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 in secondary markets. 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 utilizes 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 limitada nos mercados secundários. 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

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 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.

Another point, and the main reason to use blockchain, is to guarantee the user of any operation. In the traditional ticketing system, there can be human errors, or even fraud, and this assures users that any operation defined will never change and will be executed as expected.

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 one.

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

[user and validator app and repository]

1.4 Document Structure

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 often be complex to understand. At its core, it leverages cryptographic concepts to function effectively. In this section, we will explain key concepts of blockchain technology, including how wallets work and their interaction with the blockchain. We will also explore some of the most popular blockchain networks, token standards, and the emerging trend of non-fungible tokens (NFTs). Figure 2.1 illustrates common blockchain concepts, which we will discuss in detail across the following sections: wallets (1) in Section 2.1.3, networks (2) in Section 2.1.4, and smart contracts (3) in Section 2.1.5.

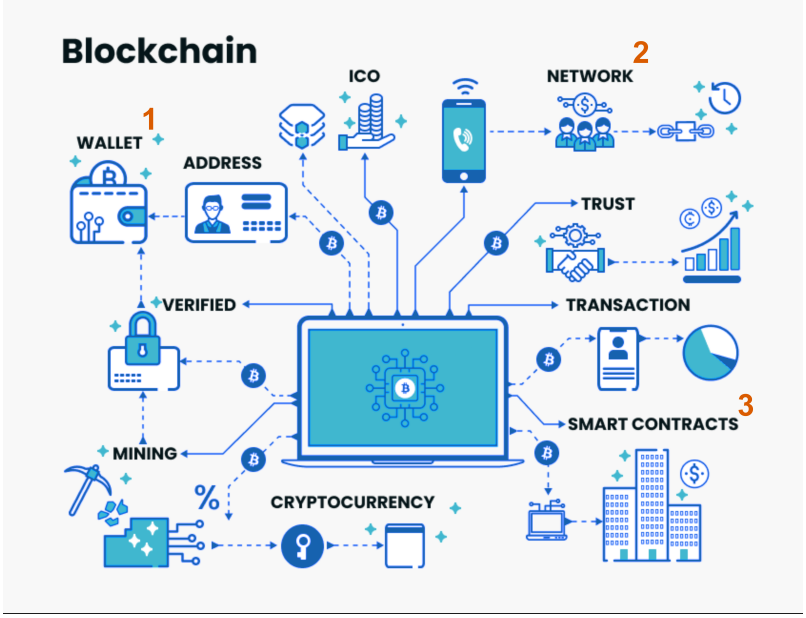

Figure 2.1: Key concepts of blockchain technology. Adapted from ...

2.1.1 Interacting with the Blockchain

A blockchain operates as a decentralized network of nodes, each maintaining a copy of the entire blockchain. To interact with the blockchain, users send transactions to the network, which are then validated and added to a block by the nodes. This process requires a wallet—an application that allows users to manage digital assets, interact with smart contracts, and send transactions. Wallets provide a user-friendly interface for accessing the blockchain, signing transactions with private keys, and viewing account balances and transaction history.

Figure 2.2 illustrates the steps involved in a blockchain transaction. To execute a transaction, users must sign it with their private key, proving ownership of the assets being transferred. The transaction is then broadcast to the network, validated by participants, and added to a block. Once confirmed, it becomes part of the immutable blockchain ledger, accessible to all network participants.

This procedure is essential for altering the blockchain state; transactions must be signed. In contrast, reading information from the blockchain requires no signatures—users can simply query the network for the desired information.

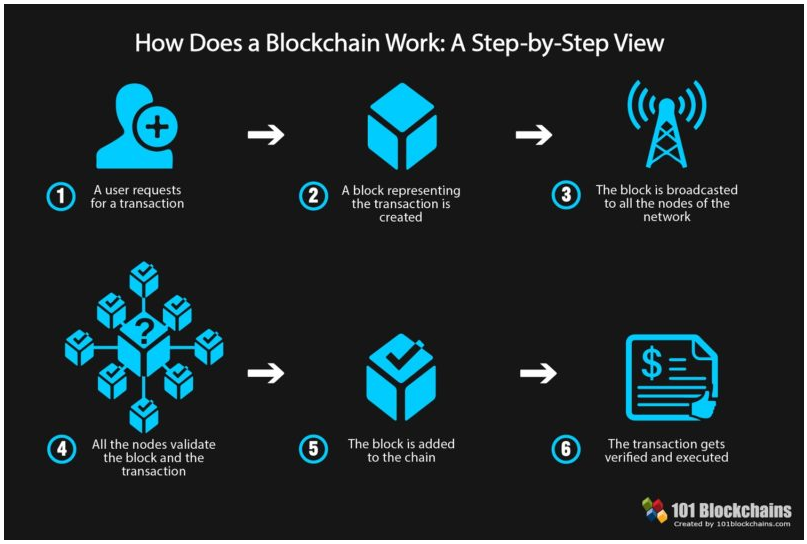

Figure 2.2: Overview of blockchain transaction processes. Extracted from ...

2.1.2 Blockchain

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

Decentralization: Blockchain operates on a decentralized network of nodes, eliminating a single point of failure or control. Each node maintains a copy of the entire blockchain.

Transparency: Data recorded on a blockchain is visible to all network participants, fostering trust as users can independently verify transaction integrity without intermediaries.

Immutability: Once recorded, transactions on a blockchain cannot be altered or deleted. This is achieved through cryptographic techniques, ensuring a tamper-proof historical record.

Security: Blockchain employs advanced cryptographic algorithms to secure transactions. Network participants verify and validate transactions through consensus mechanisms, preventing unauthorized changes.

Smart Contracts: Smart contracts are self-executing contracts with predefined rules encoded in software. They automate transactions and enforce agreements without intermediaries, enabling the creation of decentralized applications (DApps) on blockchain networks.

Blockchain technology has applications across various industries, including finance, supply chain management, healthcare, and decentralized finance (DeFi). Its potential to enhance security, transparency, and efficiency has led to widespread exploration of its capabilities. This ecosystem is often referred to as Web3, a new paradigm for the internet aimed at decentralizing control and empowering users with greater ownership and privacy over their data.

A compelling aspect of blockchain is how it harnesses human nature—greed and self-interest—to create a secure and reliable system. Nodes are incentivized to maintain blockchain integrity by earning cryptocurrency rewards, ensuring that a majority of the network remains honest and cooperative, thus making the system resistant to attacks.

2.1.3 Wallets

Cryptocurrency wallets utilize cryptographic principles to securely manage and interact with digital assets on blockchain networks. Key cryptographic aspects of wallets include:

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

Digital Signatures: When initiating a transaction, it is digitally signed with the wallet's private key, serving as proof of authorization. Digital signatures are generated using cryptographic algorithms such as ECDSA or RSA, depending on the blockchain protocol.

Hash Functions: Wallets use cryptographic hash functions to create unique representations of transaction data, known as transaction hashes. These hashes verify transaction integrity and prevent tampering.

Seed Phrases and Mnemonic Codes: Some wallets use mnemonic codes or seed phrases to back up access to funds if the private key is lost. These phrases, generated from random words, serve as a human-readable representation of the private key.

By leveraging these cryptographic techniques, wallets provide a secure means for users to store, manage, and transact with digital assets, ensuring confidentiality, integrity, and authenticity.

2.1.4 Networks

Since Bitcoin's inception in 2009, blockchain technology has evolved significantly, with numerous platforms emerging, each offering unique features. Prominent networks in the decentralized ecosystem include:

Bitcoin (BTC): The first and most well-known cryptocurrency, Bitcoin was introduced by an anonymous entity under the pseudonym Satoshi Nakamoto. It operates on a decentralized network using a Proof of Work (PoW) consensus mechanism, designed as a peer-to-peer electronic cash system.

Ethereum (ETH): Ethereum is a decentralized blockchain platform that enables the creation and execution of smart contracts and DApps. Transitioning from a PoW to a Proof of Stake (PoS) consensus model with the Ethereum 2.0 upgrade, it enhances scalability and energy efficiency.

Polygon (MATIC): Polygon is a Layer 2 scaling solution for Ethereum that addresses scalability issues by offering faster and cheaper transactions through sidechains.

Solana (SOL): Solana is designed for high-performance DApps and cryptocurrencies, using a unique combination of Proof of History (PoH) and Proof of Stake (PoS) for high throughput and low latency.

These examples highlight the diversity of blockchain networks, each with distinct features and capabilities. As the ecosystem continues to evolve, new technologies will emerge, expanding the possibilities for decentralized applications and digital assets.

Many networks, like Ethereum and Polygon, are built using Solidity, allowing compatibility with the Ethereum Virtual Machine (EVM). This enables developers to deploy smart contracts across multiple networks using the same codebase, enhancing reach

and leveraging network effects. Other notable EVM-compatible networks include Binance Smart Chain, Avalanche, Arbitrum, and Optimism.

2.1.5 Smart Contracts

Smart contracts are self-executing contracts with terms encoded in software. They automatically enforce agreements when predefined conditions are met, eliminating the need for intermediaries. Key characteristics include:

Autonomy: Once deployed, smart contracts operate independently, executing transactions without human intervention, ensuring impartial and transparent enforcement of terms.

Trust: Smart contracts leverage blockchain's trustless nature, allowing parties to rely on contract execution without a trusted third party, supported by the decentralized and immutable nature of the blockchain.

Security: Due to blockchain's cryptographic principles, smart contracts are highly secure and cannot be altered once deployed, providing reliability.

Efficiency: By automating execution, smart contracts reduce costs and processing times associated with traditional contract enforcement, enhancing overall efficiency.

Versatility: Smart contracts can perform complex functions beyond simple transactions, managing digital assets and interacting with other contracts, enabling diverse DApp functionalities.

Smart contracts have applications across finance, supply chain management, real estate, healthcare, and more. Their potential to revolutionize contract execution in the digital age is significant.

2.1.6 Token Standards

Token standards define the rules and functionalities of digital tokens on blockchain networks, facilitating interoperability and usability. Key token standards include ERC-20, ERC-721, and ERC-1155, as illustrated in Figure 2.3.

Figure 2.3: Overview of token standards. Extracted from ...

ERC-20 (Ethereum Request for Comment 20): The most widely used token standard on Ethereum, ERC-20 governs fungible tokens, allowing for seamless trading on exchanges. It includes functions for transferring tokens and querying balances.

ERC-721 (Ethereum Request for Comment 721): This standard governs non-fungible tokens (NFTs), each of which is unique and represents ownership of a specific asset. ERC-721 tokens are commonly used for digital art and collectibles.

ERC-1155 (Ethereum Request for Comment 1155): This hybrid standard supports both fungible and non-fungible tokens within the same contract, allowing efficient management of multiple token types and reducing deployment costs.

These standards represent just a few examples shaping the landscape of tokenization. As blockchain technology evolves, new standards will likely emerge, driving further adoption across various industries.

2.1.7 Non-Fungible Tokens (NFTs)

In the context of event ticketing systems, NFTs offer great potential by providing a secure and verifiable means of ticket issuance, transfer, and validation. NFT-based tickets can incorporate unique metadata—such as event details, seat numbers, and access permissions—creating a rich, customizable experience for organizers and attendees. They can also be used to issue limited edition or VIP tickets, granting exclusive access and additional benefits to holders.

2.2 Related Work

In this section, we explore some of the most prominent traditional ticket-selling platforms and their functionalities. Additionally, we discuss the application of NFTs (Non-Fungible Tokens) in industries such as art and gaming, highlighting key projects and trends.

2.2.1 Traditional Ticket-Selling Platforms

Several traditional platforms are widely used for ticket sales today. Ticketline and Blueticket are two of the largest and most reputable Portuguese companies specializing in ticket sales for a variety of events, including concerts, sports matches, theater performances, and exhibitions. These platforms also collaborate with physical retailers, such as Worten, Fnac, and El Corte Inglés, offering users the convenience of purchasing tickets in person.

Both platforms maintain comprehensive websites where event organizers can list their events. Users can browse events, view detailed information, and purchase tickets online. Options are available to either print the tickets at home or collect them at physical outlets.

2.2.2 Application of NFTs

NFTs have seen widespread adoption in industries like art and gaming, where they are used to represent ownership of unique digital assets. In the art industry, NFTs are employed to create and sell digital artworks, which are authenticated on the blockchain to ensure uniqueness. These digital pieces can be traded or sold, with creators often benefiting from resale royalties.

In the gaming industry, NFTs are used to represent unique in-game items, such as skins, weapons, or characters, that can be bought, sold, and traded between players. These items can enhance gameplay or serve as collectibles. One of the most famous NFT projects is the Bored Ape Yacht Club, which features a collection of digital apes (in JPEG format) that have sold for millions of dollars. Beyond just being digital art, ownership of these NFTs grants access to exclusive events, merchandise, and a vibrant community of collectors.

3

Requirements Analysis

This Chapter will present the requirements analysis for the system and is divided into 3 sections: the requirements in the Section 3.1 where it presents the functional and non-functional requirements, the use cases in the Section 3.2, and the system architecture in the 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.

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.

Table 3.1: Functional Requirements

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.

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

Table 3.2: Non-Functional Requirements

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. 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: Ticketchain owner, organizer, validator, and user. Each actor shares a common use case: authentication, which is responsible for verifying user identities within the system.

In Section 3.2.1, we detail the use cases for the Ticketchain owner, focusing on system settings management and the oversight of event organizers. Section 3.2.2 covers the use cases for organizers, including event creation, management, and validator selection.

Section 3.2.3 describes the validator's use case, which involves ticket validation to ensure that only users with valid tickets can enter events. Finally, Section 3.2.4 outlines the use cases for common users, such as purchasing, gifting, refunding, and reselling tickets.

3.2.1 Ticketchain Owner

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

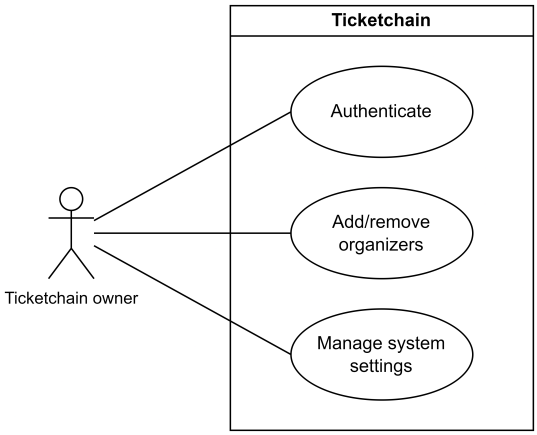

Figure 3.1: Ticketchain Owner Use Cases

3.2.2 Organizer

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

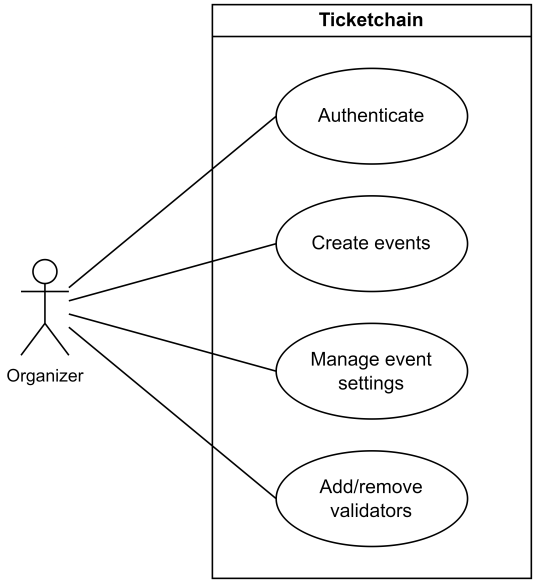

Figure 3.2: Organizer Use Cases

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

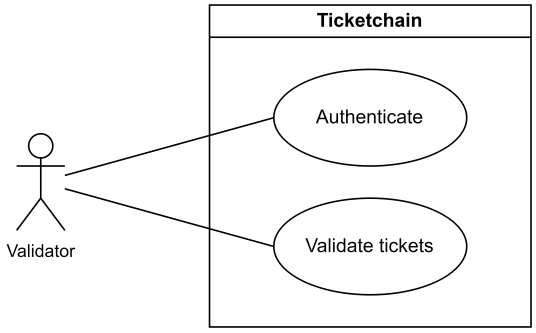

Figure 3.3: Validator Use Cases

3.2.4 Common User

Figure 3.4 presents the use cases for common users. 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.

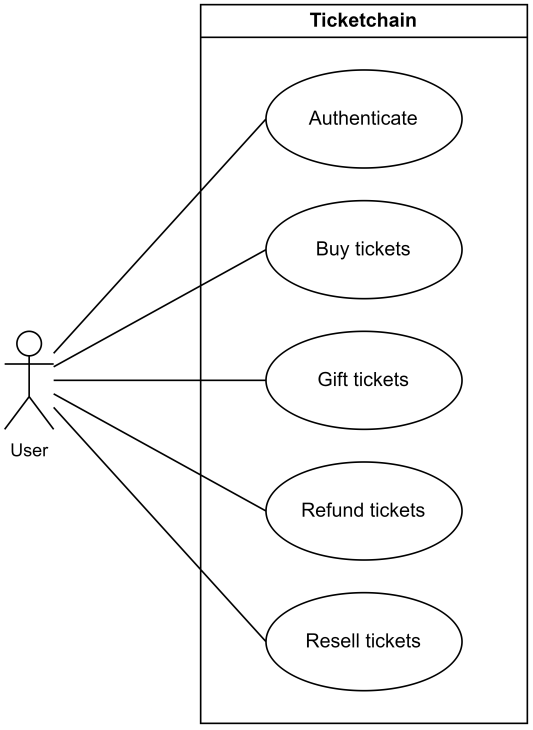

Figure 3.4: Common User Use Cases

3.3 Architecture

Being a blockchain project, the goal is to not use the so called Web2 technologies, this means the traditional approaches of having a backend running on a server and similar services, but rather to use only Web3 technologies for us to understand exactly the limitations there are by adopting solely the blockchain ecosystem. Ideally, of course, in the future this project benefits greatly by merging these two approaches.

The system's architecture is illustrated in Figure 3.5, showing how each component of the project interacts, aligned with the requirements discussed in previous sections.

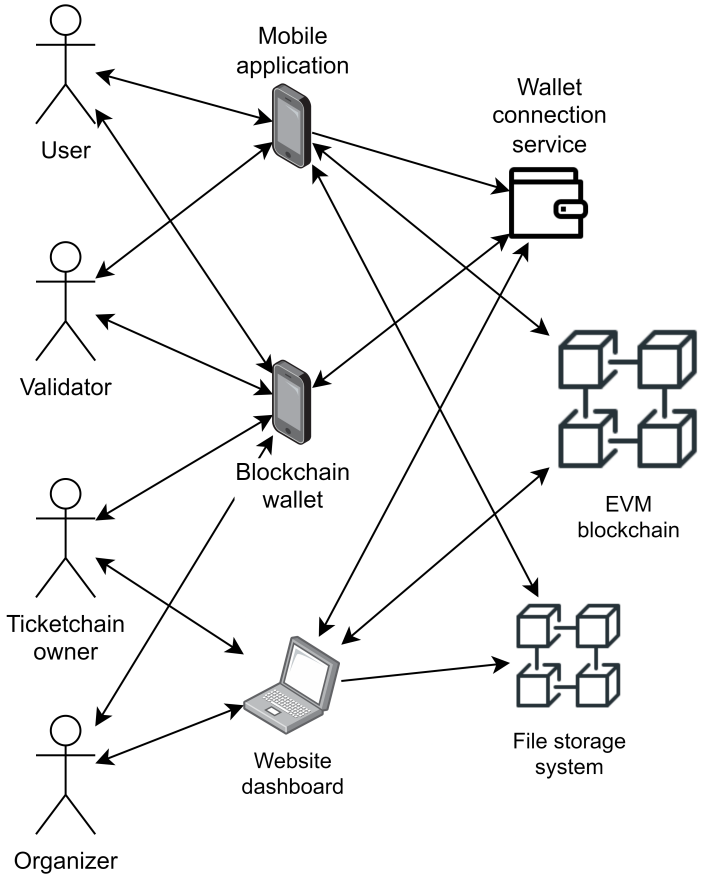

Figure 3.5: System Architecture

The architecture comprises a mobile application for both regular users and validators, as well as a web dashboard for the Ticketchain owner and event organizers to manage system settings and events. Due to time constraints, this part of the system has not yet been implemented; for now, interactions will be conducted directly with the smart contract.

As detailed in the use cases section, all users must authenticate before accessing the system. Instead of a traditional login using email and password, authentication will require wallet software to interact with the blockchain. Therefore, a wallet connection service is necessary to link users' wallets to the system.

Once the wallet connection is established, the system will communicate with the smart contract to display event-related information and update the status of events or tickets. The entire codebase will be deployed on an EVM-compatible blockchain. For ticket images, a file storage system will be needed. While image uploads will typically be managed through the dashboard, this feature has yet to be implemented, necessitating manual uploads for the time being.

4

Implementation

[Brief introduction of the chapter]

4.1 Mobile App

The mobile app is the main interface for the users to interact with the system, where they can authenticate, see the events, and buy and manage tickets. It will have a main page to check the events and a page for each event, where the user can see the details and buy the tickets, and will also have a page to see the tickets he owns.

We also made the validator logic in the same app to simplify the process, so we don't have to make a separate app for the validators, however in a real scenario we would have separate apps.

The app will be developed using the Flutter framework, which is a cross-platform framework that allows us to build apps for Android and iOS from a single codebase. It's made by Google and it's very popular for its ease of use and performance.

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.

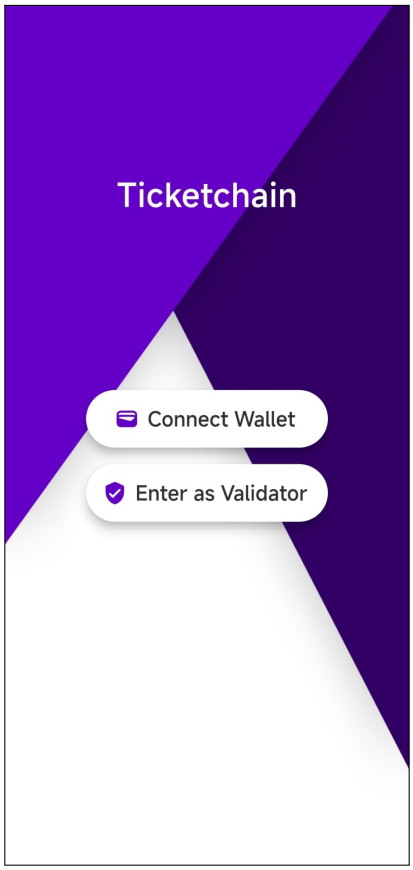

Figure 4.1: Authentication page

To authenticate the users, we will use the Wallet Connect service, which supports a variety of wallets to interact with. What this service does is establishes a connection between the app and a wallet, so that when a function needs to be called, the wallet receives the prompt and signs the transaction after the user's approval.

The Figure 4.2 shows what happens when we try to authenticate as common users which triggers the Wallet Connect service. This lists all the wallets that are supported by the service and the user can choose the one he uses.

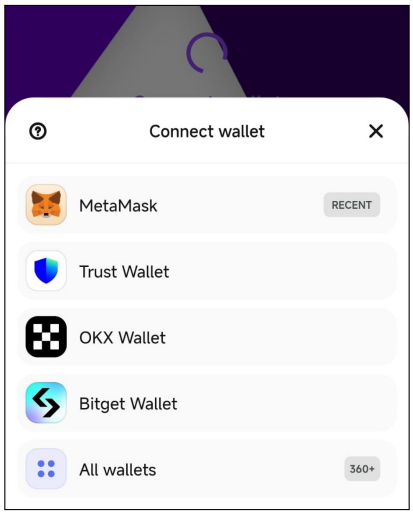

Figure 4.2: Wallet connect prompt

After choosing one, the wallet app will open automatically and the user will have to approve the connection, as the Figure 4.3 shows. We're using MetaMask as the external blockchain wallet, which is the most popular blockchain wallet and it's available as a browser extension and as a mobile app. To use it, you just have to create a wallet there and you'll be all set to start interacting with the app.

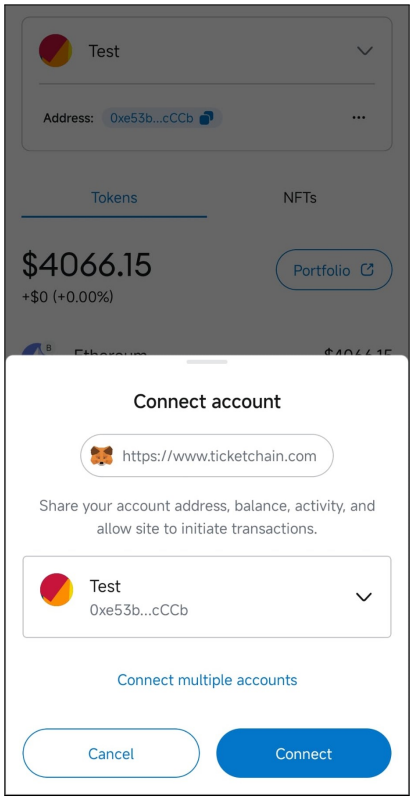

Figure 4.3: MetaMask connect

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 main page of the app, where he can see the events that are available. The Figure 4.4 shows the main page of the app, where the user can see the events and search for them.

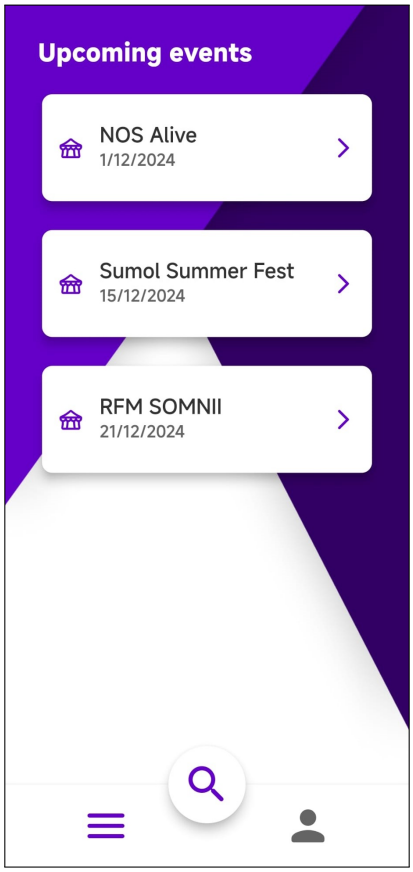

Figure 4.4: Main page

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

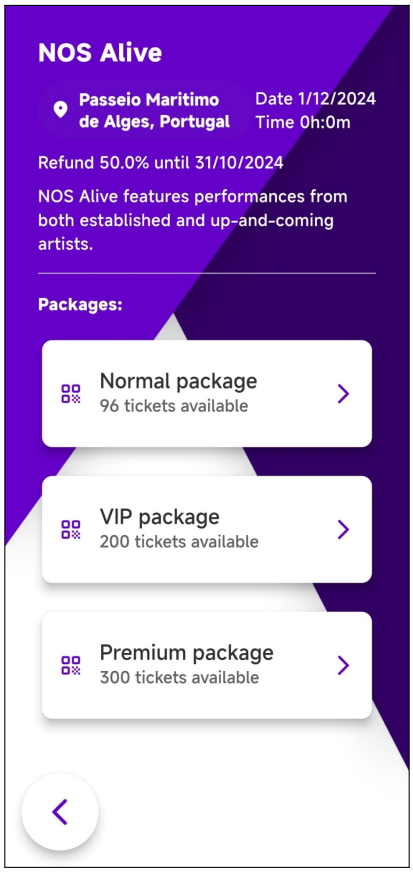

Figure 4.5: Event page

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.

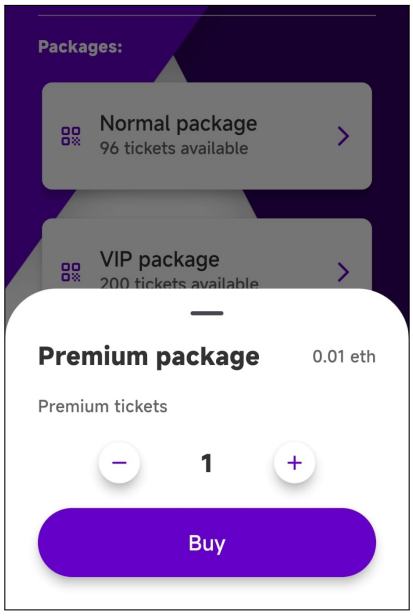

Figure 4.6: Buy tickets prompt

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.

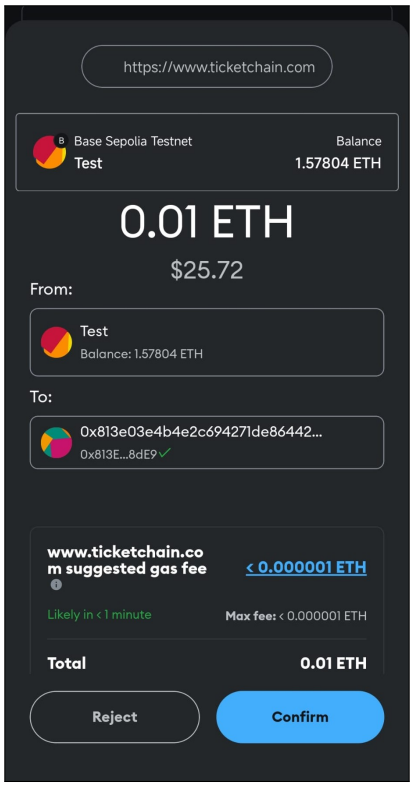

Figure 4.7: MetaMask transaction prompt

4.1.3 Tickets

After confirmation, the user is redirected back to the app, where he will be able to see the events which he owns any tickets, on the profile page, the second tab with the profile icon, along with the button to disconnect from the wallet, like shown in the Figure 4.8.

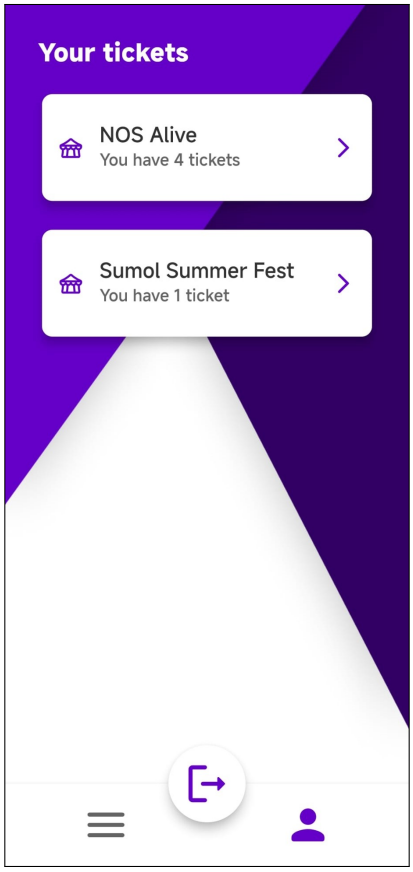

Figure 4.8: Profile page

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. This means the ticket has already been validated. In the real world, a user do this doesn't make sense, but we did it to make sure we handle every case.

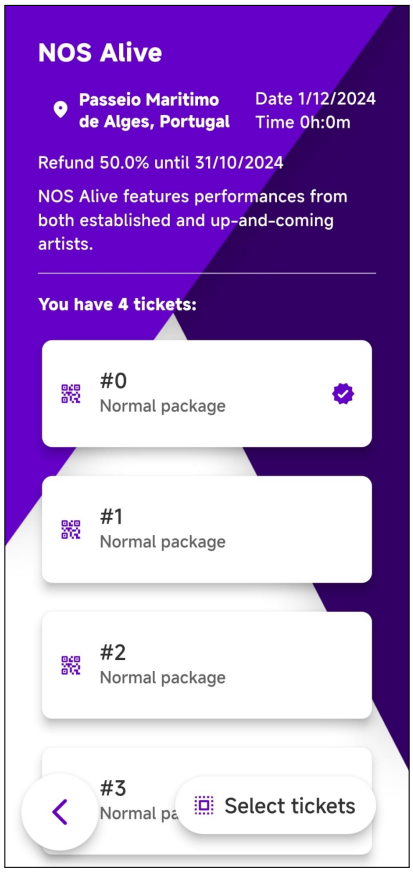

Figure 4.9: Tickets page

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

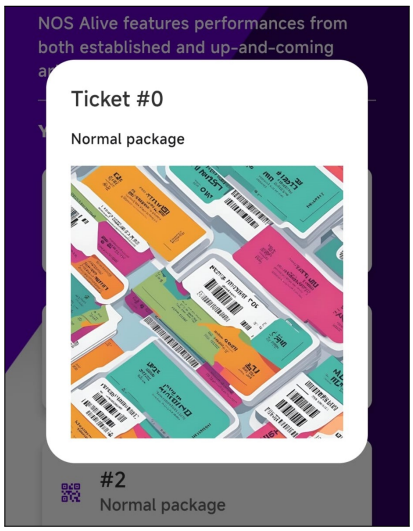

Figure 4.10: Ticket information

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, like 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.

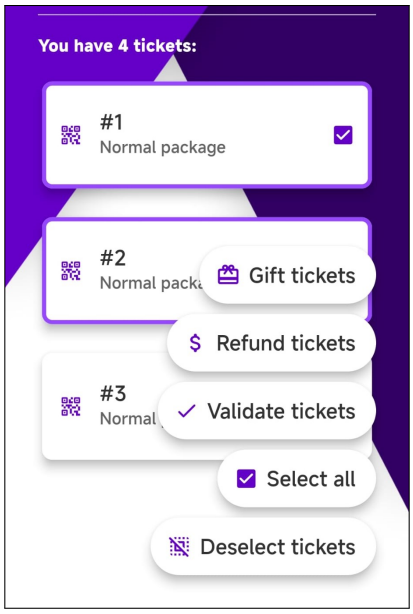

Figure 4.11: Ticket operations

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 TODO.

4.2 TODO Validator App

For the validator app, we will have a simple interface for the validators to execute the process mentioned in the Figure 4.19.

[TODO mention the wallet being local]

Since the validators need to sign the message, the organizer just has to make sure the validators have enough funds to pay for the transaction.

4.3 Solidity

We'll be doing the smart contracts in Solidity, because it's the most popular language for Ethereum smart contracts, and makes it easy to deploy to any EVM compatible blockchain. This language is similar to other languages like C++, but it has some unique features that make it very powerful for smart contracts.

One important aspect is that when a contract is public, anyone can call its functions, that's why we need to restrict their access and take into account that possibility when defining the operations of the entire contract. So it's always important to have a mind-set that any function (endpoint) can be called by anyone, which is a pretty different approach from traditional web development.

Another thing is that contracts run atomically, so if something should fail or revert, the changes don't take place. Modifiers are a good example where this can be used to prevent reentrancy attacks, which are a common security issue in smart contracts, and restrict the access to certain functions, reverting if an intruder tries to call them.

We can define, for example, a call to register an event to be restricted to only organizers or a call to validate tickets to be limited to only validators. This is possible because there are a few keywords like *msg.sender* that tell us which user address called the function and *msg.value* that tell us how much value was sent with the transaction. This value is essentially the amount of money the user sends in a transaction, necessary, for example, to buy the tickets. We'll check if the value matches the correct price and reverts otherwise.

Each blockchain network has its own currency, in which is commonly referred to as ether, on the EVM networks. This is the unit of value that is used to pay to execute transactions, but in solidity we cannot work with floating point numbers, so we have to work with the smallest unit of ether, which is called wei. Similar to how we have 1 euro is 100 cents, we have 1 ether is 10^{18} wei, same as on the bitcoin network we have the 1 bitcoin (BTC) is 10^{8} satoshis (SATS). This is the unit being used when we call the *msg.value* variable, so we always have to account for the conversion.

Another unique feature of Solidity is the *address* type, which is basically a string strict to the Ethereum address format. That's how we'll be storing the addresses of the organizers and the events on the contracts. The only difference between a user and a contract is that a contract has code associated with it, so it can execute functions and store data, while a user can only send transactions.

This way we can understand why, to send a transaction, paying for the tickets for example, we send value along with it, matching the expected price according the logic

of the buy method. Essentially we're sending money to the contract, which is stored in the contract's balance, like a normal user wallet. This is how the contract can store money and manage it, for then the organizer to withdraw its profit to his own wallet.

The deployment will be done using Foundry, 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.

4.4 Main Smart Contract

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 and you can 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 to keep track of these events.

With this in mind, we created the Ticketchain main smart contract UML, like the Figure 4.12 shows. This contract will track the organizers and the events associated with the system, along with the method to register a new event with the necessary data, restricted to only organizers (to avoid unauthorized people to interact).

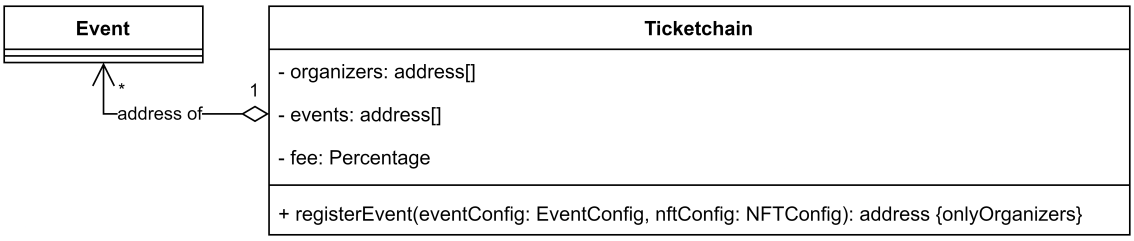

Figure 4.12: Main smart contract UML (simplified)

With this structure, since we have this main contract where all the events of the system are stored, we can simply make a call to get them all, showcasing them in the app's home page for users to search. Any event that is deployed outside the system or if it gets removed from there, it won't be shown to the users.

4.5 Event Smart Contract

For the event contract, we'll be extending the ERC721 standard and 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 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.

4.5.1 ERC721 Structure

The standard was obtained through the OpenZeppelin 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 [TODO ref], and looking into the most important variables and methods of the standard shown in the Figure 4.13, 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 IPFS, which is a decentralized storage system, so the metadata is not stored on the blockchain itself (onchain), which would be very expensive, but rather on a decentralized storage system (offchain), which is much cheaper.

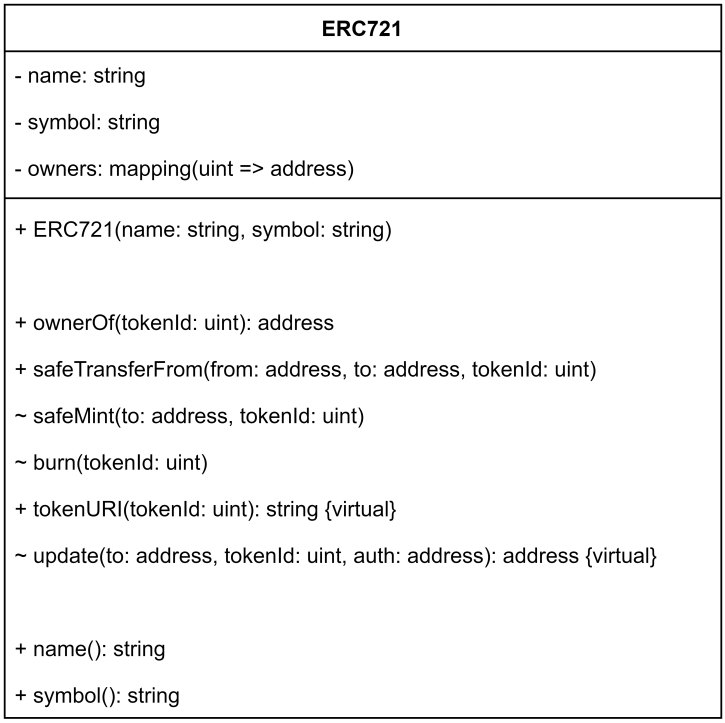

Figure 4.13: ERC721 UML (simplified)

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 4.13 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 4.17, about how the packages logic is implemented.

4.5.2 Event Behavior

So the event 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 the event 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 4.13, 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.

The Figure 4.14 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.

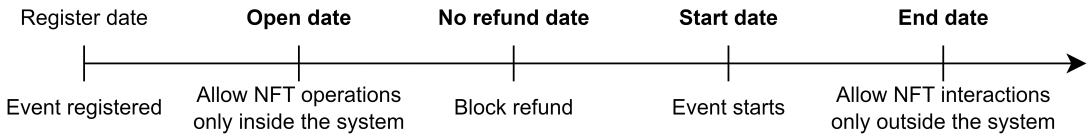

Figure 4.14: Event lifecycle

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, like shown in the Figure 4.15, 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.

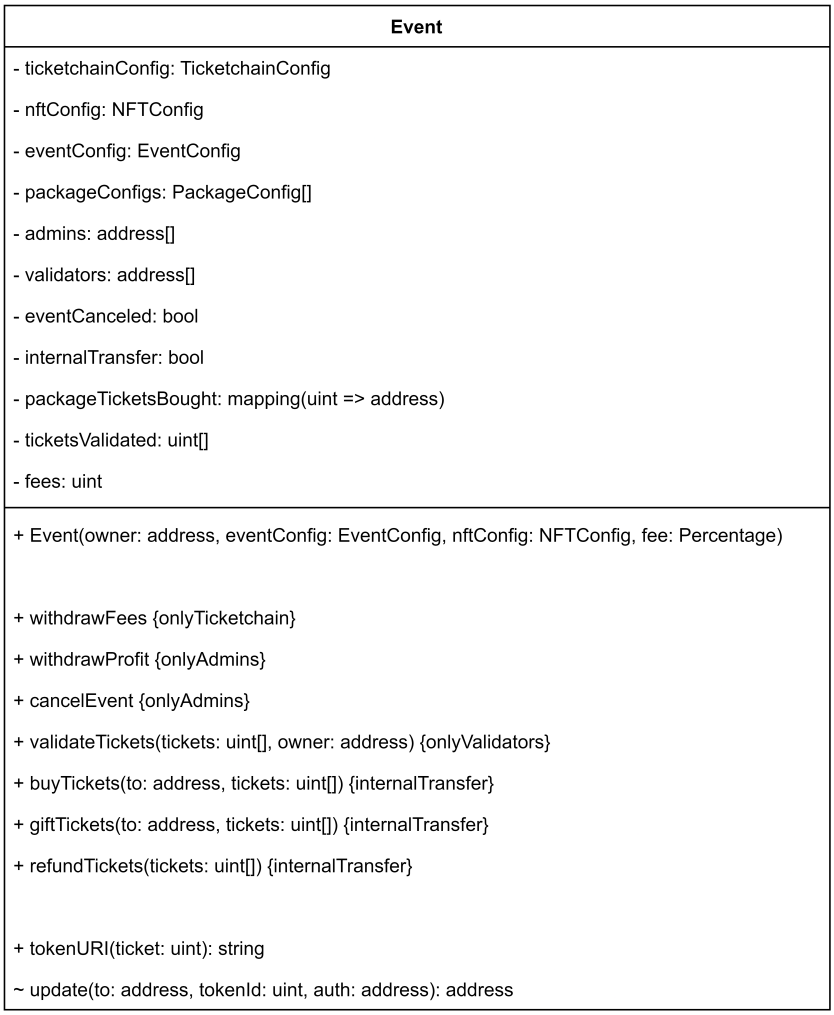

Figure 4.15: Event smart contract UML (simplified)

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 4.16.

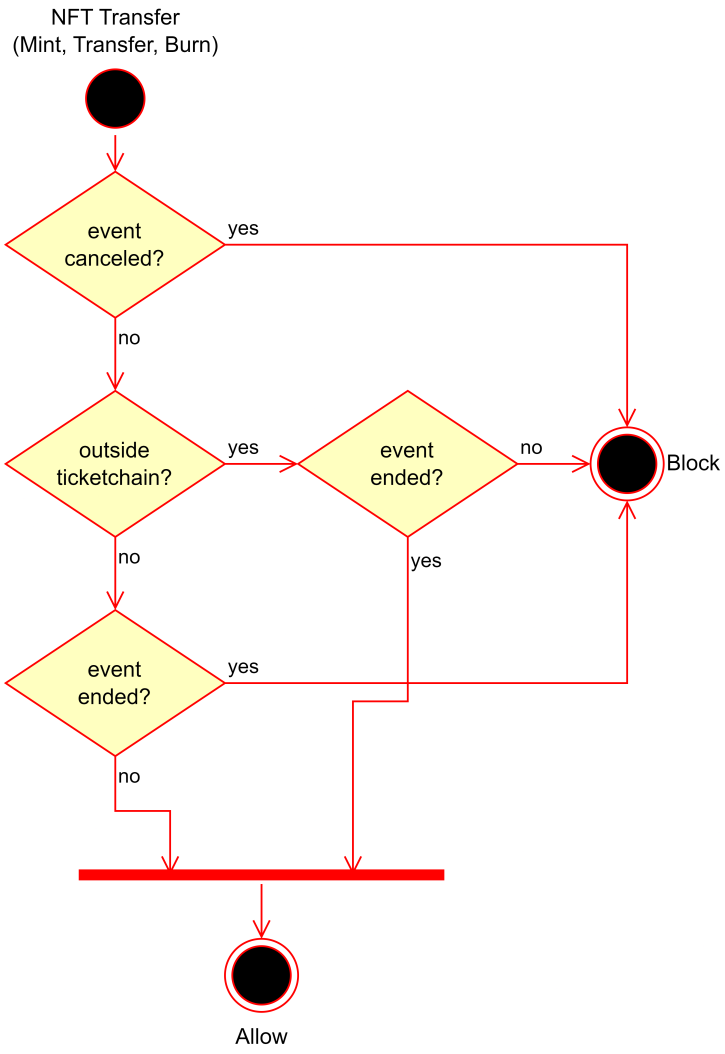

Figure 4.16: NFT flowchart

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 4.5.

4.5.3 Structs

As is possible to see in the Figure 4.15, and also in the Figure 4.12, 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 small values in wei, like 1000 wei, so applying this formula won't lose much precision (note that 1 ether is 10^{18} wei).

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 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 4.5.4.

4.5.4 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 4.17 illustrates.

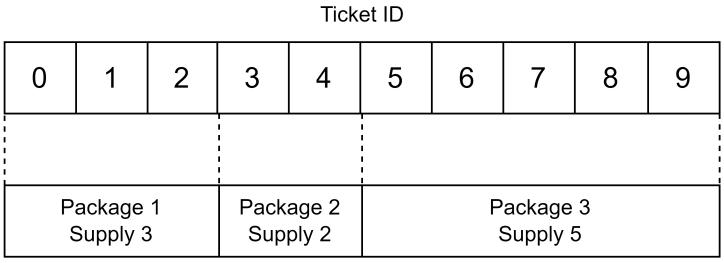

Figure 4.17: Package logic

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.

4.5.5 Metadata Storage

To store the NFTs metadata files, we'll be using the InterPlanetary File System (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 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 4.5.4, 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 4.18 shows, being the packages 1 and 3 with shared metadata, and the package 2 with individual NFTs.

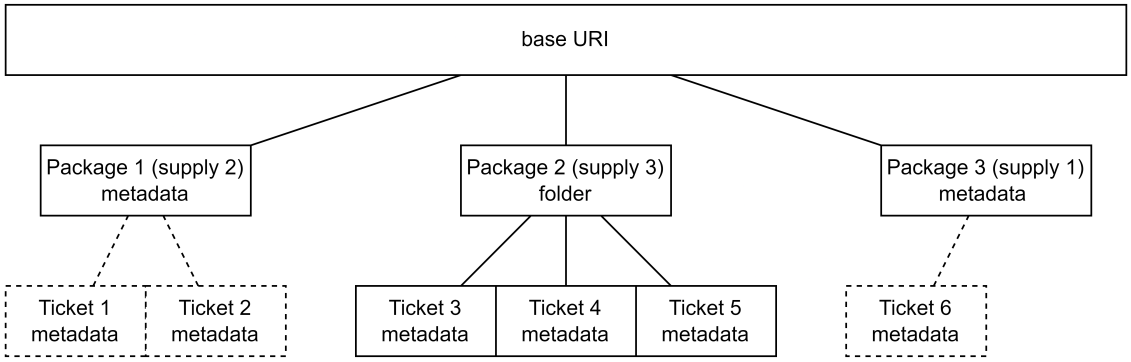

Figure 4.18: Metadata storage

4.5.6 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 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 this into account when setting the system fee, to make it sustainable for the organizer to use our system.

We'll set a fee on the main 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 tickets 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 compensates for the trust that the users and organizers will have on it.

4.5.7 Ticket Validation

For the ticket validation, 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, 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 normal blockchain transaction.

Essentially the user will have to sign a message for the validator to check if the signature is valid, and if it is, it means the user is the actual owner of the ticket. There's a cryptographic method to recover the address of the signer using the original message and it's signature.

Knowing this, 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, anyone who has it, could pretend to be a different address. The idea here is to have a unique message for each user at the time of the event, so it forces the user to sign it in the moment.

The process defined is shown in the Figure 4.19, where the user reads the QR with a generated message from the validator, 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 from the signature and the message, 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.

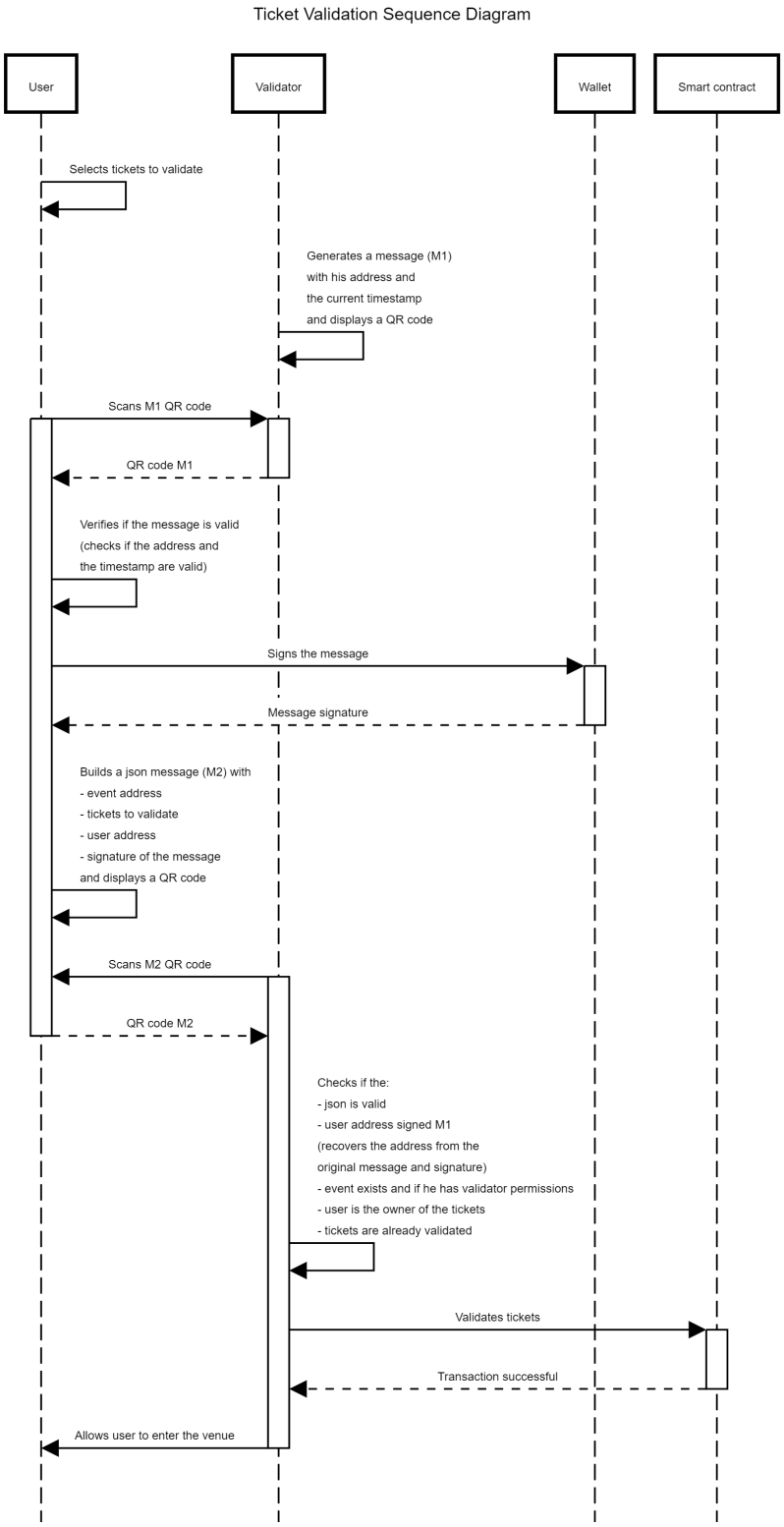

Figure 4.19: Ticket validation

All this is reduced to a single transaction on the blockchain because, depending on the network, the finality of a transaction (the time it takes for a transaction to be fully

registered on the blockchain) can take a while. This will be discussed in the next Section 4.6. 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.

[vv network choice and fees maybe chapter 3 vv]

4.6 Network Choice

The network choice is a very important aspect of the project, because it defines the limitations and the costs of the system. We want to aim for a blockchain that has low fees, is fast, secure and scalable, and since this is done using solidity, we have to aim for an EVM compatible network. To see which network matches our requirements, we need to analyze the statistics of the most popular EVM compatible networks, because these are the more secure ones.

Following the Figure 4.20, we can see the main differences between the networks with the highest transactions per second (TPS). The main aspects to take into account from this Figure are the TPS and the finality.

#	NAME	REAL-TIME TPS	MAX REC. TPS	MAX THEOR. TPS	BLOCK TIME	FINALITY
1	Hedera	1,812 tx/s	3,288 tx/s	10,000 tx/s	2s	7s
2	Solana	703 tx/s	7,229 tx/s	65,000 tx/s	0.43s	12.8s
10	Aptos	224 tx/s	10,734 tx/s	160,000 tx/s	0.22s	0.9s
3	Stellar	122 tx/s	176 tx/s	1,137 tx/s	5.78s	6s
4	opBNB	94.42 tx/s	548 tx/s	4,762 tx/s	1s	16m
6	Tron	83.97 tx/s	236 tx/s	2,516 tx/s	3s	57s
7	Sei	46.28 tx/s	256 tx/s	12,500 tx/s	0.43s	0.38s
5	Base	44.68 tx/s	293 tx/s	1,429 tx/s	2s	16m
9	BNB Chain	39.14 tx/s	1,731 tx/s	2,222 tx/s	3.01s	7.5s
8	Polygon	35.01 tx/s	429 tx/s	649 tx/s	2.17s	4m 16s
13	Arbitrum	21.42 tx/s	669 tx/s	40,000 tx/s	0.25s	16m
15	Celo	12.93 tx/s	268 tx/s	476 tx/s	5s	0s
11	Ethereum	12.72 tx/s	62.34 tx/s	119 tx/s	12.06s	16m
17	Algorand	10.55 tx/s	5,716 tx/s	9,384 tx/s	2.86s	0s
14	Concordium	7.98 tx/s	38.49 tx/s	2,000 tx/s	2.29s	4s
12	Bitcoin	6.91 tx/s	13.2 tx/s	7 tx/s	10m 4s	1h
19	Kaia	6.51 tx/s	3,142 tx/s	28,922 tx/s	1s	0s
16	Optimism	6.46 tx/s	67.41 tx/s	714 tx/s	2s	16m
18	Fantom	3.57 tx/s	181 tx/s	1,476 tx/s	0.87s	0s
21	Scroll	3.25 tx/s	59.43 tx/s	136 tx/s	2.88s	18m
23	Avalanche	1.91 tx/s	92.74 tx/s	357 tx/s	2.1s	0s
20	Arthera	1.82 tx/s	112 tx/s	9,804 tx/s	2.21s	1s
22	MultiversX	1.79 tx/s	220 tx/s	15,000 tx/s	6s	6s
24	Gnosis Chain	1.21 tx/s	80.9 tx/s	156 tx/s	5.19s	4m
25	Starknet	0.92 tx/s	31.15 tx/s	238 tx/s	3m 23s	2s
28	Telos	0.68 tx/s	8.46 tx/s	15,200 tx/s	0.5s	2m 6s

Figure 4.20: Network comparison. Extracted from ...

Blockchains with higher TPS means they are more scalable, meaning they can process that amount of transactions in one second. Similar to how Visa and Mastercard work, they can process thousands of transactions per second, so the users don't have to wait for a long time for the transaction to be processed. In our case, a high TPS is important, but even more is the finality.

The finality is the time it takes for a transaction to be fully registered on the blockchain,

meaning the time it takes for the transaction to be irreversible. This is important because we want the users to be able to enter the event as soon as possible, so the transaction to validate the tickets must be as fast as possible. This is the value that, if it takes too long, we will be waiting for each user individually to enter the event, which would be a problem. It's possible to see Ethereum down there, which has an average of 16 minutes for the finality, which is way too much for what we need.

Unfortunately a lot of networks there are not EVM compatible, so we have to choose between the ones that are. The two options that we could consider are the BNB Chain (also known as Binance Smart Chain (BSC)) and the Avalanche, being the first one the most popular and with higher scalability, but the second one has a way better finality.

The Avalanche chain would be the best bet, but having close to 2 tx/s (transactions per second), maybe its not scalable enough. For the BSC, it has a good enough finality and a good TPS so it would be a good choice for the project. This chain is also very popular and was created by Binance, one of the biggest companies in the blockchain ecosystem, so it's very secure and reliable.

4.6.1 Fees

With this in mind, we still need to take into account the network fees. If this chain had high fees, then we would be forced to switch to another. Luckily, the BSC has very low fees, as the Figure 4.21 shows, being the peak at 0.50 USD. Note that these are the fees to pay for a transaction, so if a user wants to buy multiple tickets, he can make it all at once instead of several individual purchases.

Blockchain	Average Fee (USD)	Peak Fee in 2024 (USD)
Bitcoin	$6.96	$10.28
Ethereum	$2.50	$50.00
Ripple	$0.01	$0.02
Litecoin	$0.10	$0.20
Solana	$0.005	$0.01
Cardano	$0.04	$0.10
BNB Chain (Binance Smart Chain)	$0.10	$0.50

Figure 4.21: Network fees. Extracted from ...

These fees are necessary as they are the cost for making the network run. This is common in the blockchain ecosystem, and it's a way to maintain the network secure and reliable, because the people processing the transactions are rewarded for their work, so they have an incentive to keep the network running. So we can never run away from

this kinds of fees, but we can choose networks that have lower fees, so the users pay the least amount possible.

5

Results

[brief introduction of the chapter]

5.1 Smart Contract Interactions

Like mentioned, the blockchain is a public ledger that stores all the transactions that are made, so we can see all the interactions with the smart contracts. We deployed the smart contracts on a testnet, so that we don't have to use real funds. This way we can simulate what would happen on the mainnet without any risks or cost. This was done with Foundry, like mentioned before, and we created some scripts to populate the system to be able to see it on the blockchain. These scripts deploy the smart contracts, adds an organizer, creates a few events, and adds some packages to each.

For each network there is an explorer that allows us to see all the transactions that are made, and the Figure 5.1 shows what it looks like. What we're seeing is the page of the interactions with the main smart contract and we can see when the contract was deployed, the adding of an organizer to the system, and the creation of the 3 events.

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: System Transactions

If we go into one of the events creation, we can see the details of the transaction and its logs. On the details we see the information of the transaction itself, like the hash, the status, and the gas information. This gas information is the cost of the transaction, and it's paid in the network's currency. The logs are the events that are emitted by the smart contract, and in this case we see the event of the creation of the event, like the Figure 5.2 shows. Not to confuse the events that are emitted by the contract and the event that is created by the organizer.

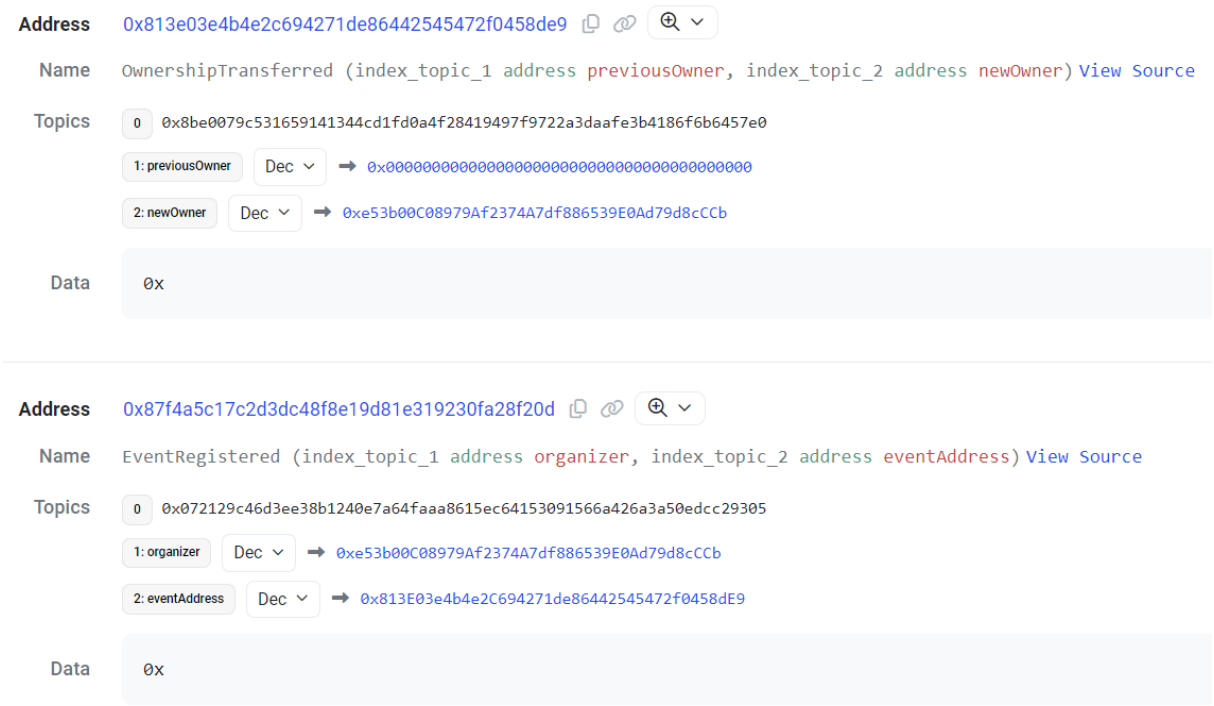

Figure 5.2: Event Transaction Logs

This shows two events being emitted and the first is the transfer of ownership. This is because we set the owner of the event contract to be the organizer, so that he has full control over it. The second is the creation of the event, and we can see the organizer

that created it and the address of the event. When we search for that address we see its interactions, like the Figure 5.3 shows. Analyzing its entries we see the addition of 3 packages, 2 additions of validators, one entry of buying tickets and one ticket validation. The added packages were done by the initial script to populate the event, the rest of the interactions were done manually.

Transaction Hash	Method	Block	Age	From		To
0xf83e0c859df...	Validate Tickets	12875399	48 days ago	0x6d8BBeFE...9D4A673F5	IN	0x813E03e4...2f0458dE9
0xc06067a650...	Add Validators	12875369	48 days ago	0xe53b00C0...d79d8cCCb	IN	0x813E03e4...2f0458dE9
0xb16861e71d...	Buy Tickets	12821542	49 days ago	0xe53b00C0...d79d8cCCb	IN	0x813E03e4...2f0458dE9
0xe2ac8cf9a75...	Add Validators	12785771	50 days ago	0xe53b00C0...d79d8cCCb	IN	0x813E03e4...2f0458dE9
0x6448ef94da9...	Add Package ...	12785771	50 days ago	0xe53b00C0...d79d8cCCb	IN	0x813E03e4...2f0458dE9
0x2335e549b9...	Add Package ...	12785771	50 days ago	0xe53b00C0...d79d8cCCb	IN	0x813E03e4...2f0458dE9
0x5830c43fe33...	Add Package ...	12785771	50 days ago	0xe53b00C0...d79d8cCCb	IN	0x813E03e4...2f0458dE9

Figure 5.3: Event Transactions

The script also does one extra thing to helps us interact with the contracts. When deploying, there's an extra step called verifying the contract. This is optional and is only useful to be able to see the code in the explorer. What happens behind the scenes is the entire code is compiled and sent to the network to be stored, and the explorer has an option of associating it with the readable code. For this to happen, we need to prove that the code matches the compiled one and that we have permission to do so, by also proving that we own the private key of the address that deployed the contract. What this process essentially does is we can read the entire code and interact with it directly on the explorer. Not to forget that, not doing this process, the explorer still has the bytecode of the contract, so someone that knows how to decode it can still see the code, since everything in the blockchain is public.

What's more useful for us is to be easier to interact with it from the explorer directly and the Figure 5.4 shows a few of the functions that we can call on the smart contract, separated by the ones that are read-only and the ones that change its state, meaning the ones that have to be signed and payed for.

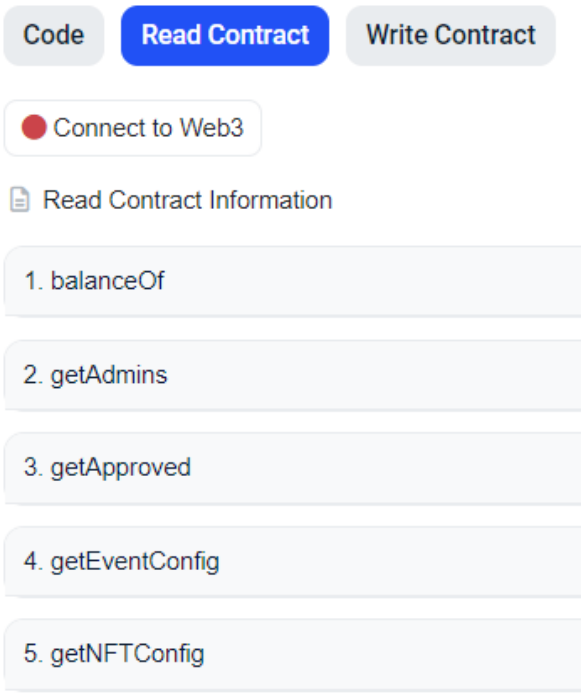

Figure 5.4: Event Read Functions

This is where we can get the details of the event, cause we made the getters for each of the structs that are important. As an example, we can check the packages configuration by calling the getter, and the Figure 5.5 shows the result of calling the getter of the package configuration, but in this case it shows as a tuple, because the explorer doesn't understand the custom structs made by us. Still, we can see its contents organized like shown in the Section **??**, showing the name, the description, the price (in wei, the smallest unit of ether), the supply, and the bool saying if the NFTs share the data.

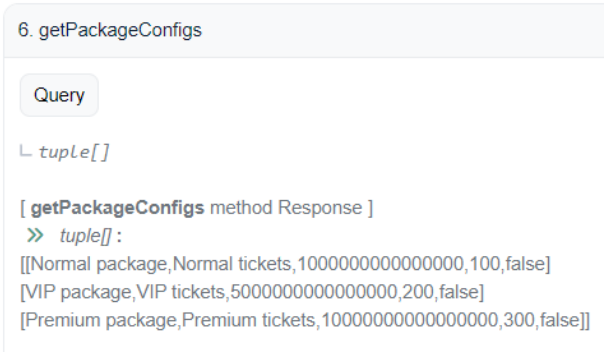

Figure 5.5: Package Config

The only difference from these read-only functions and the write ones is that you'll have to be connected to a wallet, set the necessary inputs and then the wallet will be

triggered for you to approve the transaction.

5.2 Mobile Application

5.3 NOTES [to remove]

[show application] [show smart contract interactions on the blockchain]

6

Conclusions

6.1 Limitations

[network fees] [finality] [users can still gift others in exchange for money, but we have to let users know they should only operate within the system]

6.2 Future Work

[marketplace (resell)] [organizer dashboard] [mention how wei can be translated to any currency later]