way the student could showcase all the additional achievements he has accomplished as he acquired NFTs.

Since student blockchain identities remain pseudonymous by default, an optional 'Proof of Ownership' mechanism could allow students to verify their blockchain identity when needed. This would enable them to selectively disclose their academic blockchain identity, making it possible to showcase their additional academic achievements to employers or institutions while maintaining privacy.

By building a blockchain-based academic portfolio, students would have a verifiable and tamper-proof record of their extracurricular accomplishments, further incentivizing engagement in skill development and additional activities beyond standard coursework.

12. Conclusion

This thesis has presented the development of the FERcoin system, demonstrating the approach taken in structuring, integrating, and securing system components when working with blockchain technology. It also highlights the key challenges encountered during development and the security measures implemented to protect the system. In addition, the decision-making process behind the selection of appropriate technologies and blockchain networks is documented, providing a reference for future projects involving blockchain-based systems. The system can be easily brought to a production-ready state once the appropriate hosting environment is selected and the specifics of the use case are determined. Integrated into the academic environment, it would provide a universal mechanism for incentivizing students to pursue additional academic achievements in designated fields. Beyond its academic application, the FERcoin system has the potential to be expanded or modified for similar use cases, serving as a foundation for other blockchain-based initiatives. Whether as a guide for system design or through the reuse of existing components, the system can contribute to further academic research and development in blockchain technology. Ultimately, FERcoin is designed to foster innovation, encourage student engagement, and promote continued exploration in the blockchain field.

BIBLIOGRAPHY

[1] Tom Abai. The solana web3.js incident: Another wake-up call for supply chain security. *Mend.io Blog*, December 2024. URL `https://www.mend.io/blog/the-solana-web3-js-incident-another-wake-up-call-for-supply-chain-s` Accessed January 2025.

[2] Frederik Braun. Sri hash generator, 2025. URL `https://www.srihash.org/`. Accessed January 2025. Source code available at `https://github.com/mozilla/srihash.org`.

[3] Vitalik Buterin. Some medium-term dust cleanup ideas. *Ethereum Magicians*, 2021. URL `https://ethereum-magicians.org/t/some-medium-term-dust-cleanup-ideas/6287`. Accessed December 2024.

[4] ChainWall. Reentrancy attack in smart contracts. *Medium - ChainWall*, 2023. URL `https://medium.com/chainwall-io/reentrancy-attack-in-smart-contracts-4837ed0f9d73`. Accessed November 2024.

[5] Simeon Cholakov. Defi front-end exploits. *Three Sigma Blog*, January 2025. URL `https://threesigma.xyz/blog/defi-front-end-exploits`. Accessed January 2025.

[6] Cyfrin. *Iterable Mapping*. Cyfrin, 2024. URL `https://solidity-by-example.org/app/iterable-mapping/`. Accessed November 2024.

[7] Solidity Developers. *Security Considerations*. Solidity Documentation, 2023. URL `https://docs.soliditylang.org/en/develop/security-considerations.html`. Accessed November 2024.

[8] Flask-Limiter Documentation. Flask-limiter documentation. *Flask-Limiter*

ReadTheDocs, 2024. URL `https://flask-limiter.readthedocs.io/en/stable/`. Accessed November 2024.

[9] Flask-WTF Documentation. Csrf protection in flask-wtf. *Flask-WTF ReadTheDocs*, 2023. URL `https://flask-wtf.readthedocs.io/en/1.2.x/csrf/`. Accessed November 2024.

[10] WTForms Documentation. form data validation in wtforms. *WTForms ReadTheDocs*, 2024. URL `https://wtforms.readthedocs.io/en/3.2.x/`. Accessed November 2024.

[11] Ethereum Stack Exchange. Gas cost for ether transfer to smart contract from eoa. *Ethereum Stack Exchange*, 2024. URL `https://ethereum.stackexchange.com/questions/82210/gas-cost-for-ether-transfer-to-smart-contract-from-eoa`. Accessed December 2024.

[12] Ethereum Foundation. *Solidity Documentation*. Solidity Documentation, 2024. URL `https://docs.soliditylang.org/en/v0.8.28/`. Accessed October 2024.

[13] Steve Marx. *Ethereum Smart Contract Storage*. ProgramTheBlockchain, March 2018. URL `https://programtheblockchain.com/posts/2018/03/09/understanding-ethereum-smart-contract-storage/`. Accessed November 2024.

[14] Mozilla Developer Network. *Subresource Integrity (SRI)*, 2025. URL `https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity`. Accessed January 2025.

[15] OpenZeppelin. *OpenZeppelin Documentation*. OpenZeppelin, 2024. URL `https://docs.openzeppelin.com/`. Accessed October 2024.

[16] OWASP. Owasp smart contract top 10. *OWASP*, 2025. URL `https://owasp.org/www-project-smart-contract-top-10/`. Accessed February 2025.

[17] Eric Shek. *Batch Ether and ERC-20 Transfer Method*. Mirror.xyz, January 2024. URL `https://mirror.xyz/ericxstone.eth/rzgmN_gp-ADEuponhQeAGk7AA8VrNGof8_cPSqEsxvM`. Accessed November 2024.

[18] Corwin Smith. *ERC20 Standard Definition*. Ethereum.org, June 2024. URL `https://ethereum.org/en/developers/docs/standards/tokens/erc-20/`. Accessed October 2024.

[19] Liran Tal. Polyfill supply chain attack embeds malware in javascript cdn assets. *Snyk Blog*, June 2024. URL `https://snyk.io/blog/polyfill-supply-chain-attack-js-cdn-assets/`. Accessed January 2025.

[20] Toni Wahrstätter. *On Block Sizes, Gas Limits and Scalability*. Ethereum Research, January 2024. URL `https://ethresear.ch/t/on-block-sizes-gas-limits-and-scalability/18444`. Accessed November 2024.

[21] Kartik Nayak Yinhong (William) Zhao. *EIP-1559 In Retrospect*. Decentralized Thoughts, March 2022. URL `https://decentralizedthoughts.github.io/2022-03-10-eip1559/`. Accessed November 2024.

Application for encouraging engagement in the academic environment based on blockchain technology and cryptocurrencies

Abstract

This paper explores the potential application of blockchain technology and cryptocurrencies in an academic environment through the development of the FERcoin system. The goal of the system is to incentivize students for academic development and introduce them to blockchain technology through a rewards-based system. FERcoin is designed as a digital currency that can be exchanged for tangible assets, such as coffee in a café, ensuring its usability and practicality. The paper covers the selection of blockchain technology and platform, smart contract development, web application implementation, and security solutions required to protect users and the system. Further possibilities for system expansion and additional functionalities are also considered. The proposed system examines the potential of blockchain technology in an academic context and its applicability in real-world use cases.

Keywords: blockchain, Web3, smart contract, FERcoin, cryptocurrency, tokenization, student incentives, system security, Flask, Docker

Aplikacija za poticanje sudionika u akademskom okruženju zasnovana na tehnologiji ulančanih blokova i kriptovalutama

Sažetak

Ovaj rad istražuje mogućnost primjene blockchain tehnologije i kriptovaluta u akademskom okruženju kroz razvoj FERcoin sustava. Cilj sustava je potaknuti studente na akademski razvoj i upoznavanje s blockchain tehnologijom putem sustava nagrađivanja. FERcoin je dizajniran kao digitalna valuta koja se može zamijeniti za materijalne vrijednosti, poput kave u kafiću, čime se osigurava njegova upotrebljivost i praktičnost. Rad pokriva izbor blockchain tehnologije i platforme, razvoj pametnog ugovora, implementaciju web aplikacije te sigurnosna rješenja potrebna za zaštitu korisnika i sustava. Također se razmatraju daljnje mogućnosti proširenja sustava i implementacije dodatnih funkcionalnosti. Predloženi sustav ispituje potencijal blockchain tehnologije u akademskom kontekstu i njezinu primjenjivost u stvarnim uvjetima.

Ključne riječi: blockchain, Web3, pametni ugovor, FERcoin, kriptovaluta, tokenizacija, poticanje studenata, sigurnost sustava, Flask, Docker