LISBON SCHOOL OF ENGINEERING

Department of Electronical Engineering, Telecommunications and Computers

Ticket management system using Blockchain technology

Rodrigo Filipe Leitão Dias

Bachelor's

Project Work to obtain the masters degree
in Informatics and Multimedia Engineering

Adviser : PhD Carlos Gonçalves

Jury:

President: [Grau e Nome do presidente do juri]

Vogal: [Grau e Nome do primeiro vogal]

September, 2024

LISBON SCHOOL OF ENGINEERING

Department of Electronical Engineering, Telecommunications and Computers

Ticket management system using Blockchain technology

Rodrigo Filipe Leitão Dias

Bachelor's

Project Work to obtain the masters degree
in Informatics and Multimedia Engineering

Adviser : PhD Carlos Gonçalves

Jury:

President: [Grau e Nome do presidente do juri]

Vogal: [Grau e Nome do primeiro vogal]

September, 2024

Abstract

The traditional ticketing industry faces challenges such as ticket scalping, fraud, and limited transparency. Blockchain technology has the potential to revolutionize ticketing by offering a secure, transparent, and efficient solution. This thesis proposes a blockchain-based ticketing system that leverages the core strengths of blockchain technology to address these shortcomings.

The system uses smart contracts to manage the ticket lifecycle securely, from creation by event organizers to purchase and transfer by users. This ensures authenticity and eliminates the risk of counterfeiting. Additionally, the system facilitates a secure and transparent secondary market for ticket resale, with fair pricing mechanisms and clear ownership tracking.

Keywords: Blockchain, Ticketing System, NFTs

Resumo

A indústria tradicional de bilhetes enfrenta desafios como a revenda ilegal de bilhetes, fraude e transparência. A tecnologia blockchain tem o potencial de revolucionar o setor de bilheteiras ao oferecer uma solução segura, transparente e eficiente. Esta tese propõe um sistema de bilheteira baseado em blockchain que aproveita as principais capacidades desta tecnologia para resolver esses problemas.

O sistema utiliza *smart contracts* para gerir de forma segura o ciclo de vida dos bilhetes, desde a criação pelos organizadores de eventos até à compra e transferência pelos utilizadores. Isto garante a autenticidade e elimina o risco de falsificação. Além disso, o sistema facilita um mercado secundário seguro e transparente para a revenda de bilhetes, com mecanismos de preços justos e um rastreio claro de propriedade.

Palavras-chave: Blockchain, Sistema de Bilheteira, NFTs

Contents

List of Figures

List of Tables

List of Listings

Acronyms

DApps Decentralized Applications. 27
DeFi Decentralized Finance. 30

ECDSA Elliptic Curve Digital Signature Algorithm. 28
EVM Ethereum Virtual Machine. 29

IPFS InterPlanetary File System. 53, 61, 62

NFTs Non-Fungible Tokens. 5, 7, 25

PoH Proof of History. 29
PoS Proof of Stake. 28
PoW Proof of Work. 28

RSA Rivest-Shamir-Adleman. 28

1

Introduction

Concerts and festivals play a big role in peoples life's, allowing them to create memorable experiences watching live performances from their favorite artists. Those are the kinds of memories that last for life, so every event organizer wants to ensure that the whole process works seamlessly, from the end user to the entire background planning of the event.

The process of organizing an event starts with the event organizer, who is responsible for the whole planning of the event, from the venue to the artists, to the marketing and ticketing. The event organizer is the one who takes the risk of organizing the event, and the one who will profit from it, and can be a company, a group of people, or even a single person, whom will hire the artists, the venue, the security, the marketing, and the ticketing.

However, there's an issue with the ticketing process, which is the scalping. Scalping is the process of buying tickets in bulk, exploiting high demand, and reselling them at significantly inflated prices. This not only disadvantages for people that genuinely want to attend but also undermines the integrity of the ticketing system. Traditional ticketing platforms often rely on centralized databases and intermediaries, providing opportunities for scalpers to manipulate the system and engage in fraudulent activities. Moreover, existing ticketing systems frequently encounter issues related to security, trust, and reliability. Centralized databases are vulnerable to cyber attacks, leading to unauthorized access, data breaches, and the manipulation of ticketing information. Trust in the authenticity of tickets and the reliability of transactions is compromised,

creating a pressing need for innovative solutions that can address these inherent challenges.

That's where blockchain comes in. Blockchain technology, renowned for its decentralized and transparent nature, presents a compelling solution to revolutionize the ticketing industry. By leveraging blockchain, it becomes possible to create a secure and tamper-proof ledger of transactions, mitigating the risk of scalping and ensuring the integrity of the ticketing process. The use of smart contracts further automates transactions, reducing the reliance on intermediaries, therefore extra costs, and enhancing operational efficiency. This allows the event organizer to have a more secure and reliable ticketing process, and the end user to have a more transparent and fair ticketing process.

1.1 Motivation

The motivation for this work is primarily to avoid ticket scalping. By using blockchain, it's possible to create a system that prevents any kind of unwanted operations because of the properties of smart contracts that enforce a certain behavior, allowing for users to resell a ticket, but not for a price higher than the original price. This is a way to guarantee that the end user will have a fair and transparent ticketing process. For this to happen, the idea is to have a marketplace where users are able to resell their tickets, enforcing a price cap on them.

Another important feature is the possibility of having partial refunds. This is something that is not always available in traditional ticketing platforms, and when it is, it's not easy to do. With blockchain, it's possible to have a system that allows for easy and instant refunds, without the need for intermediaries. Enabling a feature like this can actually be advantageous for the organizers, if they're expecting the venue to be full. This way, when users buy their tickets and then realize they can't attend, they have a reason to refund, making that ticket available again for the original price, making a profit for the organizer.

1.2 Objectives

We will have a website that will allow Ticketchain to approve organizers into our system to be able create events, so that no random entity can create freely, which would

lead to spam. This would also be for the event organizers themselves where, if allowed, they would be able to manage their events, and manage admins and validators associated to them.

There will also be an app for the users to check the events and manage their tickets. The events shown are gonna be the ones stored in our Ticketchain system. They will also have access to the marketplace, where they will be able to resell their tickets for a price no higher than the original.

For the validators, we will have yet another app that allows them to validate user tickets at the entrance of the venue. These validators are selected by the organizer for the event, and their job is to truthfully verify the tickets, without any chance of fraud. These validators can be either a person with their app to operate or even some automated machine, like a turnstile.

1.3 Contributions

The components implemented can be found on the GitHub. The repository for the frontend is stored at Ticketchain frontend repository [10] and the blockchain smart contracts code on Ticketchain backend repository [2], which was deployed on the Base Sepolia Testnet at BaseScan [1] so you can interact directly with.

1.4 Document Structure

The document is structured in the following way: **Chapter 1: Introduction** provides an overview of the project, including its context, motivation, objectives, and contributions; **Chapter 2: Background and Related Work** introduces essential concepts and technologies that serve as the foundation for the project; **Chapter 3: Requirements Analysis** outlines the functional and non-functional requirements of the project as well as the use cases and the architecture; **Chapter 4: Implementation** describes the implementation of the project, including the technologies used; **Chapter 5: Results** presents the results of the project and how it meets the requirements; **Chapter 6: Conclusions** summarizes the project, reflecting on the achievements and limitations, and suggesting future work.

2

Background and Related Work

This Chapter presents the background and related work for the system. The background in the Section 2.1 will present the necessary information, like the blockchain concepts and the fundamentals of how the entire system works. The related work in the Section 2.2 will present projects with similarities to the system and how NFTs are more commonly used.

2.1 Background

Blockchain is a relatively recent technology that can be complex to grasp. At its core, it relies on cryptographic principles to ensure secure and transparent transactions. In this section, we will explain key blockchain concepts, including how wallets work and interact with the blockchain, in Sections 2.1.1 and 2.1.2, while also exploring prominent blockchain networks, token standards, and the rise of NFTs. Figure 2.1 provides a visual overview of these concepts, which will be detailed in the following subsections: Wallets **(1)** in Section 2.1.3, Networks **(2)** in Section 2.1.4, and Smart Contracts **(3)** in Section 2.1.5. We'll also explain more technical concepts in the scope of this project, like Solidity **(4)** in Section 2.1.6, Token Standards **(5)** in Section 2.1.7, and NFTs **(6)** in Section 2.1.8.

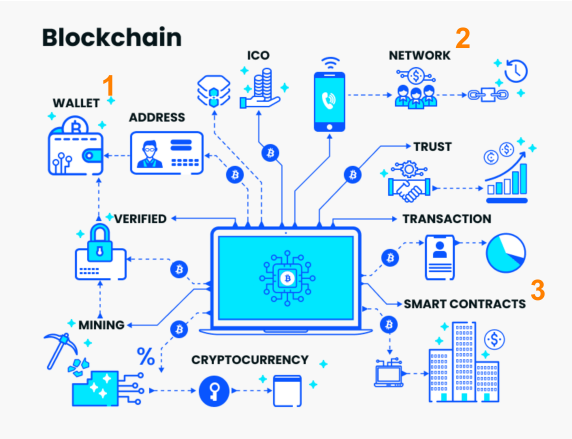

Figure 2.1: Key concepts of blockchain technology. Adapted from [3]

2.1.1 Interacting with the Blockchain

A blockchain operates as a decentralized network of nodes, each maintaining a full copy of the blockchain. To interact with the blockchain, users send transactions to the network, which are validated and added to blocks by nodes. This process requires a wallet—an application that allows users to manage digital assets, interact with smart contracts, and initiate transactions. Wallets provide a user-friendly interface to sign transactions, manage keys, and view account balances. Figure 2.2 illustrates the steps involved in a blockchain transaction.

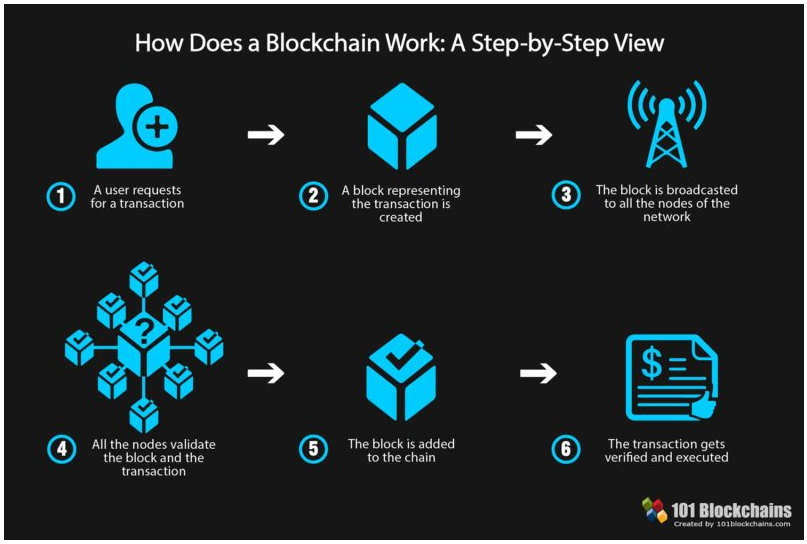

Figure 2.2: Overview of blockchain transaction processes. Extracted from [11]

To execute a transaction, the user must sign it with their private key, proving ownership of the assets being transferred (step 1). The transaction is then broadcasted to

the network (steps 2 and 3), validated by nodes (step 4), and added to the blockchain ledger, where it becomes immutable and accessible to all network participants (steps 5 and 6).

While transactions require signatures to alter the blockchain, reading data from the blockchain is signature-free; users can freely query the blockchain for information.

2.1.2 Blockchain

Blockchain is a decentralized and distributed ledger technology that enables secure and transparent data sharing across a network. A blockchain consists of a series of blocks, each containing a list of transactions. These blocks are linked in chronological order, forming a tamper-proof record. Key characteristics of blockchain technology include:

Decentralization: Blockchain operates on a decentralized network of nodes, eliminating the need for central control and reducing points of failure.

Transparency: Data on the blockchain is visible to all participants, fostering trust and accountability, as transactions can be verified independently.

Immutability: Once recorded, transactions cannot be altered or deleted, creating an irreversible and secure historical record through cryptography.

Security: Blockchain uses advanced cryptographic techniques to secure transactions, with network participants validating and verifying transactions via consensus mechanisms.

Smart Contracts: Smart contracts are self-executing programs stored on the blockchain that automate agreements between parties without intermediaries, enabling the creation of Decentralized Applications (DApps).

Blockchain has potential applications across numerous industries, including finance and healthcare. By decentralizing control and enhancing privacy, blockchain is often seen as the foundation of Web3, the next phase of the internet.

A unique aspect of blockchain is its use of incentives. Nodes are rewarded with cryptocurrency for validating transactions, creating a system that aligns self-interest with network integrity, ensuring a majority of honest participants.

2.1.3 Wallets

Cryptocurrency wallets are essential for interacting with blockchain networks and managing digital assets. Key cryptographic features of wallets include:

Private and Public Keys: Wallets generate a pair of cryptographic keys: a public key (address) for receiving assets, and a private key for signing transactions. This relationship is based on asymmetric cryptography, ensuring that only the owner can authorize transactions.

Digital Signatures: Transactions are signed using the private key, providing proof of authorization. Signatures are generated via cryptographic algorithms like Elliptic Curve Digital Signature Algorithm (ECDSA) or Rivest-Shamir-Adleman (RSA), depending on the blockchain.

Hash Functions: Wallets use cryptographic hash functions to create unique transaction hashes, ensuring data integrity and preventing tampering.

Seed Phrases: Some wallets offer seed phrases as a backup to recover access if the private key is lost. These phrases are generated from random words and serve as a human-readable form of the private key.

Using these cryptographic tools, wallets securely manage and facilitate transactions on blockchain networks, ensuring confidentiality, integrity, and authenticity.

2.1.4 Networks

Since the introduction of Bitcoin in 2009, various blockchain networks have emerged, each with unique features. Some prominent networks include:

Bitcoin (BTC): The first and most widely known cryptocurrency, Bitcoin uses a Proof of Work (PoW) consensus mechanism to secure peer-to-peer transactions.

Ethereum (ETH): Ethereum is a decentralized platform that supports smart contracts and DApps. It is transitioning from PoW to Proof of Stake (PoS) with Ethereum 2.0, improving scalability and reducing energy consumption.

Polygon (MATIC): Polygon is a Layer 2 scaling solution for Ethereum, offering faster and cheaper transactions by utilizing sidechains.

Solana (SOL): Solana is designed for high-performance DApps, using a combination of Proof of History (PoH) and PoS to achieve high transaction throughput with low latency.

Ethereum and its compatible networks (like Polygon) use Solidity, a popular language for smart contracts. Many other networks are also compatible with the Ethereum Virtual Machine (EVM), enabling cross-network deployments with the same code.

2.1.5 Smart Contracts

Smart contracts are autonomous programs stored on the blockchain that automatically execute when predefined conditions are met. Key properties include:

Autonomy: Smart contracts operate without human intervention once deployed, ensuring impartial enforcement of terms.

Trust: Because smart contracts run on decentralized, immutable blockchains, parties can rely on their execution without intermediaries.

Security: Once deployed, smart contracts cannot be altered, providing a secure and reliable platform for digital agreements.

Efficiency: Automating contract execution reduces transaction costs and processing times, increasing overall efficiency.

Versatility: Smart contracts manage digital assets, execute complex operations, and enable a variety of decentralized applications.

2.1.6 Solidity

Solidity is the leading language for developing smart contracts on EVM-compatible blockchains like Ethereum. With syntax similar to C++, it offers specific features suited for blockchain development.

Public Functions: Public functions in Solidity can be called by anyone, requiring careful access control to prevent unauthorized use.

Modifiers: Modifiers in Solidity allow developers to control access to functions, protecting contracts from security vulnerabilities like reentrancy attacks.

Currency: Solidity handles currency in the smallest unit of ether, called wei. For example, 1 ether equals 10^{18} wei.

Addresses: Solidity uses a specific address type to manage users and smart contracts, treating both similarly except that contracts can execute functions and store data.

2.1.7 Token Standards

Token standards define the rules for digital tokens on blockchain networks. The most common standards include:

ERC-20: The most widely used standard for fungible tokens, enabling seamless trading on exchanges and easy integration into Decentralized Finance (DeFi) applications.

ERC-721: A standard for NFTs, used for unique assets like digital art and collectibles.

ERC-1155: A flexible standard that supports both fungible and non-fungible tokens within a single contract, reducing deployment costs and complexity.

2.1.8 Non-Fungible Tokens (NFTs)

NFTs hold great potential for event ticketing systems, offering secure and verifiable ways to issue, transfer, and validate tickets. NFT-based tickets can include metadata such as event details, seat numbers, and access privileges, creating a customizable experience for both organizers and attendees. NFTs also enable the creation of limited-edition tickets with special privileges for VIP holders.

2.2 Related Work

This section reviews some of the leading traditional ticket-selling platforms and their functionalities, followed by an exploration of the application of NFTs in various industries, including art and gaming, highlighting key projects and emerging trends.

2.2.1 Traditional Ticket-Selling Platforms

Several established platforms are widely used for ticket sales. Two of the largest and most reputable Portuguese companies in this space are Ticketline [16] and Blueticket [4]. These platforms cater to a wide range of events, such as concerts, sports matches, theater performances, and exhibitions. In addition, they collaborate with physical retailers, including Worten [18], Fnac [8], and El Corte Inglés [6], providing users with the flexibility to purchase tickets both online and in person.

Both platforms feature comprehensive websites where event organizers can list their events. Users can browse upcoming events, access detailed event information, and purchase tickets online, with options to print tickets at home or pick them up at designated physical outlets.

2.2.2 Application of NFTs

NFTs have gained significant traction in industries such as art and gaming, where they represent ownership of unique digital assets. In the art industry, NFTs enable creators to mint and sell digital artworks, which are authenticated and tracked on the blockchain, ensuring their uniqueness and provenance. NFT artworks can be traded on various platforms, with many artists benefiting from royalties generated from secondary sales.

In gaming, NFTs are utilized to represent unique in-game assets, such as skins, weapons, or characters. These items can be traded between players, adding a new dimension to gameplay and player ownership. One of the most notable NFT projects is the Bored Ape Yacht Club [5], a collection of digital avatars in JPEG format that have sold for millions of dollars. Beyond digital ownership, membership in the Bored Ape Yacht Club offers exclusive benefits, such as access to private events, merchandise, and a dedicated community of collectors.

3

Requirements Analysis

This Chapter presents the requirements analysis for the system and is divided into 3 sections: Requirements in Section 3.1 where it presents the functional and Non-Functional Requirements; Use Cases in Section 3.2; and Architecture in Section 3.3.

3.1 Requirements

This section outlines the system requirements, categorized into functional and non-functional requirements. The functional requirements, described in Section 3.1.1, define the specific features and behaviors the system must support. In contrast, the non-functional requirements, covered in Section 3.1.2, address aspects like performance, security, and user experience.

3.1.1 Functional Requirements

Table 3.1 lists the functional requirements along with their respective descriptions, outlining the core functionalities that the system must implement.

Table 3.1: Functional Requirements

Requirement	Description
Wallet software	The system must connect to a wallet software to interact with the blockchain, enabling the signing of transactions.
Smart contract interaction	The system must interact with the smart contract to display event-related information and update the status of events or tickets.
File storage system	The system must be able to upload the NFTs metadata to a file storage system, to store and retrieve them.

We can see that the system's core functionalities revolve around the interaction with the blockchain. The system must connect to a wallet software to sign transactions, interact with the smart contract to manage events and tickets, and store NFT metadata in a file storage system. These requirements are essential for the system to function as intended and provide a seamless experience for users.

3.1.2 Non-Functional Requirements

Table 3.2 presents the non-functional requirements, defining the system's performance, security, scalability, and overall quality to ensure an optimal user experience.

Table 3.2: Non-Functional Requirements

Requirement	Description
Scalable	The system must scale to accommodate a large number of users and events without performance degradation.
Low Fees	The system must minimize operational fees to provide cost-efficient transactions.
Fast	The system must ensure fast processing times to offer a seamless experience for users during events.
Secure	The system must ensure security to prevent fraud or unauthorized access.
User-Friendly	The system must be intuitive and easy to use, facilitating the buying and selling of tickets.

The requirements for scalability, low fees, and speed are largely influenced by the

choice of the blockchain network. As discussed in Section 2.1.4, each blockchain network offers different characteristics, and selecting the right one is crucial for meeting these requirements. For now, since we'll be using testnets (networks with fake funds), this study can be done later when deploying to a real network. Meanwhile, security and user-friendliness are more dependent on the system's design and implementation. The system must be engineered to prevent fraud and ensure a smooth user experience.

3.2 Use Cases

This section presents the use cases for the system, categorized by four key actors: the Ticketchain Owner in the Section 3.2.1 where we detail the use cases for the Ticketchain owner, focusing on system settings management and the oversight of event organizers; the organizer in the Section 3.2.2 that covers the use cases for organizers, including event creation, management, and validator selection; the Validator in the Section 3.2.3 where we describe the validator's use case, which involves ticket validation to ensure that only users with valid tickets can enter events; and the Common User in the Section 3.2.4 where we outline the use cases for common users, such as purchasing, gifting, refunding, and reselling tickets. Each actor shares a common use case: authentication, which is responsible for verifying user identities within the system.

3.2.1 Ticketchain Owner

As illustrated in Figure 3.1, the Ticketchain owner has the specific use case of managing event organizers.

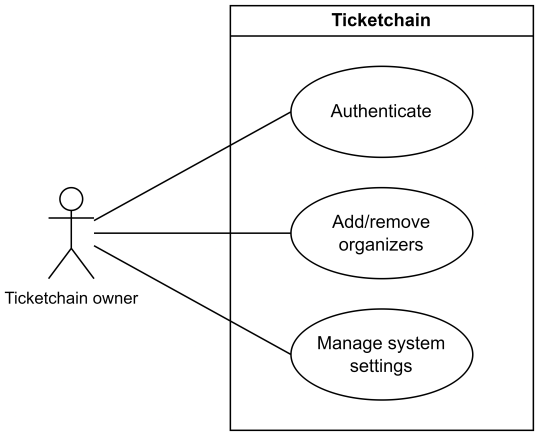

Figure 3.1: Ticketchain Owner Use Cases

This control is essential to ensure that only authorized individuals have access to the system. Additionally, the Ticketchain owner can manage system settings, which are crucial for tailoring the system's behavior to meet the needs of organizers.

3.2.2 Organizer

As shown in Figure 3.2, the organizer has several key use cases, including event creation and management.

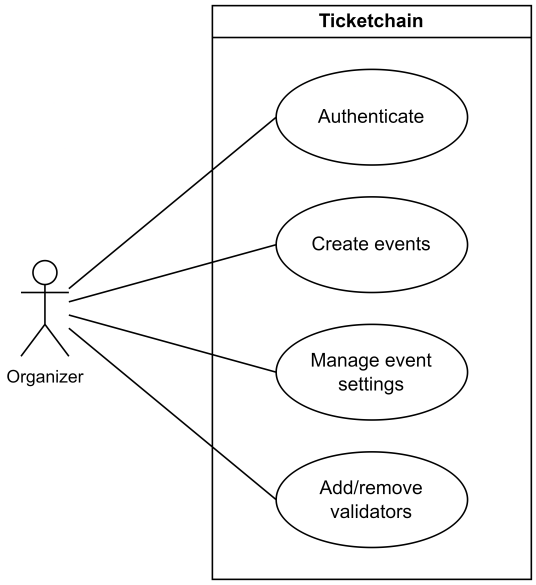

Figure 3.2: Organizer Use Cases

The organizer also oversees the selection of validators for each event, ensuring that only authorized personnel can validate tickets. Furthermore, the organizer can manage event settings, such as updating information or canceling events when necessary.

3.2.3 Validator

For validators, as illustrated in Figure 3.3, the primary use case is to validate users' tickets, allowing entry to the event.

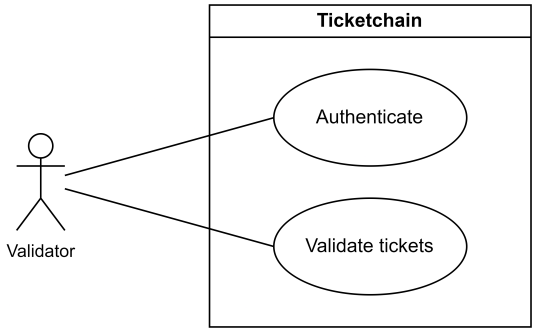

Figure 3.3: Validator Use Cases

This step is critical for maintaining security and ensuring that only users with valid tickets can participate.

3.2.4 Common User

Figure 3.4 presents the use cases for common users.

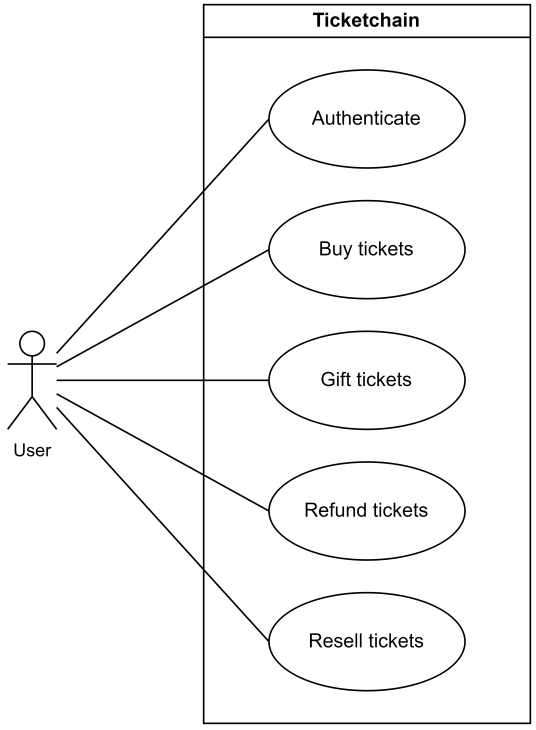

Figure 3.4: Common User Use Cases

Users can purchase tickets, gift them to others, request refunds (depending on event policies), and resell tickets (with the condition that they cannot sell at a price higher than the original). These use cases provide users with the flexibility to manage their tickets according to their preferences while adhering to system regulations.

3.3 Architecture

As this work is a blockchain-based project, the goal is to minimize the use of traditional Web2 technologies, such as server-based backends and related services. Instead, we aim to rely primarily on Web3 technologies to fully understand the limitations and possibilities of adopting a blockchain-only ecosystem. In the future, a hybrid approach combining both Web2 and Web3 could offer significant advantages. The system architecture is depicted in Figure 3.5, illustrating how the various components of the project interact, in line with the requirements discussed in previous sections.

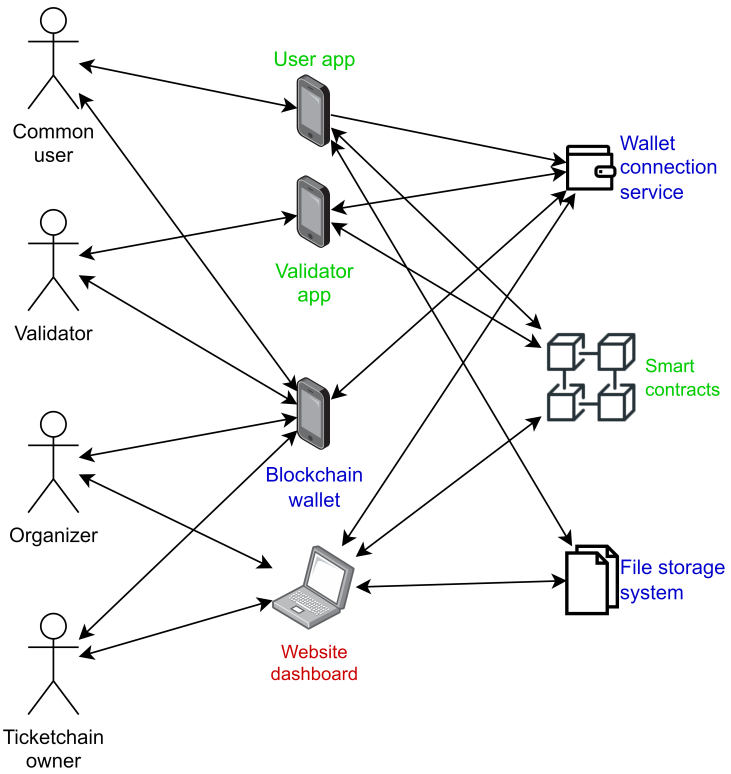

Figure 3.5: System Architecture

The architecture includes two main mobile applications—one for regular users and the other for validators—as well as a website dashboard for the Ticketchain owner and

event organizers to manage system settings and events.

The components highlighted in blue represent external services, while green components denote those we are developing in the context of this project. The red component, the website dashboard, has not yet been implemented due to time constraints, so for now, interactions will occur directly with the smart contract. Additionally, to streamline development, the two mobile applications have been consolidated into a single app, allowing us to focus on the core functionality.

As described in previous sections, all users must authenticate before accessing the system. Instead of traditional email and password logins, authentication will require wallet software to interact with the blockchain. Thus, a wallet connection service is needed to link user wallets to the system.

Once authenticated, the system will interact with the smart contract to display event-related information and update the status of events or tickets. The smart contracts will be deployed on an EVM-compatible blockchain.

Regarding ticket images, a file storage system will be required. Although image uploads will eventually be handled via the dashboard, this feature is not yet implemented, necessitating manual uploads for now.

4

Implementation

In this chapter, we discuss the implementation of the system according to the proposed architecture. The user application is described in Section 4.1, the validator application in Section 4.2, and the smart contracts in Section 4.3, along with the tools required for deploying the system. Although the user and validator applications are discussed in separate sections, we have implemented a single application that serves both roles in this project. However, in a real-world scenario, these would typically be developed as separate applications.

4.1 User App

The user application serves as the primary interface for users to interact with the system. Through this app, users can authenticate, browse events, and manage their tickets. The app features a homepage for event browsing, an event-specific page where users can view details and purchase tickets, and a dedicated page to manage owned tickets. This section is structured as follows: authentication is covered in Section 4.1.1, event browsing in Section 4.1.2, and ticket management in Section 4.1.3.

The app is developed using the Flutter framework, a popular cross-platform solution by Google that allows us to build applications for both Android and iOS from a single codebase. Flutter is known for its ease of use and high performance, making it well-suited for this project.

4.1.1 Authentication

The Figure 4.1 shows the starting point of the app that separates the authentication of the common users from the validators.

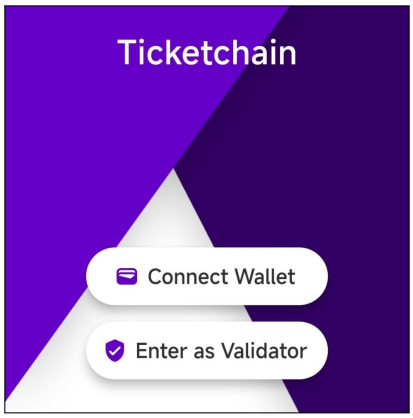

Figure 4.1: Authentication page

To authenticate users, we use the Wallet Connect service [17], which supports a variety of wallets for user interaction. This service establishes a secure connection between the app and a wallet, enabling the wallet to receive prompts and sign transactions.

Figure 4.2 illustrates the Wallet Connect interface that appears when users attempt to authenticate. This prompt displays all the supported wallets, allowing the user to choose their preferred option.

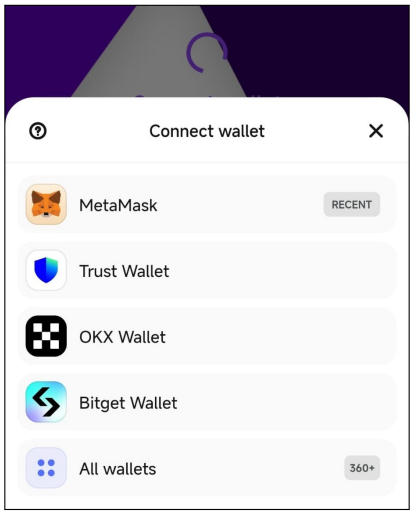

Figure 4.2: Wallet Connect prompt

Once a wallet is selected, the corresponding wallet app will automatically open, and the user must approve the connection, as shown in Figure 4.3. For this implementation,

we use MetaMask [13], a popular blockchain wallet. Users simply need to create a MetaMask wallet to begin interacting with the app.

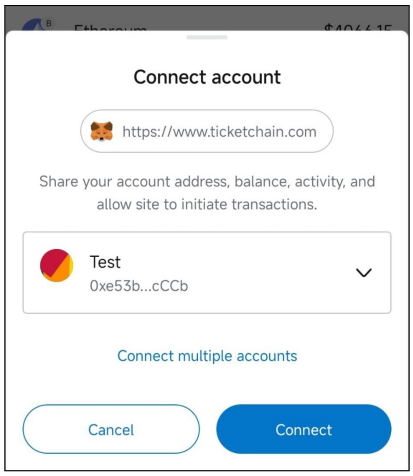

Figure 4.3: MetaMask connection approval

The authentication is a one-time process, so the user doesn't have to do this every time he wants to interact with the app. Basically when a connection is established, the next time the app tries to reconnect to the wallet, it will skip the prompt.

4.1.2 Events

After the common user authenticates, he will be redirected to the home page of the app, shown in Figure 4.4, where he can see the events that are available.

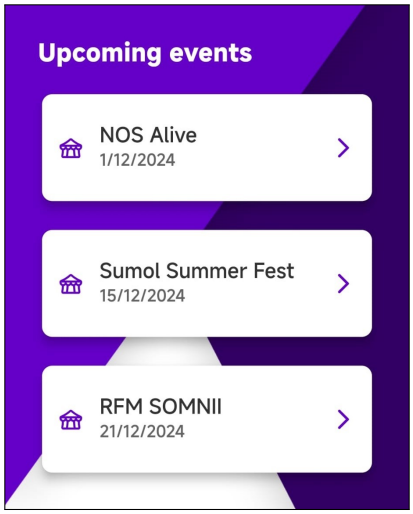

Figure 4.4: Home page

When clicking in one of the events, the user will be redirected to the event page. Figure 4.5 shows the event page, where the user can see the details of the event and the packages available.

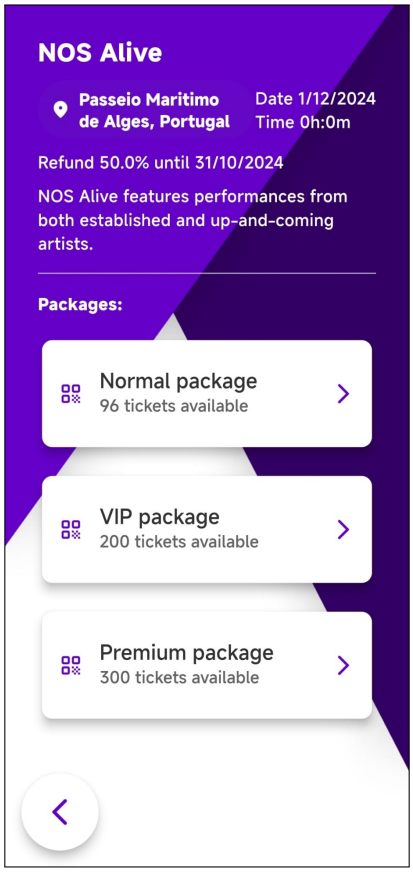

Figure 4.5: Event page

The user sees the name, description, location, date, packages and even the refund information. When the user taps on the package he wants to buy, the prompt shown in the Figure 4.6 appears, where the user can choose the amount of tickets he wants to buy. We went with this approach of only choosing the amount of tickets to buy, but in the future a feature like seat selection could be implemented.

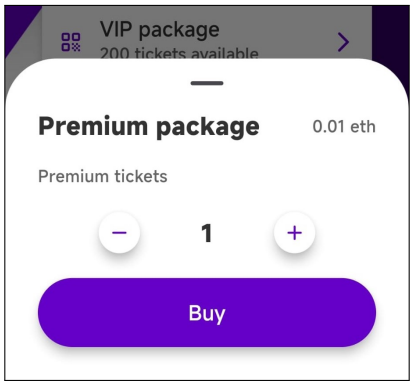

Figure 4.6: Buy tickets prompt

The user can then confirm the purchase, and the wallet will prompt the user to sign the transaction, as shown in the Figure 4.7. It displays the amount of money the user has to pay, to which address he's interacting with, and the total cost to execute the transaction.

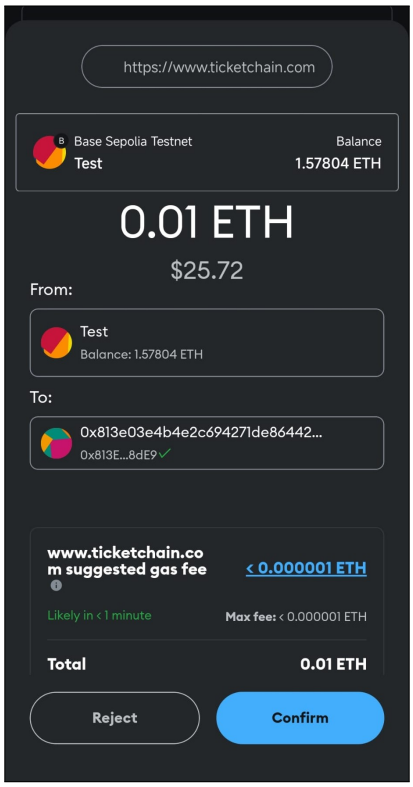

Figure 4.7: MetaMask transaction prompt

4.1.3 Tickets

After confirmation, the user is redirected back to the app where, on the second tab with the profile icon, the profile page, he will be able to see the events which he owns any

tickets, as shown in the Figure 4.8.

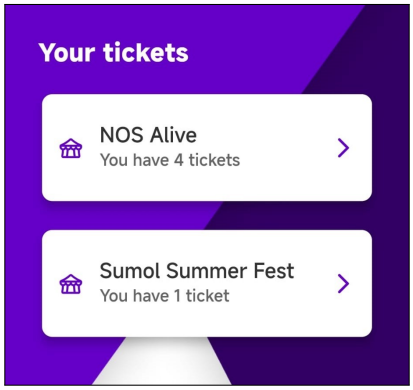

Figure 4.8: Profile page

Going into one of them, like the Figure 4.9 shows, the user can see the tickets he owns for that event. It's a similar page as the normal event one, but with the tickets he owns instead of the packages available. We see 4 tickets here and the first one has a mark on it, which means the ticket has already been validated.

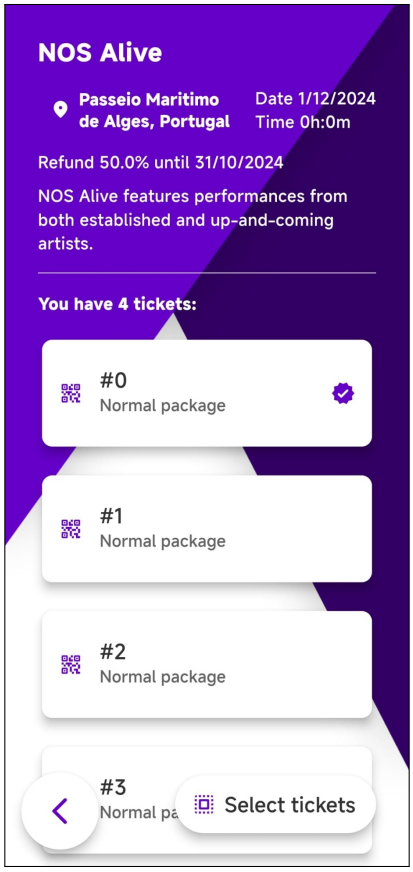

Figure 4.9: Tickets page

Clicking on one of the tickets, it shows us the basic ticket information, along with its image, as shown in the Figure 4.10.

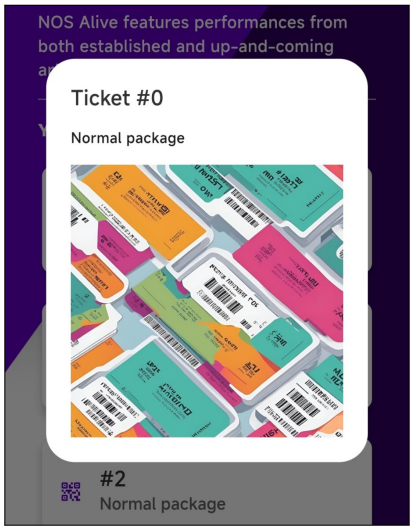

Figure 4.10: Ticket information

For operating the tickets, the user can simply click on the select tickets button which will allow him to choose the tickets which he wishes to operate, as shown in the Figure 4.11. In this case the user sees only 3 tickets (while the Figure 4.9 shows 4) because since the first is already validated, it's not possible to operate on it anymore. We see the options to gift, refund and validate the tickets he selected.

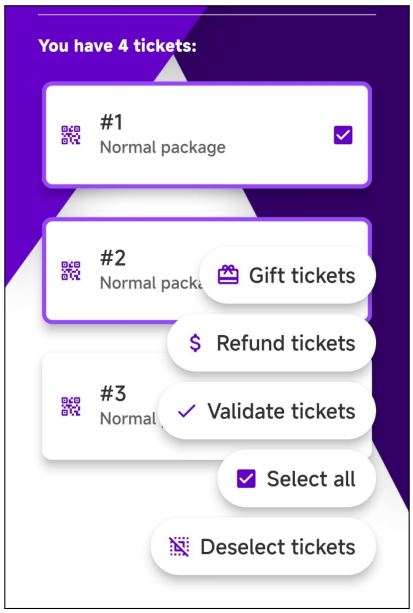

Figure 4.11: Ticket operations

The gift option will ask the user for the address to which he wants to gift the tickets. After that, it will trigger the wallet to sign the transaction, and the tickets will be transferred to the address.

The refund option will ask the user to confirm the refund, and trigger the wallet to confirm the transaction in which the tickets will be burned, making them available again for other users to buy, and returning the correct amount of money to the user.

The validate option will start the validation process, which will be explained in the Section 4.2.2.

4.2 Validator App

For the validator app, we will have a simple interface for the validators to execute the validation process. Like mentioned, this is currently done in the same app as the common users, but in a real scenario we would have a separate app for the validators.

Ideally we will authenticate the same way as common users, but for the sake of making it easy to demonstrate and test, the app will behave as a wallet. Essentially this means that the app holds the private keys and signs the transactions.

This section is divided in two: the validator address in Section 4.13 and the ticket validation in Section 4.2.2.

4.2.1 Validator Address

Since this is the case, we need a way to see the address of the validator, so that the organizer can add him to the event and give him the necessary funds to operate. The reason for the necessary funds is mentioned in the Section 4.3.3.5.

The Figure 4.12 shows the starting point of the app after authentication, where the validator has the option to check the address and to validate the tickets.

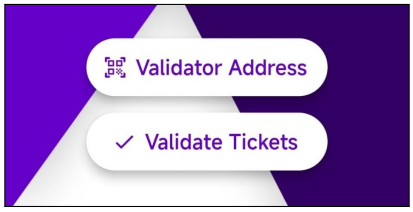

Figure 4.12: Validator page

When the validator taps the address he will see his address, as well as its QR code, as shown in the Figure 4.13.

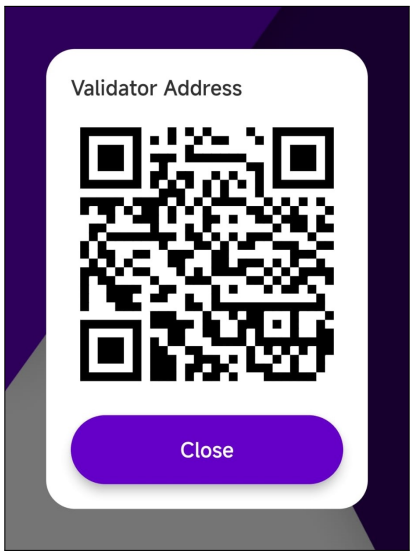

Figure 4.13: Validator address

When the validator select the validate tickets option, he will start the process of validating the tickets, which will be explained in the next Section 4.2.2.

4.2.2 Validate Tickets

For the ticket validation process, we must take into consideration a lot of aspects, because it's not just checking if the user address has a ticket associated to him. This is because, since the data is on the blockchain and it's public, anyone can see the addresses where each ticket belongs to, and pretend he's the owner of the ticket. For this to be secure, we need to guarantee the user is actually the owner of the address, and here is where the cryptographic message signature comes in, the same process that happens when executing a blockchain transaction.

We came up with the process shown in the Figure 4.14, that shows the steps between the actors and the system to validate the tickets.

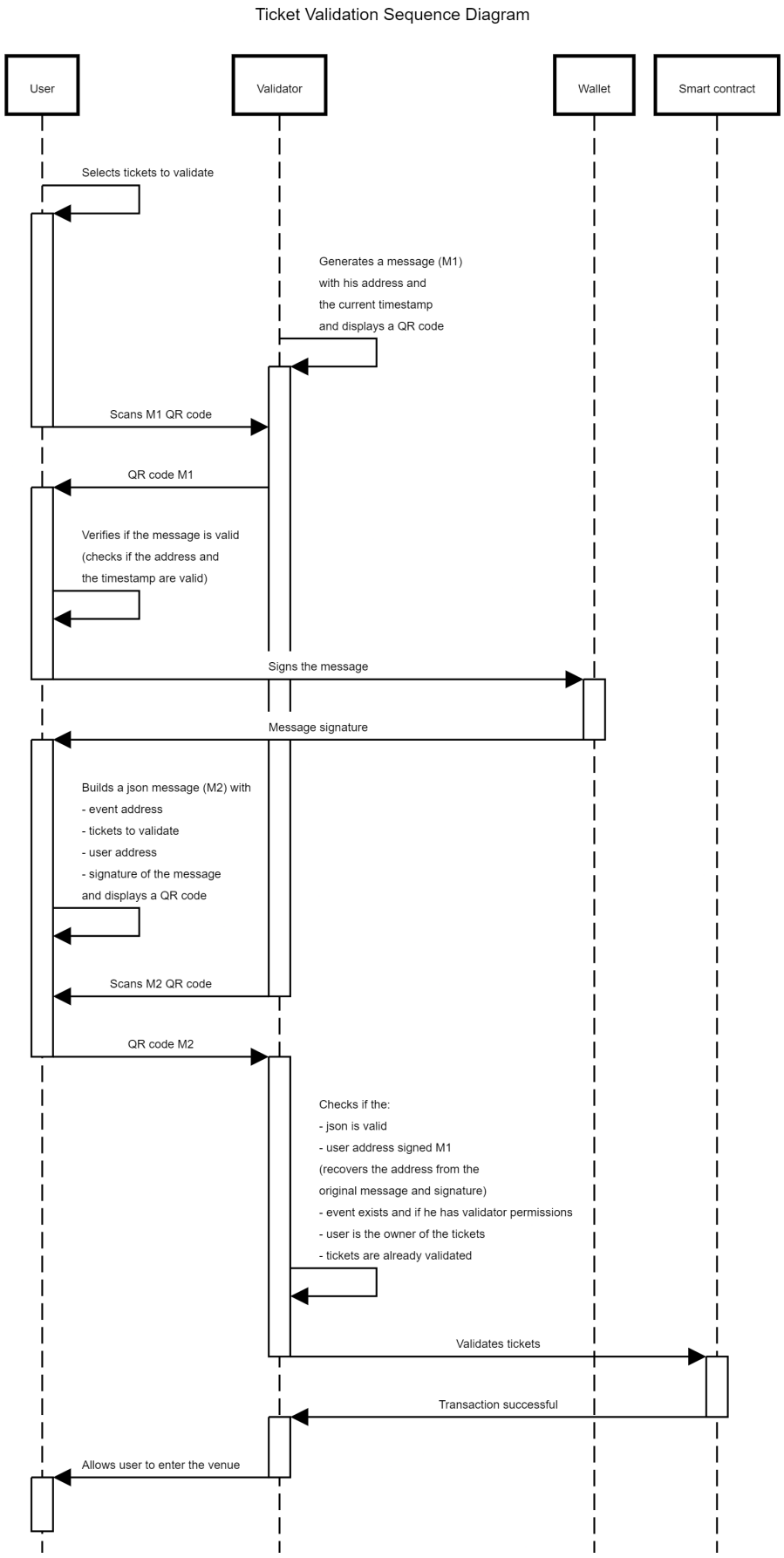

Figure 4.14: Ticket validation

Here we see that the user reads the QR with a generated message from the validator, then signs it with his wallet, and generates a JSON with the signature and useful information like the tickets to validate, the event, and the user address. After the validator reads the JSON in the QR code, he checks if the parameters match the ones on the blockchain, gets the address using a cryptographic method to recover it through the original message and it's signature, and checks if the user is the owner of the tickets. If everything matches, the validator will trigger a transaction to mark the tickets as validated, to avoid people sharing the accounts and using the same ticket multiple times.

Both parties need to know the original message, so it matches. We could just use a default message for everyone, so the users would just need to give the signature to the validator, but this could become a security issue in case the signature gets leaked, because anyone who has it could pretend to be someone else. The idea here is to have a unique message for each user at the time of the event, so they are forced to sign it.

All this is reduced to a single transaction on the blockchain because, depending on the network, the finality of a transaction (a term used in the blockchain ecosystem that defines the time it takes for a transaction to be fully registered) can take a while. This was planned to be done in a single transaction to avoid congestion at the entrance of the venue, so the users can enter the event with the least amount of delay. This is an important aspect to consider when designing the system to fullfil the system being fast, so like mentioned in the Section 3.1.2, the network choice is crucial.

4.3 Smart Contracts

We split this section into the main three contracts of the system: the Ticketchain contract in Section 4.3.1, the ERC721 contract in Section 4.3.2, and the Event contract in Section 4.3.3.

The deployment of the smart contracts will be done using Foundry [9], a toolkit for the development and deployment of the smart contracts, which makes it easy to deploy to any network, and also to test the contracts locally.

So smart contracts are similar to C++ mainly because it lies on a class-like structure with variables to store data and methods, where the main difference is that a class is called a contract. You can also extend others to integrate their functionalities. That's essentially what's gonna happen with each event, extending the ERC721 standard, making it a collection of NFTs, where each NFT is a ticket.

Since this is the behavior we want (each event being a NFT collection), we will have to deploy (instantiate, in C++) a new contract for each event, so we will have a main contract (called Ticketchain, since it's the main contract) to keep track of them. This structure can be visualized in the Figure 4.15, which shows the simplified UML of the system.

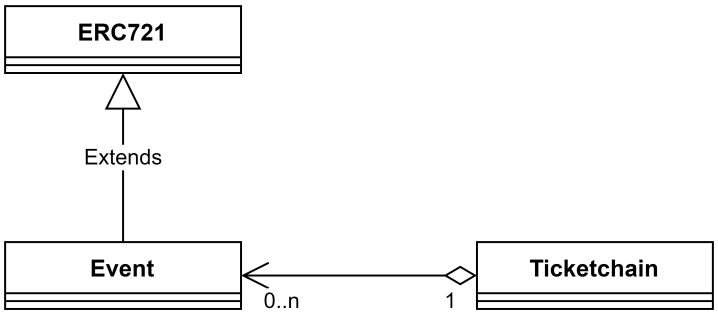

Figure 4.15: System UML (simplified)

We see that the Event contract extends the ERC721 standard, and the Ticketchain contract will deploy the events and keep track of them. The contracts will be explained in detail over the next sections.

4.3.1 Ticketchain Contract

The Ticketchain contract is the main contract of the system, from where the events are registered and stored. It will have the necessary methods to register new events and to add organizers. The Figure 4.16 shows the most relevant methods of the contract.

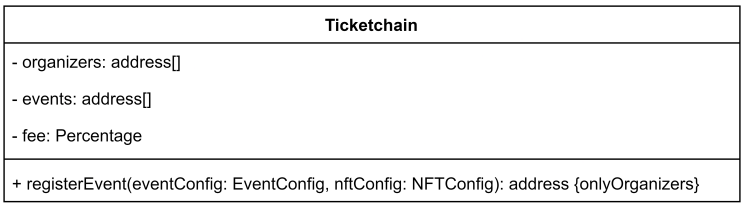

Figure 4.16: Ticketchain UML (relevant methods)

We have method to add organizers, so only the ones that are allowed can create events. This is a good feature to have because it prevents unauthorized people to create events

and possibly scam the users. The organizer is the one that will be able to add the necessary information about the event, like the name, description, location, dates, and packages, and also to manage the event, like adding packages, changing the dates, and refunding the tickets.

When an event is created, we transfer the ownership of the event to the organizer, so he can manage it. This is a common practice in the blockchain ecosystem, where the ownership of the contract is transferred to the user that created it, so he he's fully responsible for it. This ownership is restricted by using modifiers that check if the user is the owner of the event, so only the organizer can execute the necessary operations.

We also have the events address list, which is a way to track all the events that are deployed, so we can easily get them all and show them in the app. The fee variable is there to store the system fee, which will be mentioned in the Section 4.3.3.5.

4.3.2 ERC721 Contract

The ERC721 standard will be extended and we will be adding the necessary methods to interact with the tickets, like buying, selling, and validating them. The reason to extend this standard and not implement the logic manually is mainly because it makes it compatible with the most common marketplaces for NFTs, which allows for users to do what they desire with them after the event. It also has the necessary methods to manage the tickets, like transferring them between users, and the necessary operations to track these operations.

The standard was obtained through the OpenZeppelin [14] library, which is a collection of secure and community-vetted smart contracts that are used by many projects in the Ethereum ecosystem. This library is a great resource for developers to build secure and reliable smart contracts.

Analyzing its source code [7], and looking into the most important variables and methods of the standard shown in the Figure 4.17, we can understand that the NFTs are simply a mapping of the token ID to the owner address, so when you execute a transaction to get a token (this process is called minting), the token ID is then associated to your address. Then for each token it's possible set a URI, which is a link to the NFTs metadata, usually being a JSON file with the necessary information about the token, like the name, description, and image.

This link could point to anything, for example a google drive file, but the common thing is to store the metadata on the InterPlanetary File System (IPFS) [12], which is a

decentralized storage system, so the metadata is not stored on the blockchain itself (on-chain), which would be very expensive, but rather on a decentralized storage system (offchain), which is much cheaper.

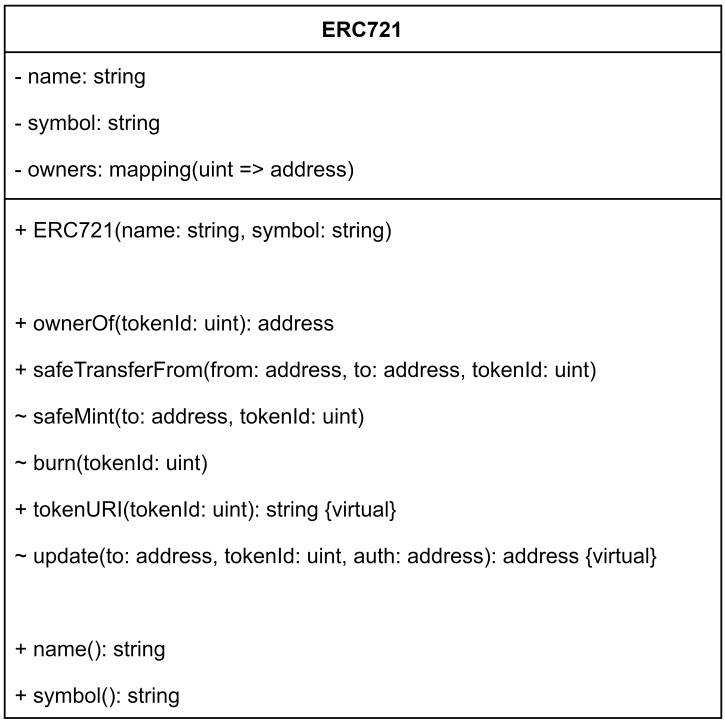

Figure 4.17: ERC721 UML (relevant methods)

The function *tokenURI* is the one that is called by default in the marketplaces to get the NFT's metadata, being one of the main reasons to extend the ERC721 standard, because it enforces the implementation of this method. In the Figure 4.17 we see that it has the *virtual* keyword, meaning this can be overridden by the contracts that extend it, to manipulate the way to store the metadata. We'll be mentioning this again in the Section 4.21, about how the packages logic is implemented.

4.3.3 Event Contract

So the event contract will be deployed and we need a certain control over the tickets. One of the aspects we need to account for is that when deploying an event, and since it will extend the ERC721, any public functions on that standard will be possible to execute. This is a problem because we don't want the users to mint tickets whenever they want or transfer them between themselves from outside the system, so we need

to restrict these operations. As we saw already on the Figure 4.17, only the *safeTransferFrom* method is public, so users could transfer NFTs between each other. We want that to be possible, just not from outside the system, since that can lead users to exploit the system and scalping the tickets easily. The minting, however, won't be an issue because it's an internal method, so we will access it from the buy method in the event and restrict it there.

4.3.3.1 Event Lifecycle

The Figure 4.18 shows the lifecycle of the event, and what restrictions are in place for the ticket operations. The dates above the line indicate the states of the event, and below a small description of the operations that are allowed when each state is reached.

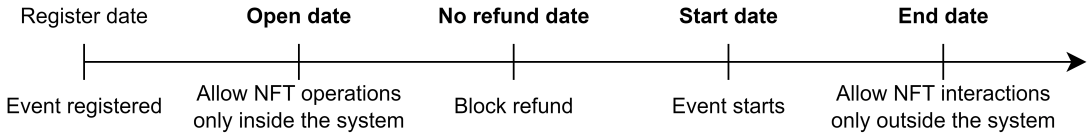

Figure 4.18: Event lifecycle

We will have 4 main states for the event after it has been registered (the ones in bold), being the *Open*, *No refund*, *Start*, and *End* dates. Once the event is registered, it will show up in the app for user to see, and the organizers can set a later open date to allow ticket minting (buying).

Open Date: Once it hits the *Open* date, we will allow the users to buy tickets, which will mint the NFTs by executing the *safeMint* method of the ERC721 contract. We will allow users to operate over the NFTs, but only within the system. If they try to call the *safeTransferFrom* directly from the ERC721 standard, it will revert, because we detect it's not being called through the system.

No Refund Date: After the *No refund* date, we will prevent the users to call the refund method, which essentially *burns* the NFTs, removing them from the user and making them available again. This is a nice operation to add because it allows the users to get their money back if they can't attend the event. The organizer decides the percentage of the refund and the deadline, which is there to prevent users to buy a big amount of tickets and then refund them last minute, which would be a way to exploit the system

(in case of a 100% refund, they wouldn't risk anything). The other good thing for the organizer is when the event is expected to be sold out. Since the users will get some money back, they will have a reason to refund their tickets if they cannot attend the event anymore, making them available again for other users to buy at the original price, making the organizer a higher profit. After this deadline, the only that'll be allowed is for users to resell their tickets in the system's marketplace, which them to sell at a higher price than the refund (but never higher than the original, of course).

Start Date: The *Start* date is there to tell the users when the event starts, so basically when the gates will open. That's the date that appears in the app, so the users know when to show up.

End Date: The *End* date tells when the event is over, unlocking all the ticket operations to outside the system. So users can simply keep the tickets as a souvenir or sell them in any marketplace, without any restrictions on the tickets, including the removal of the price cap.

With this behavior in mind, we came up with the Event UML, as shown in the Figure 4.19, where we added the necessary methods for the organizer/admins to manage the event and the users to handle the tickets, which then trigger the corresponding methods of the ERC721 contract. We added a possibility to have admins so the organizer can distribute the workload of executing the necessary operations to people he trusts.

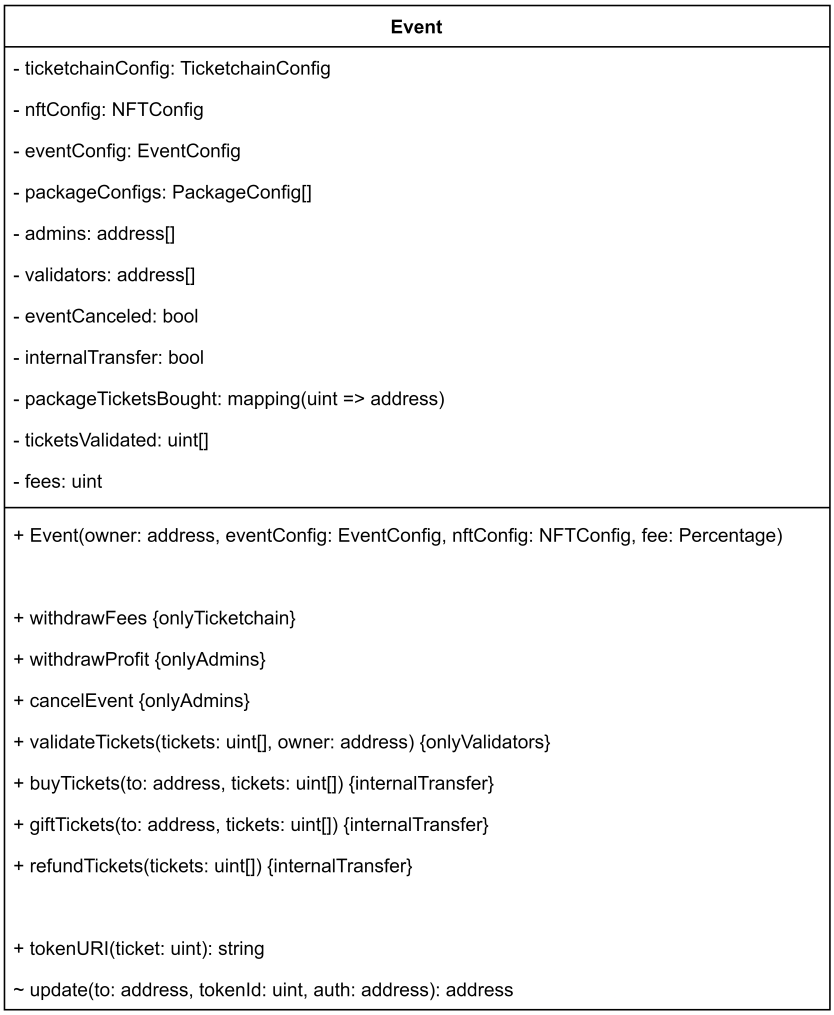

Figure 4.19: Event UML (relevant methods)

As we can see, the *update* method has been overridden from the ERC721 standard to restrict the interactions with the NFTs according the defined behavior. This method is the one that gets called anytime there's an operation on any NFT, so we can implement here the necessary logic to restrict the operations, and we can visualize that in the ~~following~~ flowchart~~,~~ shown in ~~the~~ Figure 4.20.

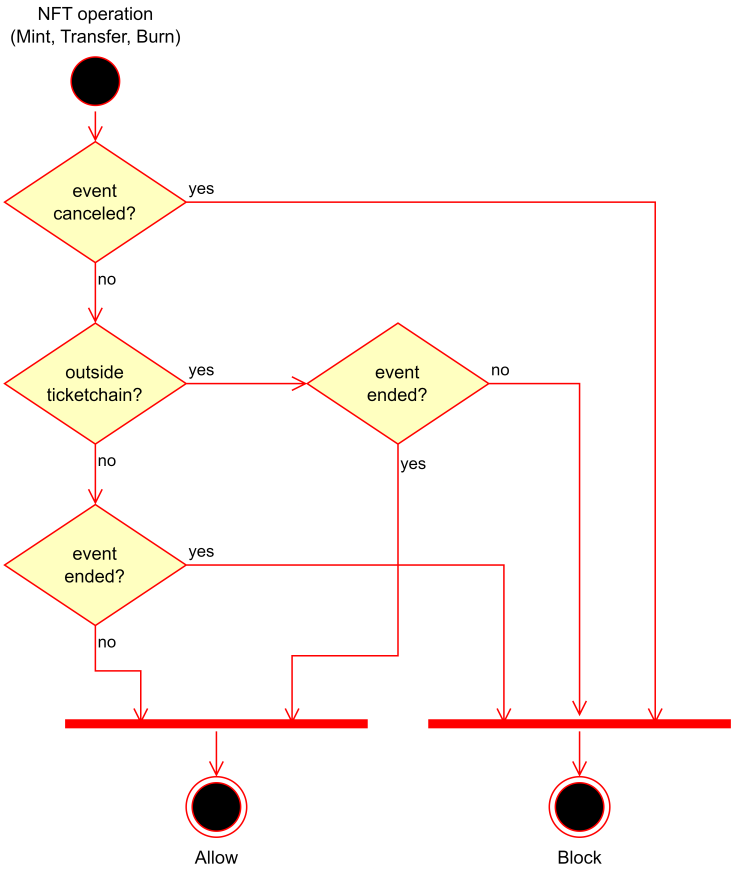

Figure 4.20: NFT flowchart

This logic is possible to implement because of modifiers, mentioned before. What we essentially do on that modifier, is set a variable to true (meaning the NFT operation is coming from within the system), execute the function it's assigned to, and then we set the variable back to false. Then we apply that modifier on the necessary functions, which we can see them with the operator internalTransfer in the event UML, shown in the Figure 4.19.

4.3.3.2 Structs

From the UML figures, we can see that we have a few custom structs to organize better the data. These structs are the *Percentage*, *EventConfig*, *NFTConfig*, *PackageConfig* and *TicketchainConfig* structs.

Percentage Struct: The *Percentage* struct is necessary because in Solidity there are no floating point numbers, so we need a way to make calculations with percentages. What

this struct does is it stores the value of the percentage and the amount of decimals it has, so if we want to calculate 55.50% of a number, we would have 555 as the value with 1 decimal, or 5550 with 2 decimals.

The struct is as follows:

Listing 4.1: Percentage struct

```
struct Percentage {
        uint256 value;
        uint256 decimals;
}
```

so to obtain a percentage of some x number, we do $y = \frac{x \times \text{Percentage.value}}{100e^{\text{Percentage.decimals}}}$.

When working with ether units, it can be common to have values like 0.00005 ether, but it's rather rare to have values in wei like 1000 wei, so applying this formula won't lose much precision (note that 1 ether is 10^{18} wei). Equating this to real world value, 0.001 ether is somewhat close to 2.5$ (at the moment), so 1000 wei would be way lower than even a cent.

TicketchainConfig Struct: The *TicketchainConfig* struct is simply to keep it stored the system address and the system fee percentage, so we can easily access this information when applying the fees and withdrawing them, and is as follows:

Listing 4.2: TicketchainConfig struct

```
struct TicketchainConfig {
    address ticketchainAddress;
    Percentage feePercentage;
}
```

NFTConfig Struct: The *NFTConfig* struct is just to store the NFTs basic information, like the name, symbol, and base URI, to ease the input of the NFTs information when registering the event:

Listing 4.3: NFTConfig struct

```
struct NFTConfig {
        string name;
        string symbol;
        string baseURI;
}
```

The *name* is the name of the NFT collection and the symbol is the abbreviation of it, like the name being *Ticketchain* and the symbol being *TCK*, for example. The base URI is the link to the metadata of the NFTs, which will be used to get the information about the tickets.

EventConfig Struct: The *EventConfig* struct is to store the event's entire configuration, like the name, description, location, dates, and refund, like this:

Listing 4.4: EventConfig struct

```
struct EventConfig {
        string name;
        string description;
        string location;
        uint256 openDate;
        uint256 noRefundDate;
        uint256 startDate;
        uint256 endDate;
        Percentage refundPercentage;
}
```

PackageConfig Struct: Lastly, the *PackageConfig* struct is there to store each package information, to keep track of the ones that are available for the event:

Listing 4.5: PackageConfig struct

```
struct PackageConfig {
        string name;
        string description;
        uint256 price;
        uint256 supply;
        bool individualNfts;
}
```

This structure will be better discussed in the next Section 4.3.3.3.

4.3.3.3 Ticket Packages

It's common to see events with different types of tickets, like VIP, standard, or even 3-day passes, each with its own price and benefits. We want to implement this feature in the system, so we can have a better control over the tickets and the users can choose the one that fits them better.

For that, we will allow the organizer to add packages, indicating the supply of each one, and as we saw already, the NFTs are a mapping of the ID to the owner, so we can organize the packages as a list, where the supply and order of them assigns the ID of each NFT to the package, like the Figure 4.21 illustrates.

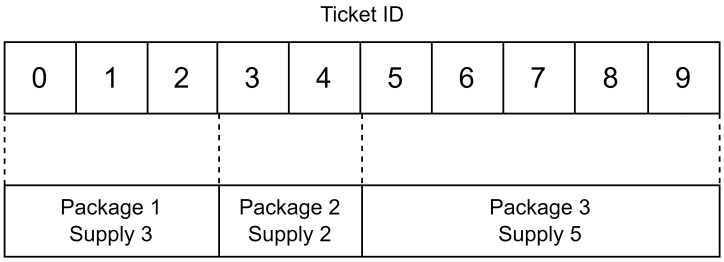

Figure 4.21: Package logic

This way, whenever we need to get a ticket for a certain package, we can go through the packages and see which one the ID is in. One only limitation with this is if the event is already open (users can buy tickets), the only thing we can allow the organizer to do is to add packages, neither remove or change their order, because that would change the package associated to the tickets, which would be a problem for the users that already bought them.

Now we just have to make sure the information obtained with the *tokenURI* method corresponds to the ticket, according to its package. For this we will have a different metadata file for each package, with the necessary information about the tickets. The *individualNfts* boolean in the *PackageConfig* struct is there to indicate if the organizer wants each ticket on the package to have its own metadata, or if they can share it.

According to this, the *tokenURI* will return an URI like *baseURI/packageId/ticketId* for a package with individual NFTs, and *baseURI/packageId* for a package with shared metadata. Like this, when we store the metadata on the IPFS, we store the metadata for each ticket inside a folder of the packages with individual NFTs, and only a metadata file for each package without individual NFTs.

4.3.3.4 Metadata Storage

To store the NFTs metadata files, we'll be using the IPFS for storing the NFTs metadata. The IPFS is a decentralized storage system where the data is stored in a distributed network of nodes (decentralized), making it very secure and reliable. This is a great

solution for storing the metadata of the NFTs because it's very cheap and easy to use, and it's a common practice in the blockchain ecosystem.

Other options would be to store the data on some kind of server, but that would be more expensive to maintain, and since we are dealing with NFTs, it's good practice to store the data in a decentralized manner, to avoid any kind of alteration on its contents, if the tickets possibly become valuable collectibles.

This kind of issue was something that has happened before, where people bought NFTs with the idea of them being somewhat valuable, but then the owner changed the contents of the metadata, executing what was called of a *rug pull*, which is a scam that made the NFTs worthless, keeping the money for himself.

To store the data on the IPFS, we will be using the Pinata [15] service, which does the heavy lifting for interacting with the storage itself. To accomplish this, we just need to arrange the files according to the ticket packages, like mentioned before in the Section 4.3.3.3, and then upload them to the IPFS, getting the link to the metadata, which we'll then store on the contract as the base URI. The files would be stored like the Figure 4.22 shows, being the packages 1 and 3 with shared metadata, and the package 2 with individual NFTs.

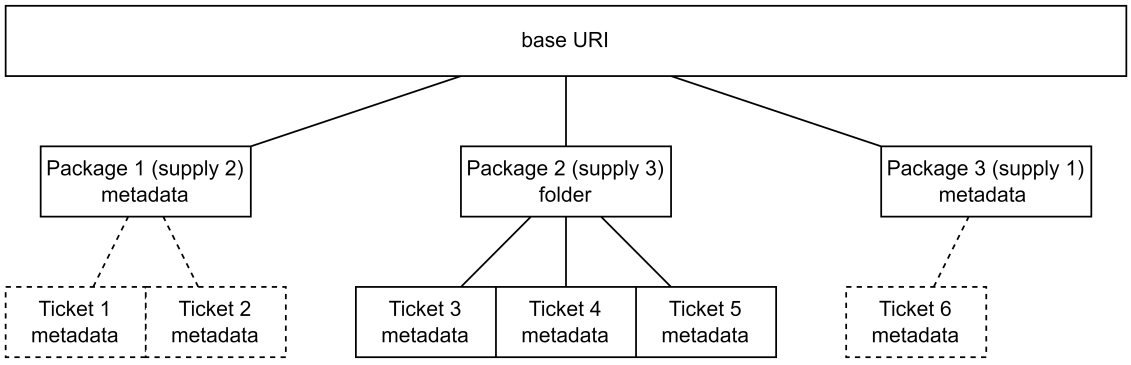

Figure 4.22: Metadata storage

4.3.3.5 System Fees

One of the most important aspects of a system like this is the business model we have to take into account. Since this is a service we want to deploy for event organizers, we need to make this sustainable and profitable. This kind of service aims to do some heavy lifting, with its own features, so we could set a fee lower than the usual on the traditional marketplaces and ticket selling platforms, since the organizers need to pay for each service.

These low fees are possible because with the system being deployed on the blockchain, it stays there while the network is running, so the only extra cost are the network fees (also called gas fees) when interacting with the event. For the users, each interaction is paid by them, so when a user buys a ticket, the only thing to take into account are the network fees, which depending on the network, can be super low.

The other kind of fee the organizer needs to look out for is for the validators to validate the tickets, which is a necessary operation to avoid people from exploiting the system. These fees are paid by the validators, which the organizer essentially manages, so we need to take them into account when setting the system fee, to make it sustainable for the organizer to use our system, since the organizers will have to funds the validators' wallets.

We'll set a fee on the Ticketchain smart contract, where will be stored in the event when registering it, so that if we decide to change it, the previous events aren't affected. This is also because we want to abstract the user of any extra fee, so the price the organizer sets, is the price the user pays, and the system fee is taken from the ticket price. In case an event gets cancelled, or a user decides to get a refund, the ticket fee is returned to the user (proportional to the refund), making the system less profit, but guarantees the users of a fair process. With this, we need to restrict the system to only withdraw any profit after the event is over. Since this is rather an uncommon case, the less profit that the system makes, possibly compensates for the trust that the users and organizers will have on it.

5

Results

In this chapter, we present the results of the decentralized ticketing system, including interactions with smart contracts, the process of buying in Section 5.1, gifting in Section 5.2, refunding in Section 5.3, and validating tickets in Section 5.4.

As mentioned earlier, blockchain serves as a public ledger that records all transactions. We deployed the smart contracts on a testnet to avoid the use of real funds, allowing us to simulate mainnet operations without incurring risks or costs. This was achieved using Foundry, and we developed scripts to populate the system, which included deploying the smart contracts, adding an event organizer, creating events, and adding ticket packages to each event.

Every network has an explorer that visualizes all transactions. Figure 5.1 illustrates the transactions related to our system.

Transaction Hash	Method	Block	Age	From		To
0xda3d9dbd83...	Register Event	12785772	44 days ago	0xe53b00C0...d79d8cCCb	IN	0x87f4a5C1...0fA28F20d
0x063dd20306...	Register Event	12785771	44 days ago	0xe53b00C0...d79d8cCCb	IN	0x87f4a5C1...0fA28F20d
0xfa2b0777f77...	Register Event	12785771	44 days ago	0xe53b00C0...d79d8cCCb	IN	0x87f4a5C1...0fA28F20d
0x673b9c792f6...	Add Organizer	12785771	44 days ago	0xe53b00C0...d79d8cCCb	IN	0x87f4a5C1...0fA28F20d
0x948308498a...	0x60806040	12785771	44 days ago	0xe53b00C0...d79d8cCCb	IN	Create: Ticketchain

Figure 5.1: Ticketchain transactions

This explorer allows us to track when the contract was deployed, the addition of an

organizer, and the creation of three events. This data is subsequently loaded into the app, as shown earlier in Figure 4.4.

5.1 Buying Tickets

To purchase tickets, users follow the procedure outlined in Section 4.1.2. Figures 4.4 and 4.9 show two available events, although there are three in total. We will now demonstrate the process of buying tickets for the third event, which has one available package containing 100 tickets, as seen in Figure 5.2.

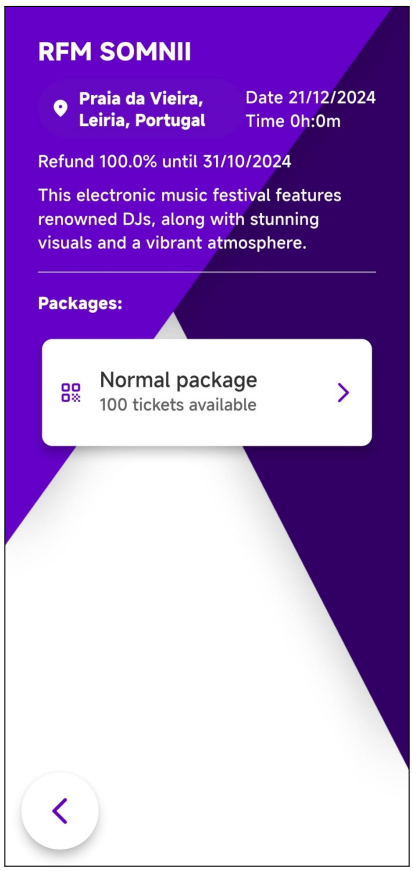

Figure 5.2: Third event page

After selecting the package, the user is prompted to choose the number of tickets to purchase. In this case, we will buy 3 tickets, as illustrated in Figure 5.3.

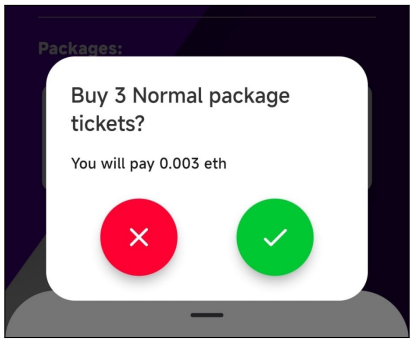

Figure 5.3: Prompt to buy 3 tickets

Upon confirmation, the user's wallet opens, and the transaction is displayed for approval (Figure 5.4).

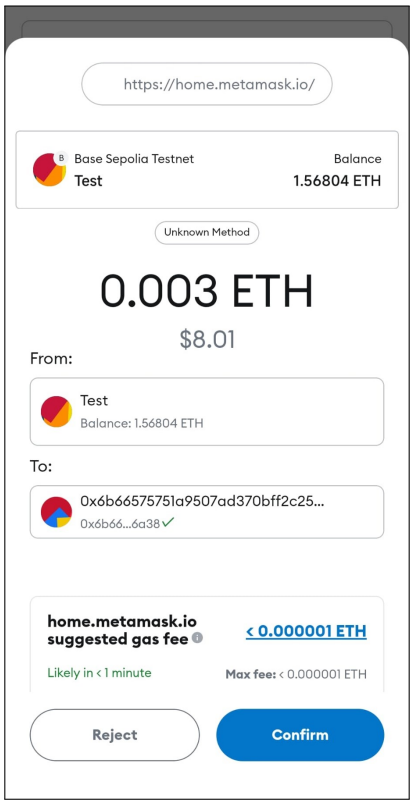

Figure 5.4: Buy 3 tickets transaction prompt

The user sends 0.003 ETH to complete the purchase. After successful payment, the profile page updates to reflect the addition of the third event and 3 purchased tickets (Figure 5.5).

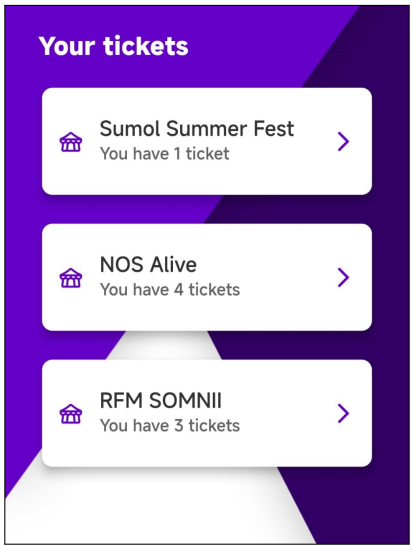

Figure 5.5: Profile page showing tickets for the third event

Returning to the event page, we see that the number of available tickets has decreased to 97, as shown in Figure 5.6.

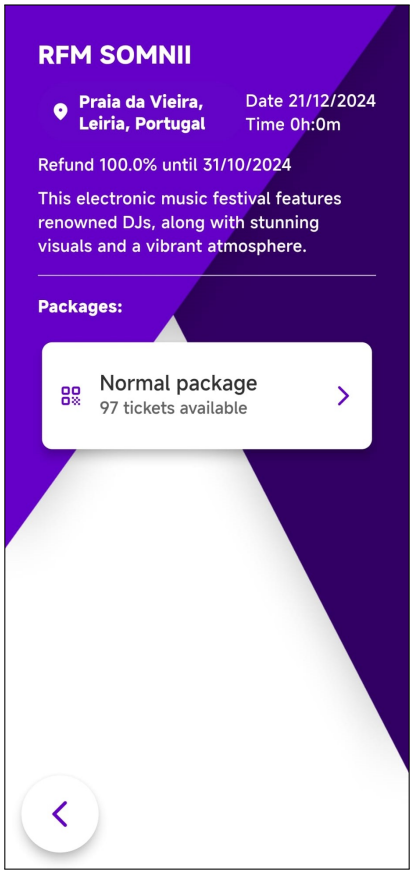

Figure 5.6: Third event page with 97 tickets remaining

The blockchain explorer shows the associated transaction, as displayed in Figure 5.7.

Transaction Hash	Method	Block	Age	From		To
0x3a539e915d...	Buy Tickets	15883572	1 hr ago	0xe53b00C0...d79d8cCCb	OUT	0x6b665757...258386a38
0x446f05ac24d...	Buy Tickets	15635321	5 days ago	0xe53b00C0...d79d8cCCb	OUT	0x112EDA21...8Ae7041c6
0x5ab609ca01...	Buy Tickets	15154798	16 days ago	0xe53b00C0...d79d8cCCb	OUT	0x813E03e4...2f0458dE9

Figure 5.7: User's transaction history

By selecting the transaction hash, we can view the transaction details (Figure 5.8).

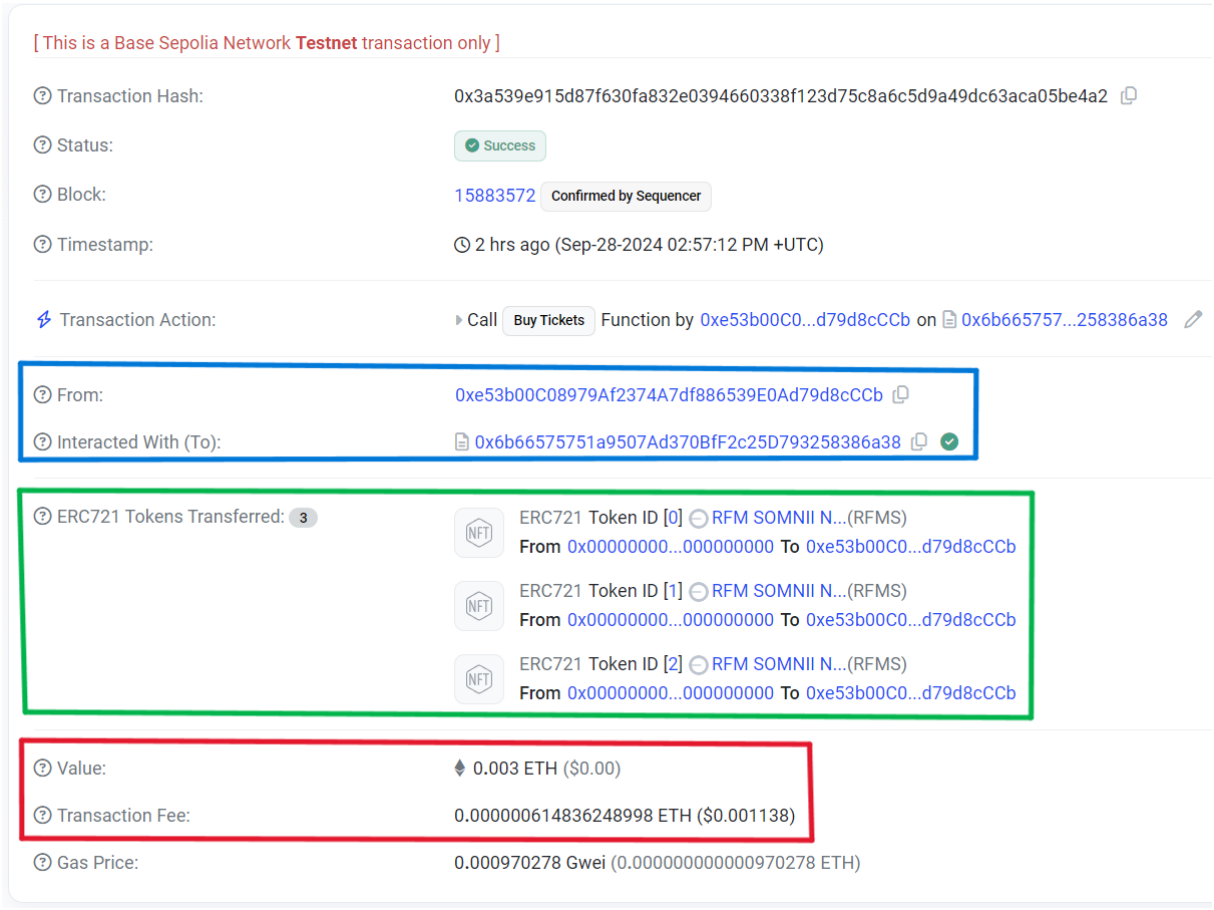

Figure 5.8: Details of the buy tickets transaction

The key information includes the following:

Blue box: The addresses involved in the transaction, where the *From* field shows the user's address and the *To* field shows the event contract address.

Green box: The NFTs (tickets) transferred, showing that IDs 0, 1, and 2 were minted and assigned to the user.

Red box: The amount of ETH sent (0.003 ETH) and the network fee paid for the transaction.

5.2 Gifting Tickets

To gift tickets, the user selects the tickets to transfer, as shown in Figure 5.9.

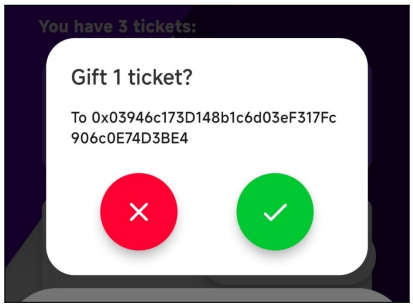

Figure 5.9: Prompt to gift tickets

After the transaction is executed, we log out and authenticate with the recipient's wallet to confirm the successful transfer (Figure 5.10).

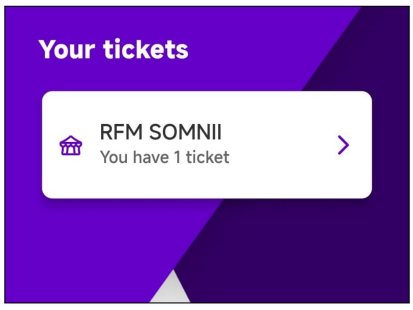

Figure 5.10: Profile page showing gifted tickets

The blockchain explorer records the gifting transaction details, as shown in Figure 5.11.

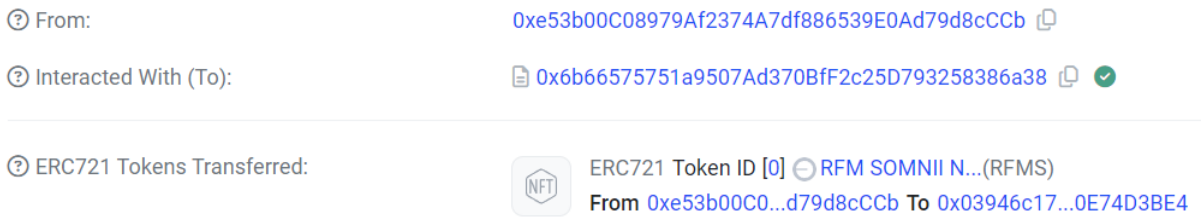

Figure 5.11: Details of the gift tickets transaction

The transaction shows that the NFT with ID 0 was transferred from the original user's address to the recipient's address.

5.3 Refunding Tickets

The refund process mirrors the gifting process, except that the tickets are burned in exchange for a refund. Figure 5.12 shows the prompt, indicating the refund amount, which in this case is 100%.

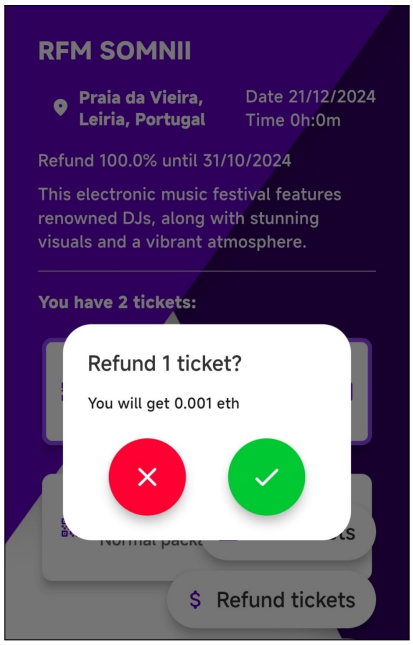

Figure 5.12: Refund tickets prompt

Once the transaction is executed, the blockchain records the details, as seen in Figure 5.13.

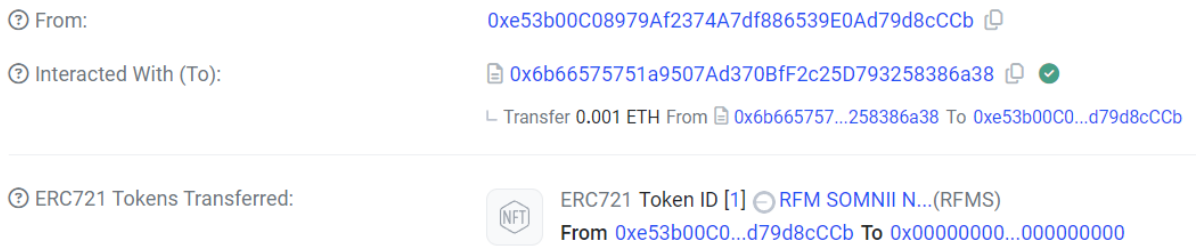

Figure 5.13: Details of the refund tickets transaction

The NFT with ID 1 is sent to the null address (0x00...00), indicating it has been burned.

Simultaneously, the user receives a refund of 0.001 ETH, and the event page reflects an increase in available tickets from 97 to 98 (Figure 5.14).

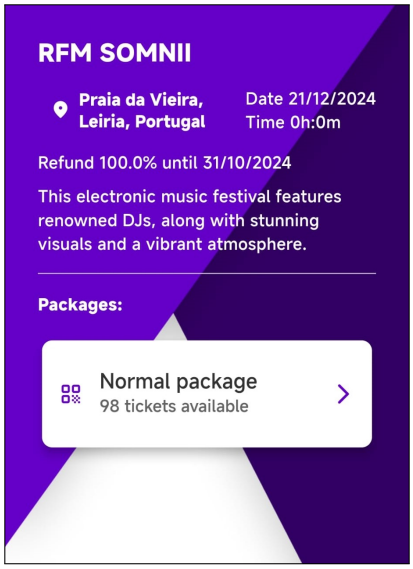

Figure 5.14: Third event page with 98 tickets available

5.4 Validating Tickets

As described in Section 4.2.2, the ticket validation process occurs between the user and the event validator. When the user selects the tickets to be validated, they are prompted to scan a QR code generated by the validator. Figure 5.15 shows the QR code displayed by the validator.

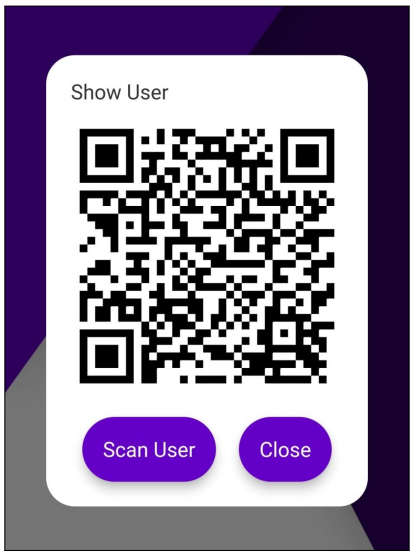

Figure 5.15: QR code generated by the validator

After scanning the QR code, the user's wallet opens and prompts them to sign a message. Figure 5.16 illustrates the message that the user will sign.

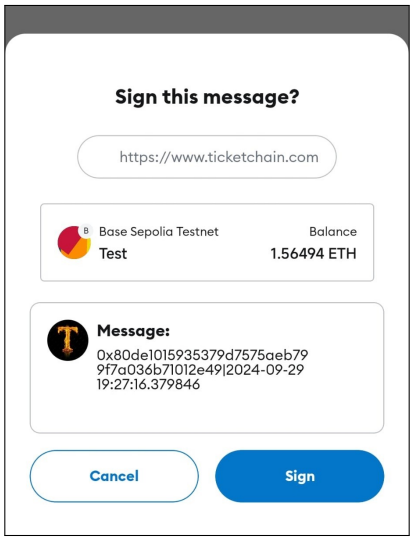

Figure 5.16: Message to be signed by the user

The message includes the validator's address along with the current date. Once the user signs the message, their device generates a QR code containing the necessary information for the validator, as shown in Figure 5.17.

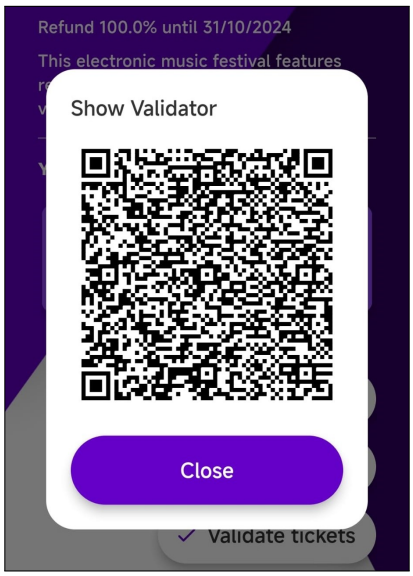

Figure 5.17: QR code generated by the user

The validator then scans the user's QR code, verifies the details, and completes the validation process. Upon successful validation, the prompt in Figure 5.18 is displayed.

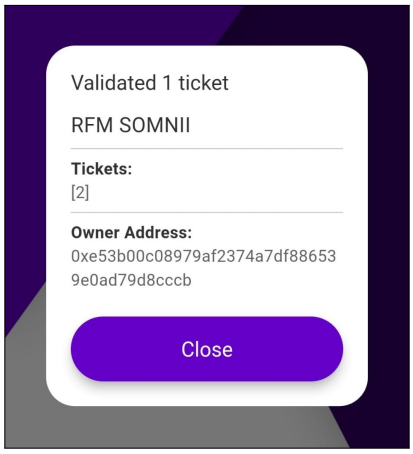

Figure 5.18: Validation success prompt

Once the ticket is validated, the user can see a checkmark next to the ticket, indicating its validated status, as shown in Figure 5.19.

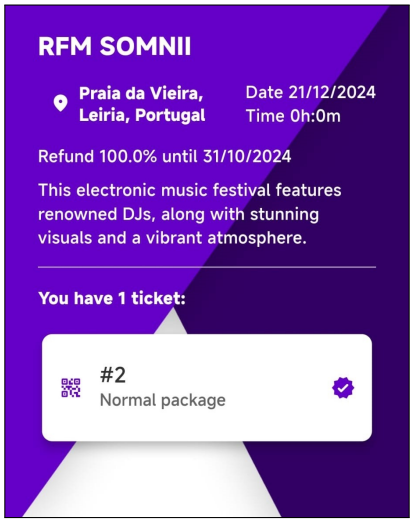

Figure 5.19: Validated ticket with checkmark

6

Conclusions

This project focused on the development of a decentralized ticketing system that allows users to purchase, gift, refund, and validate tickets. Leveraging blockchain technology, we deployed smart contracts on a testnet (test network), which ensured transparency and security throughout the ticketing process.

The primary objective was to create a system that could be viable for real-world use. We believe this goal has been met. The system is fully operational and accessible to any user with a compatible wallet on the deployed network. The system's scalability allows for the addition of multiple event organizers, events, and users without compromising performance.

A key feature of the system is its ability to prevent ticket scalping by utilizing blockchain's immutable nature. Additionally, the system ensures users cannot be defrauded, as the ticket validation process confirms legitimate ownership. Furthermore, the system enables validators to validate multiple tickets simultaneously, making it possible for one user to purchase tickets for a group.

We also incorporated a refund feature, enabling users to return tickets if they are unable to attend an event. This is an uncommon feature in traditional ticketing systems and could potentially benefit event organizers, as refunded tickets can be resold at the original price.

6.1 Limitations

Several limitations emerged during the development of this project, most of which are tied to the current state of blockchain technology.

The primary limitation is network fees, which users must cover when performing transactions, even if these fees are as low as 0.0001 euros. This poses a particular challenge for organizers, who must account for the cost of validators' operations. Though low fees accumulate slowly, they can become significant depending on the number of tickets validated.

Another limitation is transaction finality—the time required for a transaction to be confirmed on the blockchain. This impacts validators, who must wait for confirmations before allowing entry to an event. Networks offering fast finality often have high fees or limited scalability in terms of transactions per second.

A well-known network that meets these criteria is Solana. However, its incompatibility with the EVM presents another challenge, as it would require rewriting the smart contracts in Rust.

Additionally, while users can gift tickets through the system, they could still exchange tickets for money in person. While we encourage users to operate within the system, we cannot control such external transactions. However, patterns of frequent gifting could be flagged for potential scalping.

6.2 Future Work

Several potential improvements were identified during this project, which could enhance both user experience and system functionality.

Firstly, we developed a single mobile application combining user and validator functionalities to streamline testing. As future work, we propose splitting this into two separate applications: one for users and another for validators. This would allow for a more user-friendly interface for general users and a more robust interface for validators.

Another feature that was not implemented is a marketplace for ticket resale. Currently, users can only buy, gift, or refund tickets. A marketplace would enable users to resell tickets, giving others the opportunity to purchase tickets for sold-out events. By capping resale prices, the marketplace could also help combat scalping.

Additionally, we did not implement a web-based dashboard for event organizers. This dashboard would allow organizers to create and manage events more easily, automating the process of managing ticket metadata, which is currently handled manually.

Further improvements could involve integrating web2 technologies alongside web3 features. For example, offering users the option to authenticate using email or social media accounts, and abstracting the wallet connection process, would improve user adoption by simplifying the blockchain interaction.

Lastly, incorporating fiat currency (e.g., euros) as a payment option, alongside network-based currency, would provide users with a clearer understanding of ticket prices. Additionally, implementing seat-specific tickets, rather than just packages, would enhance the system's usability in venues like stadiums or theaters, elevating the platform to a more professional level.

References

[1] "Basescan." (), [Online]. Available: https://sepolia.basescan.org/address/0x87f4a5c17c2d3dc48f8e19d81e319230fa28f20d.

[2] "Ticketchain backend repository." (), [Online]. Available: https://github.com/Krfld/Ticketchain-Foundry.

[3] "Concept of blockchain technology." (), [Online]. Available: http://www-cs-faculty.stanford.edu/~uno/abcde.html.

[4] "Blueticket." (), [Online]. Available: https://blueticket.meo.pt/.

[5] "Bored ape yacht club." (), [Online]. Available: https://www.boredapeyachtclub.com/.

[6] "El corte inglés." (), [Online]. Available: https://www.elcorteingles.pt/.

[7] "Erc721 github." (), [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v5.0.1/contracts/token/ERC721/ERC721.sol.

[8] "Fnac." (), [Online]. Available: https://www.fnac.pt/.

[9] "Foundry." (), [Online]. Available: https://book.getfoundry.sh/.

[10] "Ticketchain frontend repository." (), [Online]. Available: https://github.com/Krfld/Ticketchain.

[11] "How does blockchain really work?" (), [Online]. Available: https://inlea.com/how-does-blockchain-really-works/.

[12] "Interplanetary file system." (), [Online]. Available: https://ipfs.tech/.

[13] "Metamask." (), [Online]. Available: https://metamask.io/.

[14] "Openzeppelin." (), [Online]. Available: https://docs.openzeppelin.com/contracts/api/token/erc721#ERC721.

[15] "Pinata." (), [Online]. Available: https://www.pinata.cloud/.

[16] "Ticketline." (), [Online]. Available: https://ticketline.sapo.pt/.

[17] "Wallet connect." (), [Online]. Available: https://walletconnect.com/.

[18] "Worten." (), [Online]. Available: https://www.worten.pt/.