Department of Information Engineering and Computer Science

Ticketing Blockchain

Federico Sentineri - Marco Baricchio - Eddie Veronese

Blockchain - Accademic year 2023/2024

Abstract

The current online ticket purchasing system faces several issues based on user feedback, with concerns surrounding security, transparency, and ease of use. Our proposal aims to address these challenges by incorporating blockchain technology and tokenizing tickets to enhance the existing service. Additionally, we plan to introduce new features to differentiate our offering. This paper will present the technologies employed, the proposed architecture, and the development of the blockchain to ensure improved performance, along with a more transparent and efficient service management.

Contents

1 Introduction

The project aims to develop an online platform for buying and selling event tickets, leveraging the security and transparency provided by blockchain technology. The application will enable users to securely purchase tickets for events organized by a variety of promoters, integrating some innovative features. Among these is the possibility for users to purchase group tickets, allowing them to secure adjacent seats and enjoy the event together with friends. Additionally, the platform will offer a ticket resale system, allowing users to sell their tickets to third other customers while ensuring the authenticity and validity of the entry passes. A distinctive aspect of the project will be the fight against the phenomenon of ticket reselling, which unfortunately characterizes the current market where tickets are often purchased at original prices only to be resold at higher amounts, or where the same ticket can be sold to multiple buyers. Thanks to the use of blockchain and smart contracts, the system will provide a secure and transparent method to address these issues, ensuring a fair and fraud-free experience for all users.

2 Market Context

In analyzing the main competitors in the ticket buying and selling sector in Italy, we focused our research on the two leading service providers: TicketOne and TicketMaster. The platforms offered by these providers have features similar to those of our project, allowing users to access a wide list of events and purchase tickets for specific seats.

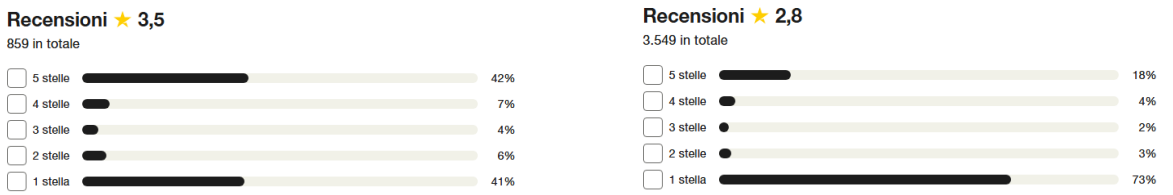
2.1 Problems

In this section, we will analyze the issues from both the users' and service providers' perspectives.

2.1.1 Problems with Current Ticket Providers from the Users

Based on the analysis of the services offered by TicketOne and TicketMaster, exploiting with reviews on TrustPilot, several issues have emerged regarding ticket management and customer support. The main concerns reported by users are as follows:

- **Problems with ticket issuance:** Many users report issues during the purchase process, particularly when payments are completed without the ticket being properly issued. This malfunction causes inconveniences and frustration for users who find themselves with a completed payment but without the purchased ticket.
- **Inadequate customer support:** The customer support service is often insufficient, with slow response times and difficulties in resolving ticket management issues, such as refund requests for expired tickets or the provision of detailed information about purchased tickets.
- **Difficulties in purchasing international tickets:** Problems also arise when purchasing tickets for events held outside the national territory. Users encounter obstacles such as payment method restrictions and challenges in managing international transactions, further complicating the purchasing experience.
- **Difficulties in group ticket purchases:** Buying multiple tickets, especially for highly demanded events, is problematic as the platform only allows the purchase of individual tickets. This makes it difficult to secure adjacent seats, forcing users to make repeated purchases in the hope of finding contiguous seats.
- **Slowness in the purchasing process:** The purchase process is often lengthy and complex, with the possibility of tickets being sold while the user is still in the validation phase, rendering the entire process futile.
- **Problems with payment methods:** Users, particularly tourists, encounter difficulties with the payment methods offered by the platforms, which are not always compatible with their preferences or international credit cards.



(a) TicketMaster feedbacks from users

(b) TicketOne feedbacks from users

Despite the numerous reported issues, service providers justify the large amount of negative feedback by claiming they are unaware of the described disruptions, allowing user dissatisfaction to persist.

2.1.2 Problems for the Providers

The seat selection and assignment option, although offering advantages in terms of customization, does not always yield optimal results. In particular, it is common for individual seats to remain unoccupied within various event sectors. This prevents users purchasing tickets in groups from selecting adjacent seats, even when the overall availability of the sector would permit it, limiting their ability to enjoy the event experience together with friends.

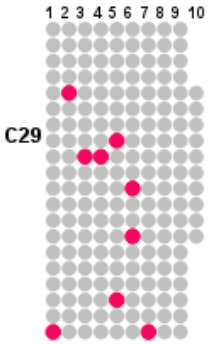


Figure 2: Example of a sector with unassigned seats

2.2 Our Solution

Our solution aims to address and resolve most of the previously described issues, with particular attention to simplifying, securing, and ensuring the transparency of the ticket purchasing process. Purchases are made directly through the blockchain, ensuring the immediate assignment of the ticket to the user at the time of the transaction. The ticket's traceability serves as a guarantee for the customer, who retains active control over ticket-related activities, such as name changes or the option to resell their ticket.
The seat assignment system has been completely redesigned to balance the needs of both users and providers. Users will be able to purchase tickets for a specific sector without selecting a specific seat, assuming that visual and auditory quality is roughly uniform within the same sector. Additionally, it will be possible to create user groups belonging to the same sector to ensure they are assigned adjacent seats during the event. The final seat assignment will take place a few hours before the concert, ensuring that members of the same group are seated together while optimizing sector occupancy to prevent the selection of separate seats from reducing overall seat availability in a given area.

3 Business Analysis

For the business analysis, we will examine both the Value Proposition Canvas and the "Why, When, How" approach.

3.1 Value Proposition Canvas

We analyze the Value Proposition Canvas of our project:

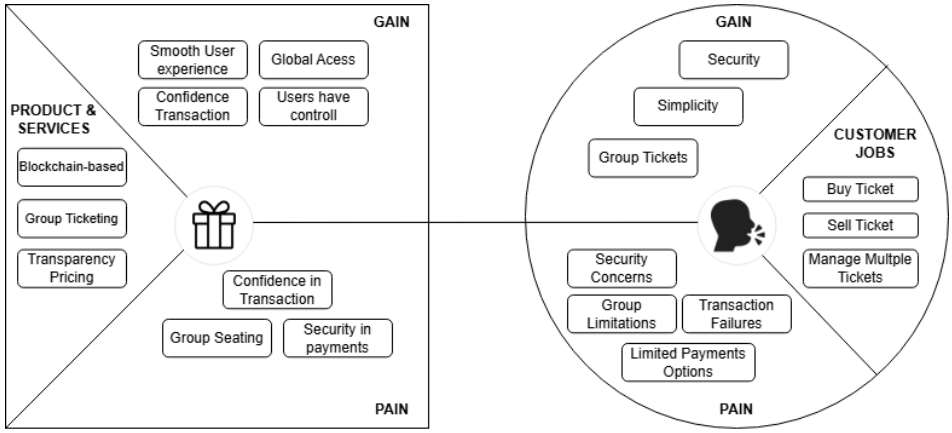


Figure 3: Value Proposition Canvas

Customer Jobs

The customer's primary tasks involve purchasing and managing tickets. Users must have access to their ticket information and the ability to modify it when needed. This includes reselling tickets, creating groups, or managing the associated names, allowing users to buy multiple tickets for friends and family.

Customer Pains

When purchasing tickets online, there is always uncertainty regarding payment security and transparency. As already highlighted in the Market Context section, current systems do not guarantee high efficiency in payment processing or post-purchase support services.

Customer Gains

The inclusion of blockchain technology would enhance payment security and provide a transparent transaction history. The purchasing process would become faster and less stressful for consumers. Additionally, the ability to create groups would streamline several manual operations currently unsupported by existing systems.

Products and Services

The application, leveraging blockchain technology, aims to strengthen the services currently offered by similar competitors. This is particularly true in terms of ticket acquisition, where it guarantees both transparency and security.

Product Gains

By improving its services, the application would deliver a superior user experience, expanding its target audience beyond domestic customers and potentially reaching international markets.

Product Pain Relievers

The application addresses user pain points by simplifying ticket management processes, providing enhanced payment security, and reducing operational inefficiencies.

3.2 Why, When, How

3.2.1 Why

The leading online ticket service providers are currently facing significant user dissatisfaction regarding their services. Customers have reported issues related to payment reliability and inadequate support for even basic operations, resulting in widespread dissatisfaction and a lack of trust in the service providers.

3.2.2 When

To successfully launch the project, two crucial elements must be established:

- **Participation from a substantial number of event organizers**: This would position the application as a central hub for users by consolidating a wide variety of events from multiple promoters.
- **A strong pre-launch marketing campaign**: This is essential to attract a significant customer base and ensure a steady flow of transactions on the blockchain.

Additionally, nothing prevents us from offering the application's features as a service to current providers such as TicketOne or TicketMaster. This approach could significantly reduce both development and launch timelines.

3.2.3 How

To participate in the application, event organizers must request authorization from the application owner to be added to the organizer set. Once approved, they will be assigned their dedicated NFT contract for event and ticket management. Organizers can then create events through the application, specifying the desired number of sections and associated tickets. Since each ticket is unique, minting will be carried out individually, with each ticket uniquely identified by a specific ID. Upon the creation of a ticket, it becomes available for purchase by customers. Upon completing the transaction, the customer becomes the owner of the NFT ticket.

4 Architecture

Our technology was developed and tested on Ganache, which enabled us to simulate an extensive blockchain environment with multiple users and perform rapid testing using Truffle. Below is a detailed overview of the key components of the architecture.

4.1 Frontend

The frontend was developed with the following technologies:

- **AngularJS**: Used to build a simple and intuitive user interface.
- **TypeScript**: Employed as the main interface to connect with the blockchain.
- **Ether.js**: A JavaScript library that enabled direct interaction with the Ethereum blockchain from TypeScript files.

4.2 Blockchain

For blockchain integration, we utilized the following tools:

- **Web3 and MetaMask**: Provided seamless connectivity to the Ethereum blockchain to access its resources.
- **Ganache**: Enabled testing of our smart contracts by simulating a local blockchain environment, ensuring efficient and reliable testing.
- **Truffle**: A development framework used for executing multiple testing scripts in a standardized manner to validate smart contract functionality.
- **Smart Contracts**: Developed using Remix Ethereum IDE and deployed to the Ganache blockchain for testing and verification.

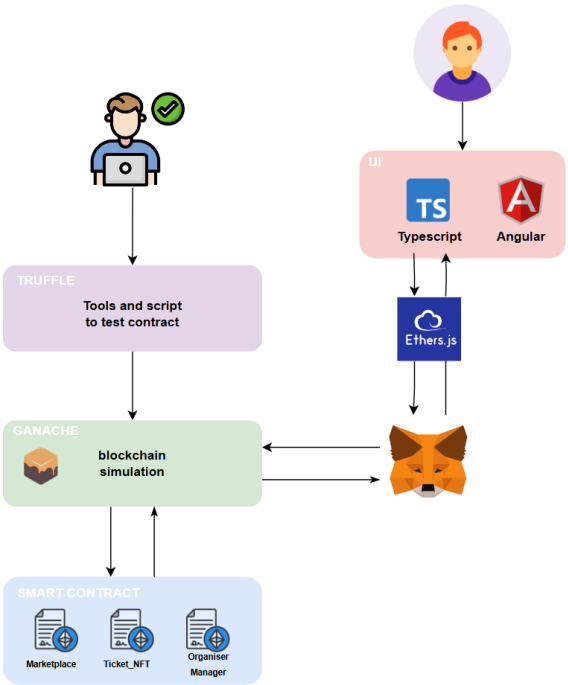


Figure 4: Architecture of the dApp

5 Smart Contracts

Let's analyze how the various contracts communicate with each other and the main functionalities of each contract.

5.1 Ticket_NFT.sol

The `Ticket_NFT.sol` contract encapsulates all the information related to events, sectors, and tickets associated with a single organizing entity. It serves as a comprehensive and structured repository for managing event-related data and ticket issuance within the system.

```
struct Event {
    uint256 id;
    string name;
    uint256 time;
    mapping(uint256 => Sector) sectors;
    uint256 sectorCount;
}

struct Sector {
    uint256 id;
    string name;
    uint256 totalSeats;
    uint256 availableSeats;
    uint256 seats_x_lines;
}

struct Ticket {
    uint256 id;
    uint256 eventId;
    uint256 sectorId;
    uint256 groupId;
    uint256 originalPrice;
    TicketStatus status;
    string seat;
    bytes32 hashName;
}
```

5.1.1 Role Management

To ensure secure and scalable permission management within our smart contract, we used the **Access-Control** contract from OpenZeppelin. This approach allows us to define distinct roles and associate them with specific addresses, providing fine-grained control over the operations each user can perform.

In our contract, the following roles were defined:

- **ORGANIZER_ROLE:** Represents the role of event organizers, such as Sony or Universal, who can manage event-related operations.
- **ADMIN_ROLE:** Holds full administrative powers over the contract, including the ability to assign or revoke the organizer role.
- **DEFAULT_ADMIN_ROLE:** The default administrative role, automatically assigned to the initial deployer of the contract (for instance, the development team).

Role Initialization During the deployment phase of the contract, the following operations are performed:

- **Assignment of the default administrator role:**

```
_grantRole(DEFAULT_ADMIN_ROLE, tx.origin);

```

  This assigns the `DEFAULT_ADMIN_ROLE` to the account that deploys the contract.

- **Assignment of roles to the organizer:**

```
_grantRole(ORGANIZER_ROLE, _organizer);
_grantRole(ADMIN_ROLE, _organizer);
_setRoleAdmin(ADMIN_ROLE, ADMIN_ROLE);

```

The organizer specified during deployment is assigned both the `ORGANIZER_ROLE` and `ADMIN_ROLE`. The latter is set as the administrator of itself, ensuring full control over administrative operations.

Dynamic Role Management After deployment, staff members can be dynamically added or removed as organizers based on operational needs:

- **Adding an Organizer:**

```
function addOrganizer(address _organizer) public onlyRole(ADMIN_ROLE) {
    _grantRole(ORGANIZER_ROLE, _organizer);
}

```

Only an account with the `ADMIN_ROLE` can add new organizers.

- **Removing an Organizer:**

```
function removeOrganizer(address _organizer) public onlyRole(ADMIN_ROLE) {
    _revokeRole(ORGANIZER_ROLE, _organizer);
}

```

The removal of organizers is also restricted to administrators.

The implementation of the role system using `AccessControl` allowed us to ensure secure, flexible, and scalable permission management. Sensitive operations are restricted to authorized users, facilitating the contract's governance.

5.1.2 Ticket Token Generation

The generation of tickets is handled individually using the ERC1155 standard. Each ticket is managed as a non-fungible token (NFT) because, upon validation, it will be treated as a unique asset. This uniqueness is ensured by the presence of both a nominative hash and a specific seat assignment. The creation process is implemented through the `_mint()` function, which allows us to issue a specific token representing the ticket. Below is the relevant code excerpt:

```
function createTicket(
    uint256 _eventId,
    uint256 _sectorId,
    uint256 _originalPrice
) external onlyRole(ORGANIZER_ROLE) {
    require(_originalPrice > 0, "Price must be greater than 0");
    Sector storage sector = events[_eventId].sectors[_sectorId];
    require(sector.availableSeats > 0, "No seats left");

    ticketCount++;
    uint256 ticketId = ticketCount;
    Ticket storage newTicket = tickets[ticketId];
    newTicket.id = ticketId;
    newTicket.eventId = _eventId;
    newTicket.sectorId = _sectorId;
    newTicket.status = TicketStatus.Available;
    newTicket.originalPrice = _originalPrice;
    newTicket.hashName = generateHash(organization_name, ticketId);
    sector.availableSeats--;

    _mint(msg.sender, ticketCount, 1, "");
}
```

This approach ensures a scalable and efficient tokenization mechanism for event ticketing, leveraging the flexibility of ERC1155 while treating each ticket as a unique asset due to its specific properties.

5.2 User Identification on Tickets - Privacy Management

During ticket validation, it is essential to verify the correspondence between the ticket and its rightful owner. However, storing sensitive user information directly on the blockchain would compromise the fundamental principle of privacy inherent to decentralized systems. To ensure a secure and privacy-compliant validation process, we decided to store a hashed version of the ticket owner's name within the ticket metadata. The hash is generated by concatenating the owner's full name (name and surname) with the ticket ID, ensuring both uniqueness and anonymity. The hashing process is implemented using the `keccak256` algorithm, as shown below:

```
function generateHash(string memory str, uint256 num) public pure returns (bytes32) {
    return keccak256(abi.encodePacked(str, num));
}
```

This solution ensures that sensitive information is never stored directly on the blockchain while still allowing for secure and verifiable ticket validation.

Dynamic Owner Name Update

As an additional feature, we have enabled the ticket owner or an authorized organizer to update the ticket's associated name. This is particularly useful in cases where tickets are transferred to another individual or if user information changes. The following function allows for the secure modification of the ticket owner's hash:

```
function setOwnerName(
    uint256 ticketId,
    string memory name,
    string memory surname
) external {
    require(hasRole(ORGANIZER_ROLE, msg.sender) || balanceOf(msg.sender, ticketId) >
    0, "Not the ticket owner");
    Ticket storage ticket = tickets[ticketId];
    ticket.hashName = generateHash(string(abi.encodePacked(name, "-", surname)),
    ticketId);
}
```

Security and Access Control

The function includes a security check to ensure that only the ticket owner or an authorized organizer can modify the ticket information. This condition is enforced by verifying that the caller either holds the `ORGANIZER_ROLE` or owns the specific ticket through the `balanceOf()` function.

Privacy and Security Considerations

The use of a hash-based identification system ensures compliance with privacy requirements by preventing the exposure of sensitive user data on-chain. At the same time, the concatenation of the user's full name and ticket ID guarantees uniqueness and reduces the risk of collisions in the hashing process. This approach balances privacy, security, and flexibility, aligning with the core principles of decentralized systems and ensuring that user information remains protected throughout the ticketing process.

5.3 Organiser_manager.sol

The `Organizer_Manager` contract is responsible for managing the organizers, who are the owners of individual companies that handle event operations. This contract provides a comprehensive mapping of all organizers and their respective ticketing smart contracts, ensuring an efficient and scalable system for ticket issuance and event management.

Dynamic Contract Creation

One of the key functionalities of the `Organizer_Manager` contract is its ability to dynamically deploy new instances of the `Ticket_NFT` contract for each organizer. This dynamic deployment is accomplished through the following line:

```
Ticket_NFT nft = new Ticket_NFT(_name, _organizer_address);
```

This operation creates a dedicated and isolated instance of the `Ticket_NFT` contract for the organizer, which is then mapped to the organizer's address in the `organization_NFT` mapping.

Parameter Details:

- `_name`: The name of the organization or company that owns the ticketing contract.
- `_organizer_address`: The address of the event organizer, who becomes the owner of the newly created ticket contract.

The dynamic creation of ticket contracts ensures that:

- Each organizer has a unique and independent ticketing contract.
- There is no data overlap or conflict between different organizers.
- Smart contracts can be efficiently created, deployed, and mapped without requiring manual contract deployments.

Core Functions of the Contract

The contract provides the following essential functions:

- `create_organization(string memory _name, address _organizer_address)`: Deploys a new instance of `Ticket_NFT`, maps it to the organizer, and emits an event to record the creation.
- `getTicketNFT(address _organizer_address)`: Returns the address of the ticket contract associated with a given organizer.
- `getOrganizerNFT(address _organizer)`: Retrieves the ticket contract address for a specific organizer.
- `getOrganizationAddress(uint _id)`: Returns the address of an organizer given its unique identifier.

Security and Access Control

The contract employs access control through the `AccessControl` mechanism. Only the administrator, holding the `DEFAULT_ADMIN_ROLE`, can create new organizations and deploy their respective contracts. This ensures controlled and secure management of event organizers and their ticketing operations.

Scalability and Extensibility

By adopting a dynamic contract creation strategy, the system is highly scalable, allowing new organizers to be seamlessly onboarded without any disruption to existing operations. This architecture also enhances modularity, as each ticket contract operates independently while remaining linked to the main management contract. This approach represents an efficient and forward-thinking design for decentralized event and ticket management.

5.4 Marketplace.sol

The contract related to the marketplace tracks all tickets currently on sale on the platform. The main information for each ticket includes the ticket identifier and the address of the contract that holds it (i.e., the contract of the organization managing the event to which the ticket belongs). Each ticket is stored in a custom structure, which allows for quick access to information without the need to search through all event organizers' contracts for tickets on sale.

```
struct Listing {
    uint256 ticketId;
    address seller;
    uint256 price;
    TicketStatus status;
    address ticketContract; // ticket contract added to the structure
}

mapping(uint256 => Listing) public listings;
uint256 public listingCount;
```

The purpose of the marketplace is to manage all the information and logic related to ticket buying and selling operations, particularly actions such as `buyTicket(uint256 listingId, string memory name, string memory surname)` which allows for transferring a ticket to a new owner. The essential elements of the method involve several checks and fundamental functions, including:

- `Balance Verification`: The first crucial step is to ensure that the buyer possesses sufficient funds to purchase the NFT ticket. If this condition is met, the transaction is carried out using the `call` function, a low-level call to transfer Ether to the `listing.seller` address, returning a boolean value `success` to indicate whether the transfer was successful.

```
require(msg.value >= listing.price, "Insufficient payment");
(bool success, ) = listing.seller.call{value: msg.value}("");
require(success, "Payment failed");
```

- `safeTransferFrom`: The ticket's contract is accessed using the `Ticket_NFT(listing.ticketContract)` function. Since the contract inherits from `ERC1155Supply`, the `.safeTransferFrom(listing.seller, msg.sender, listing.ticketId, 1, "")` method can be invoked, allowing the transfer from the seller to the buyer.

```
Ticket_NFT(listing.ticketContract).safeTransferFrom(listing.seller, msg.sender,
    listing.ticketId, 1, "");
```

- `Ticket Value Updates`: After the transfer is completed, the ticket fields are updated, including setting the new owner's name and marking the ticket as owned and no longer available for sale.

```
Ticket_NFT(listing.ticketContract).setOwnerName(listing.ticketId, name, surname);
Ticket_NFT(listing.ticketContract).updateTicketStatus(listing.ticketId, Ticket_NFT
    .TicketStatus.Owned);
listing.status = TicketStatus.Owned;
```

6 Testing

6.1 Setup

To test the smart contracts created and the interactions between them we used two tools: Ganache and Truffle.

Ganache is a personal blockchain that simulates a local network on which it is possible to deploy and test contracts, without the need to use a public network; the application offers a GUI that allows you to view the status of your accounts in detail, all transactions and events issued. In order for the tests to be stable between each other, during the creation of the workspace, we used a fixed mnemonic, so as to always have the same account addresses in case of redistribution. The main GUI pages we used were:

- Account: allows you to monitor 10 pre-funded accounts with 100 ETH, useful for carrying out transactions
- Transactions: allows you to verify the transactions carried out, the gas used, the events issued and the status of the accounts and any addresses of contracts created

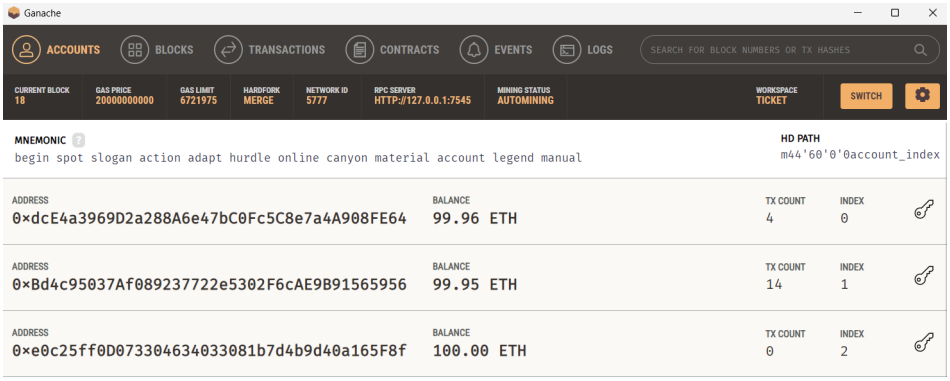


Figure 5: Ganache GUI

Truffe is a framework for the creation of blockchain applications that simplifies the writing, deployment and testing phase, in particular it is organized into various elements including:

- Contracts folder: containing the .sol files of the smart contracts, in our case three
- Migrations folder: contains the script for deploying contracts on the blockchain
- Test folder: containing the automated tests to verify the behavior of the contracts (one for each contract for a total of approximately 30 tests)
- Truffle-config.js file: allows you to set the environment for deployment and testing

In our case the Truffle environment was connected to Ganache by setting the localhost for deployment in the configuration file: this allowed us to run the tests locally multiple times without problems with funds or account generation.

```
development: {
      host: "127.0.0.1",     // Localhost (default: none)
      port: 7545,            // Standard Ethereum port (default: none)
      network_id: "*",       // Any network (default: none)
     },
```

6.2 Test Files

Three test files were used to test the system, each dedicated to a smart contract, but all designed to interact with each other.

6.2.1 01_test_organizer_manager

This contract validates the functionality of the Organizer_manager.sol contract, i.e. it manages the registration and control of organizers:

- creation of organizations: Verifies the correct creation of organizations and the assignment of NFT associate addresses
- permissions: check that only the admin can add new organizers
- information recovery: ensures that the information of the organizers' NFT contracts can be recovered
- edge cases: Prevents the creation of duplicate or unauthorized organizations

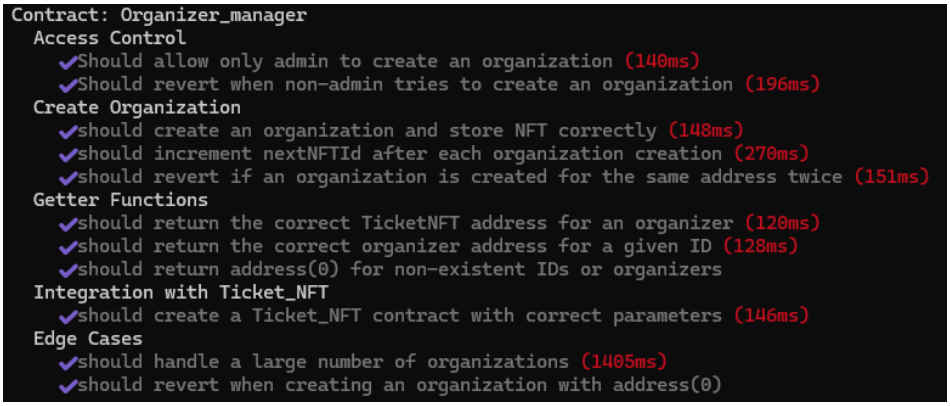


Figure 6: Test examples for Organizer_manager.sol

6.2.2 02_test_ticket

This file verifies the main functionalities of the Ticket_NFT.sol contract, which deals with managing events, sectors and tickets:

- deployment and roles: check the assignment of the admin and organizer roles within the NFT contract and verify that unauthorized users cannot act on it
- event and sector management: Tests the creation of events and sectors, ensuring that only organizers can perform these operations
- tickets: Check the creation and management of tickets, ensuring that only organizers can create tokens
- edge cases: Prevents the creation of zero-price tickets and monitors their status

6.2.3 03_marketplace

Check the management of the market for the purchase and sale of tickets via the Marketplace.sol contract:

- ticket sales: checks that only the owner can put a ticket on sale and checks that its internal status is updated
- purchase: Tests the purchase of tickets, ensuring correct transfer, sufficiency of funds and payment management
- removal: only allows the owner to remove a ticket from sale
- edge cases: ensures that tickets that have already been sold cannot be put up for sale again.

7 Frontend Usage

The front end of the project was created using Angular, a framework that allows you to create dynamic pages in a fluid way.

7.1 Project structure

The angular project consists of various folders and elements:

- Components: contains the Angular components, each made up of 4 files: .html, .ts, .css and .spec.ts; there are multiple ones in the project, one for each page of the graphic interface
- Services: contains the functions and logic for communications with the blockchain, there are 4 services: one for the connection with metamask, and 3 others, one for each deployed contract (Factory, Ticket_NFT and Marketplace)
- Utils: contains a static class to pass event data between the various components and a formatter for the date and time
- Classes: contains the classes necessary to represent the information of events, sectors and tickets to represent them in the various functions

7.2 Flow of requests

The flow of functions for interacting with the blockchain is divided into several distinct phases. Initially, the connection to metamask is requested via the connect function of the "Connection Service", in this way the information on the wallet address that will be used to sign the operations is obtained and saved.

```
constructor() {
  this.provider = new ethers.BrowserProvider((window as any).ethereum);

  (window as any).ethereum.on('accountsChanged', async (accounts: string[]) => {
    console.log('Account changed:', accounts[0]); // Log the new account
    await this.connect(); // Reconnect to the new account
  });
}

async connect() {
  // Request account access from MetaMask
  await this.provider.send("eth_requestAccounts", []);
  this.signer =  this.provider.getSigner();

}
```

Figure 7: Connection to Metamask

Subsequently, when a button is pressed to perform an operation from the graphical interface, the necessary information is taken from the .html file and passed to the respective .ts file of the component. The latter calls a function in the service with the specific data. Within the service, the contract instance is created via the ethers library, which uses the contract's ABI, its deployment address on the blockchain and the connection provider, in our case metamask; this instance is used to call a contract function and through the receipt the output data is returned to the component

```
constructor(private connection: ConnectionService) {

  this.contract = new ethers.Contract(
      contract_factory_Address,
      contract_factory_ABI,
      connection.getProvider() );
}
```

Figure 8: Creation of contract

```
async createEvent(name: string, time: number): Promise<number> {
    if(!this.contract){
        throw new Error('Contract not initialized');
    }

    const tx = await this.contract['createEvent'](name, time);
    const receipt = await tx.wait();

    console.log("Receipt", receipt);
    const eventId = receipt.logs[0].args[0].toString();

    console.log(`Event created with ID: ${eventId}`);
    return parseInt(eventId);
}
```

Figure 9: Function to create an event

8 Future Improvements

8.1 Future Deploy

Currently, the smart contracts have only been deployed locally on the Ganache network. Once the project is completed, the plan is to deploy it on the Polygon network, which, unlike Ethereum, offers faster transactions and lower gas costs due to congestion optimization.

8.2 Database Inclusion

To enhance the quality of service, an optimal choice would be to include a database for storing supplementary or non-essential metadata related to the blockchain. The inclusion of a database such as **Pinata** (a storage platform tailored for decentralized applications) would enable the saving of immutable and non-critical information associated with events, such as event dates and images. By offloading non-essential information, we would improve both the performance and reduce transaction costs on the blockchain.

8.3 Concurrency Management

To prevent multiple users from attempting to purchase the same ticket, a feature will be added to temporarily lock a ticket for a specific amount of time. During this period, the customer must complete the transaction before the ticket becomes available for others to purchase.

8.4 Platform Earnings

Currently, our application does not provide earnings for the platform owner, which is an important feature to implement before the application's launch. We are considering a fixed transaction fee, similar to current competitors, which could be around $2 per ticket. Of course, the conversion to the payment token (Ethereum in our case) will follow the current market exchange rate for the currency.

8.5 Refund System

A common issue with existing applications is poor management of refunds in the case of canceled events. A future feature will be the automatic refund of all users holding tickets for a canceled event, along with the necessary logic to manage such refunds.

8.6 Ticket Exchange

A similar option to refunds could be the possibility of exchanging a ticket for an event with one of equal or lesser value for a subsequent event, in the event that no one purchases the user's ticket. This would provide users with greater peace of mind when making purchases on the platform, as they would have the guarantee of being able to switch events. To prevent misuse of this service, a small fee would be applied for each ticket exchange.

References

[1] Angular. *Documentation Angular usage*. URL: https://angular.dev/overview.

[2] openzeppelin. *Documentation Access Controll usage*. URL: https://docs.openzeppelin.com/contracts/3.x/access-control.

[3] openzeppelin. *Documentation ERC1155Supply usage*. URL: https://docs.openzeppelin.com/contracts/4.x/api/token/erc1155.

[4] soliditylang. *soliditylang Documentation usage*. URL: https://docs.soliditylang.org/en/latest/contracts.html#abstract-contracts.

[5] ticketmaster. *ticketmaster website*. URL: https://www.ticketmaster.it/.

[6] ticketone. *ticketmaster website*. URL: https://www.ticketone.it/artist/olly/olly-tutta-vita-tour-2025-2026-3782087/.

[7] TruffleSuite. *Documentation Ganache usage*. URL: https://archive.trufflesuite.com/docs/ganache/.

[8] TruffleSuite. *Documentation Truffle usage*. URL: https://archive.trufflesuite.com/docs/truffle/.

[9] trustpilot. *feedback about ticketone/ticketmaster*. URL: https://it.trustpilot.com/review/www.ticketone.it.