

Peer to Peer Systems and Blockchains Academic Year 2021/2022 Project Work Web App of TRY: a nfT lotteRY

Calogero Turco

Update: August 29, 2022

1 Project Work components: Ethers, React,..

This project work is a web app based on an NFT Lottery contract. The web app was developed using Javascript with the React library for the front-end components and the ethers.js library for the back-end interaction with the blockchain[1]. To connect a wallet with the web app the Metamask browser extension, Metamask allows the user(manager or player) to pay the necessary fees for executing a contract function call. Metamask itself acts as a provider to the ether.js library, which means that it interacts with the blockchain, reading it and forwarding function calls. The web app was tested on a Ganache blockchain locally deployed.

Figure 1 shows the architecture of the web app and its connection to the Ganache blockchain. The main components of the ethers.js library are:

- **provider**: object providing a connection and an interface to the Ethereum test net, allowing to read events, logs, and information about the current state of the blockchain, such as the last mined block number. The provider that is specified in the provider creation is "window.ethereum" which connects to Metamask [2].
- **interface**: this object allows to encode and decode the bytecode data read from the blockchain to the Javascript higher level language, to do so it uses the ABI (Application Binary Interface) of the smart contract, which specifies the encoding of data for the smart contract. The ABI can be obtained from the Remix IDE.
- **contract**: it is an object that connects to a contract already deployed on the blockchain and allows interaction with the contract via function calls. A contract object instantiated without a signer can only execute view function calls.

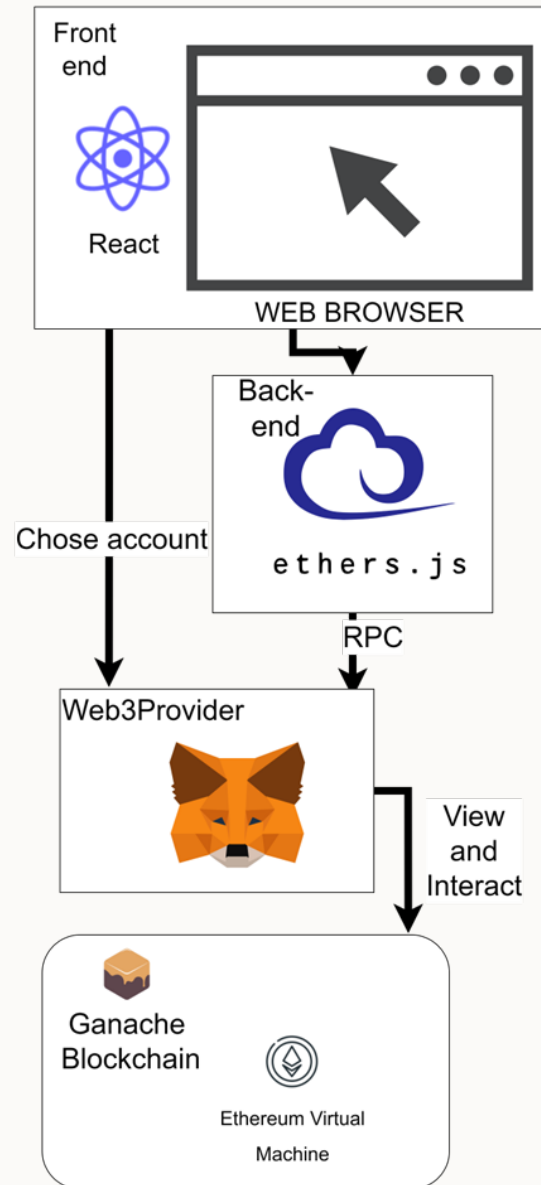


Figure 1: Image showing the different layers of the web app

- **signer**: this object, obtained from a provider object method, allows instantiating a contract object that can perform function calls that write on the blockchain.
- **contractFactory**: object that given an ABI, a contract bytecode, and a signer object, allows deploying a new contract on the blockchain.

Regarding the project directory, the file **App.js** in the **src** directory contains the majority of the web app code of the project, with the functions that use the ether.js library in it. In the sub-directory **components**, there is the code of some GUI components that take the user inputs, such as the **NumberBox.js** file that implements the component that takes the user numbers to play in a ticket and that can be seen in figure 11. The project directory also includes the contract ABI and the contract bytecode, allowing the manager to access them to deploy a new Lottery contract.

2 How to run webapp

The requirements to run the web app are:

- **NodeJS** containing npm installed.
- **Ganache**

Before running the web app it is necessary to download the required React modules with the terminal command:

```
npm install
```

The web app can be run in two ways, running a production build or a developmental build. To create a production build terminal, make the KittiesWebApp directory the current working directory, and use the commands:

```
npm run build  
npx serve -s build
```

the web app can then be opened in localhost:3000 or at the address specified by the command in the terminal. Alternatively, the web app can be started as a development build with the command:

```
npm start
```

Once the web app has started, the Metamask extension must be connected to the Ganache test network, this can be done by adding a network to Metamask with the RPC SERVER address specified by Ganache, which can be seen in figure 2. Also to perform pure function calls, that write on the blockchain, it is necessary to have an account with a balance on Metamask. To do as in figure 3 an Account can be imported from Ganache, by using a predefined account private key.

3 GUI and implementation

The web app is divided into two different sections, one for the manager of a Lottery contract and one for the player.

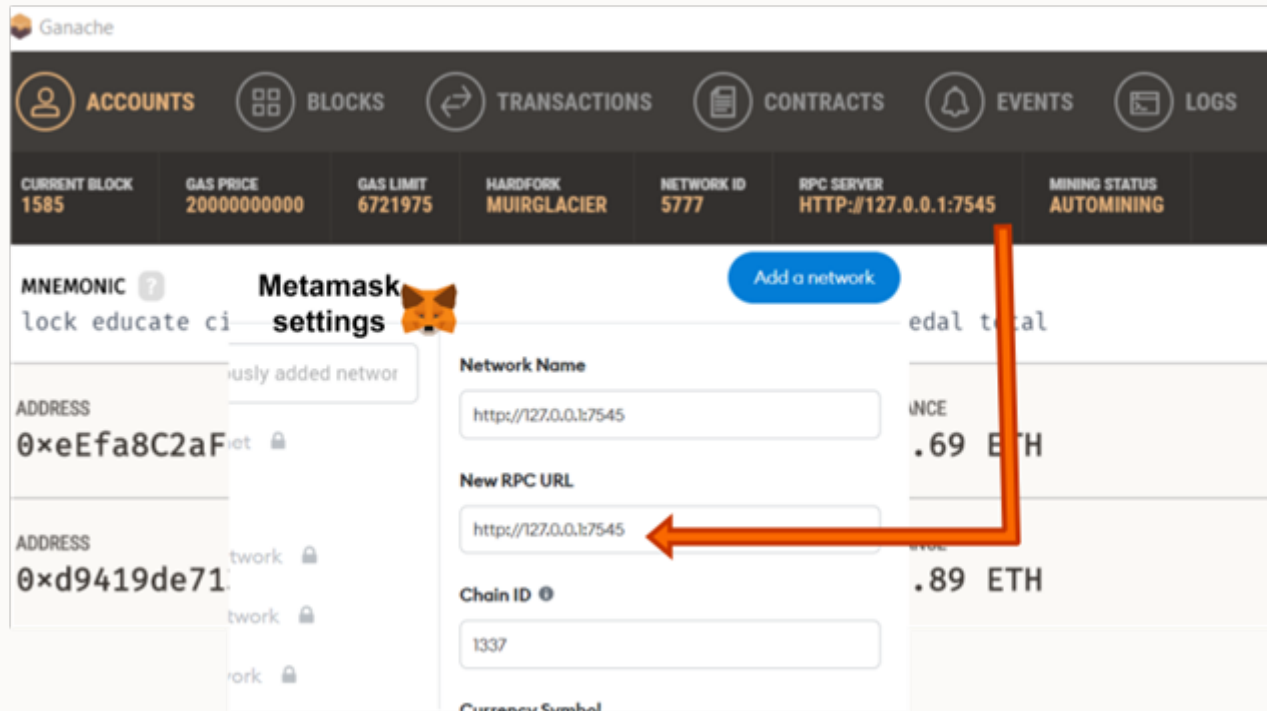


Figure 2: Image showing how to add a Ganache network to Metamask

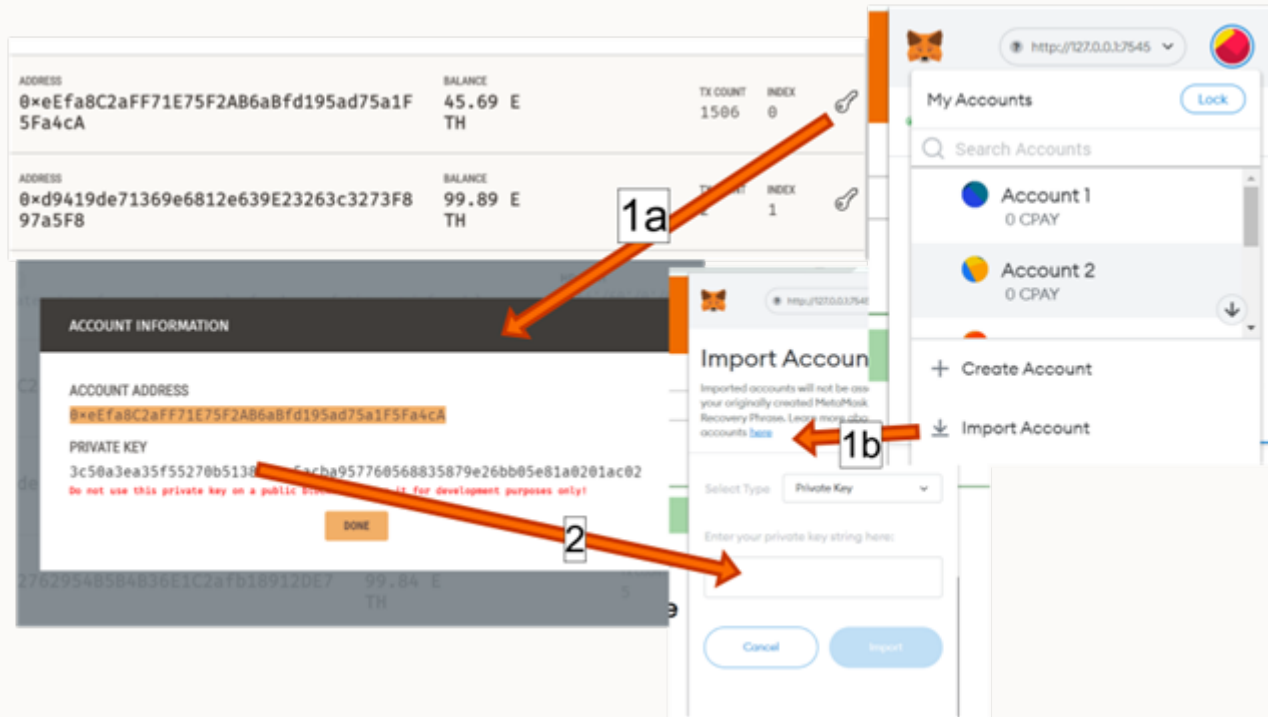


Figure 3: Figure showing how to import an account from a Ganache network to Metamask

3.1 Manager

As seen in figure 4 the Manager has to connect its wallet account to the web app before creating a new contract, the method `connectAccountHandler` triggers the Metamask browser extension to prompt the manager to select an account.

After connecting an account, the manager can decide to manage an existing contract created on previous

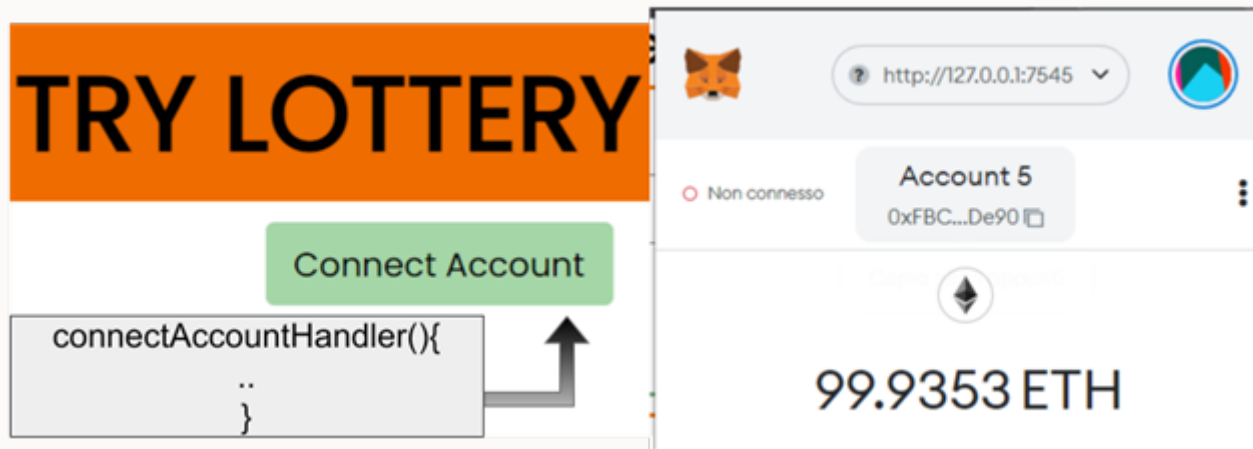


Figure 4: GUI of manager with the button to connect its Metamask account to the web app, the grey rectangle shows the Javascript method executed when clicking the Connect Account button

occasions or to create a new contract, it can be seen in figure 5. When inserting the address of an existing contract it is checked, in the method `ManageExistingLotteryContract()`, if the contract at the address provided

exists and if the account connected to the web app is the owner of the contract. The creation of the contract then instead uses a `ContractFactory` object to deploy the contract, with the `ContractCreate()` method.

Before the start of a round, the GUI is in the state of figure 6, the button `Start New Round!!!` allows starting a new round. The button triggers the Javascript method `startNewRound()`, which executes the instruction `await contract.startNewRound()`, this instruction returns a Javascript promise. A promise object will contain the result of the function call when it becomes available. To get the result of the promise, it is possible to call its `wait()` method. After obtaining a result from the function call, which consists of transaction logs, those are decoded using the contract interface, this allows the web app to read and display the event triggered by the function or the function return values. The `roundStarted` event of the contract `startNewRound()` function is decoded and used to display the block in which the round starts.

After creating the lottery, the manager can close the lottery at any moment, by pressing the dedicated button. The manager can see all the kitties minted and available to give as prizes with the button `Show rewards Kitties`. The GUI state after a round has started can be seen in figure 7, the manager can draw numbers when the current block number is equal to the round starting block number plus the round duration. After pressing the `show rewards Kitties` button, the manager can also mint a new kitty of a specified class.

After numbers are drawn, the manager can see them in the GUI. As in figure 8, the manager can also select the address to which to send the fees collected by the contract during the round, as can be seen in figure 8, also the manager can send the fees directly to the current account address used to manage the Lottery.

TRY LOTTERY

Manage existing Lottery contract

Contact Address

Type Contract Address

Enter Contract Address

manageExistingLottery(){
check if created, round, prizes
}

Create new Lottery contract

Set fee

fee value (wei)

Set round duration

round duration (blocks)

Create Lottery!!!

ContractCreate(){
}

Account Address: 0xeefa8c2aff71e75f2ab6abfd195ad75a1f5fa4ca

Figure 5: GUI of manager to manage an existing contract or set fees and round duration for a new contract

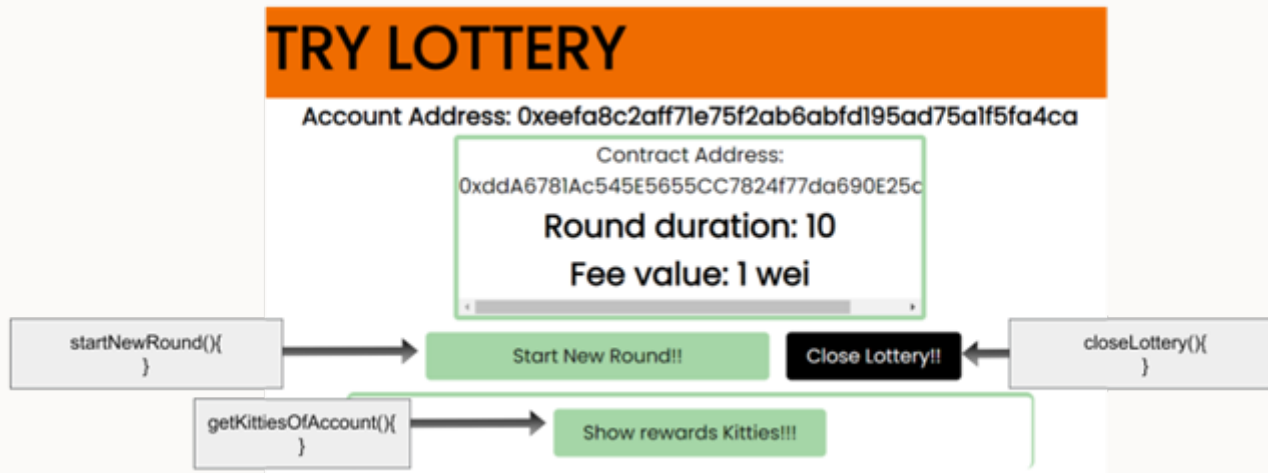


Figure 6: GUI of a manager to start a new round, close lottery, and see the minted kitties not given as prizes yet

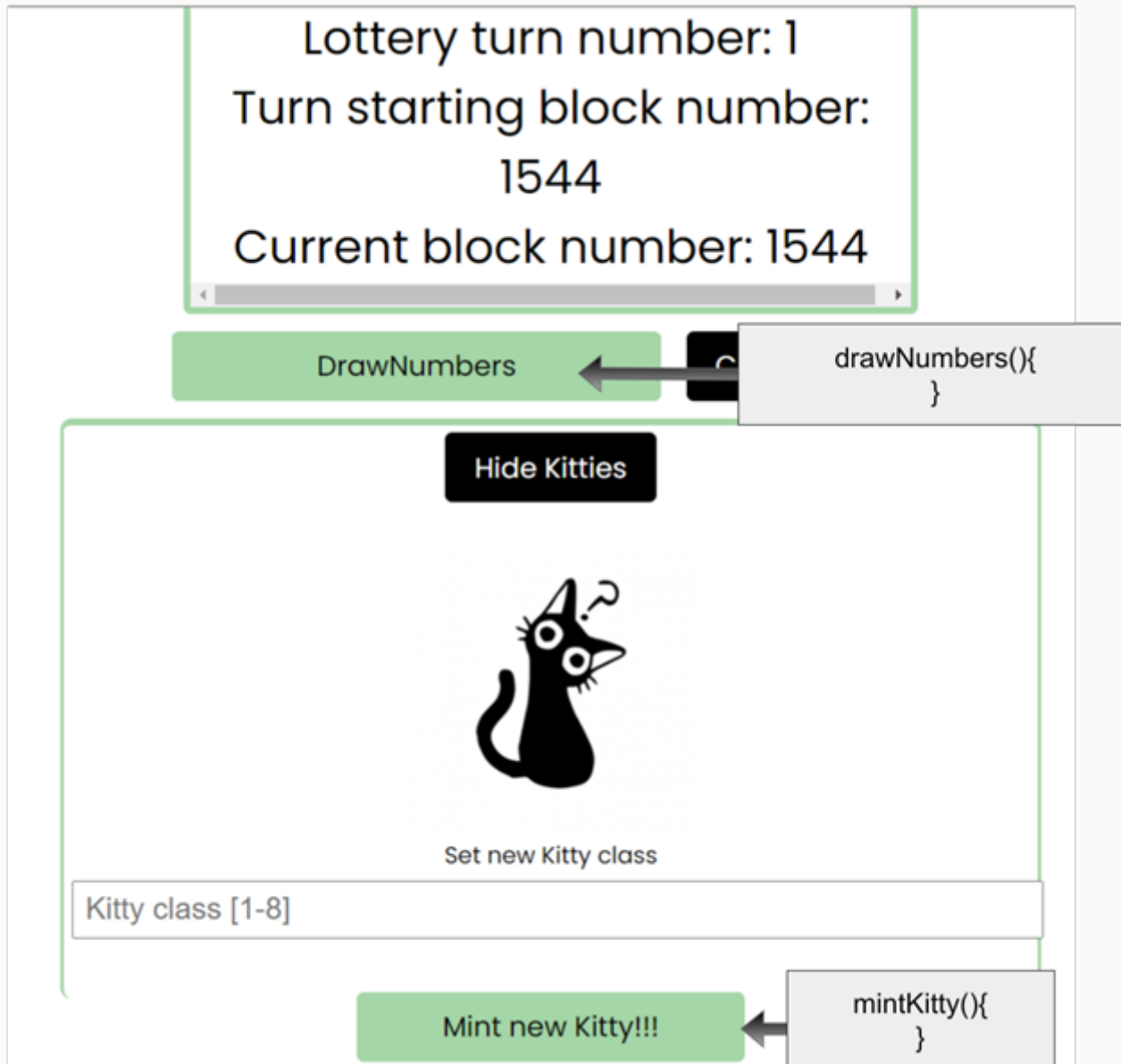


Figure 7: GUI of a manager to draw numbers and mint a kitty¹²

Current block number: 1548

23,41,2,19,2,23

Give Prizes and collect fees to current Account address

Custom account to send fees

Type account to send collected fees

Enter Account address

Close Lottery!!

givePrizes(){
}

givePrizesToSelectedAddress()
{
}

Figure 8: GUI of a manager to give prizes and collect fees

3.2 Player

A player can connect to a lottery contract if it knows its address in the blockchain by typing it. All the Lottery contracts deployed are presented in a list (as seen in figure 9), this list is updated constantly to **notify the user on new contracts created**. The list was created with the method `findLotteryCreated()` by reading the logs of the blockchain and finding all the `createdContract` events triggered, the list also constantly updates when a new contract is created, as the `checkNewLotteryCreated()` method registers a listener, using the provider object that activates its code each time the event `createdContract` is triggered.

To play with the desired Lottery the player must insert a valid contract address and connects connect an account to the web app as prompted in figure 10. When connected to the contract, the player can see on the GUI information about the lottery status, if there is an active round then the user can buy a ticket by inserting the numbers it wants to play or use the generate numbers button that can be seen in figure 11 and then pressing the Play lottery button. The player can see the list of kitties received as prizes by pressing the Show Kitties!!! button.

A player could also transfer the ownership of one of its kitties to another account (in figure 12).

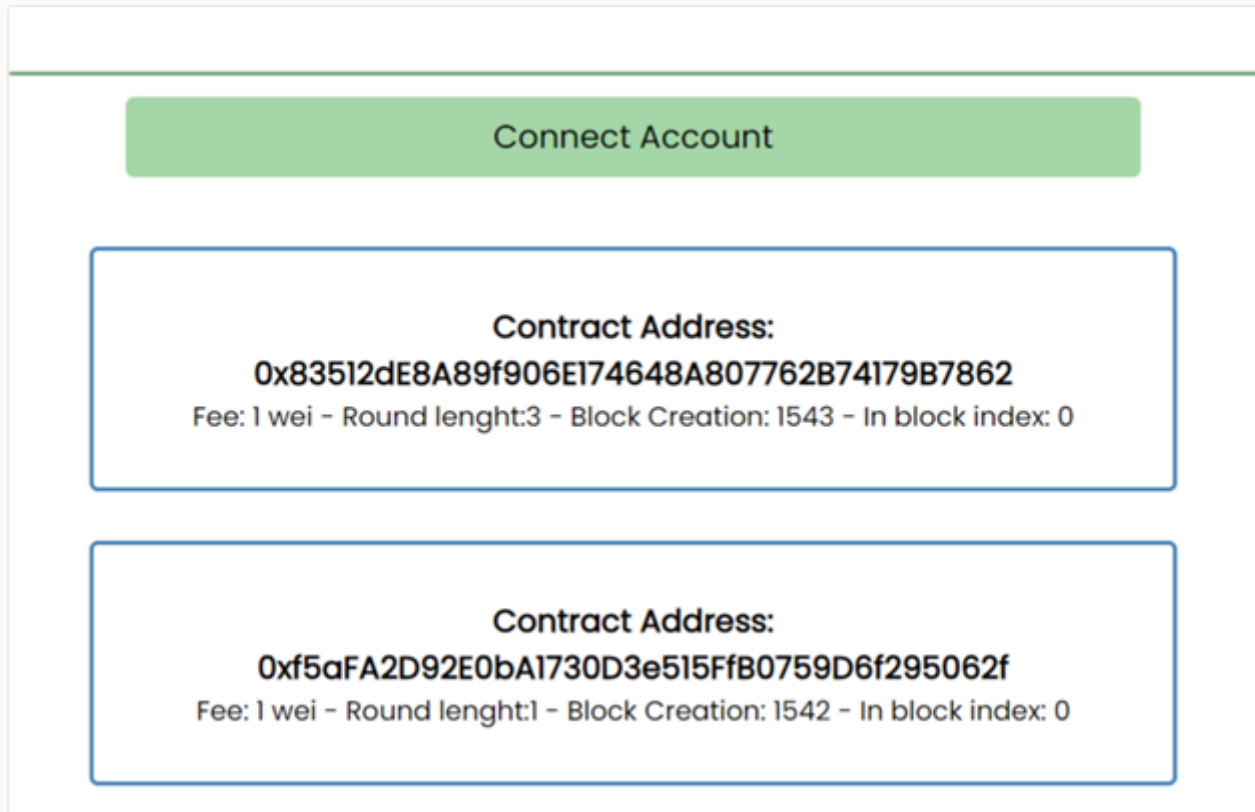


Figure 9: GUI of player displaying the Lottery contract that is on the blockchain

TRY LOTTERY

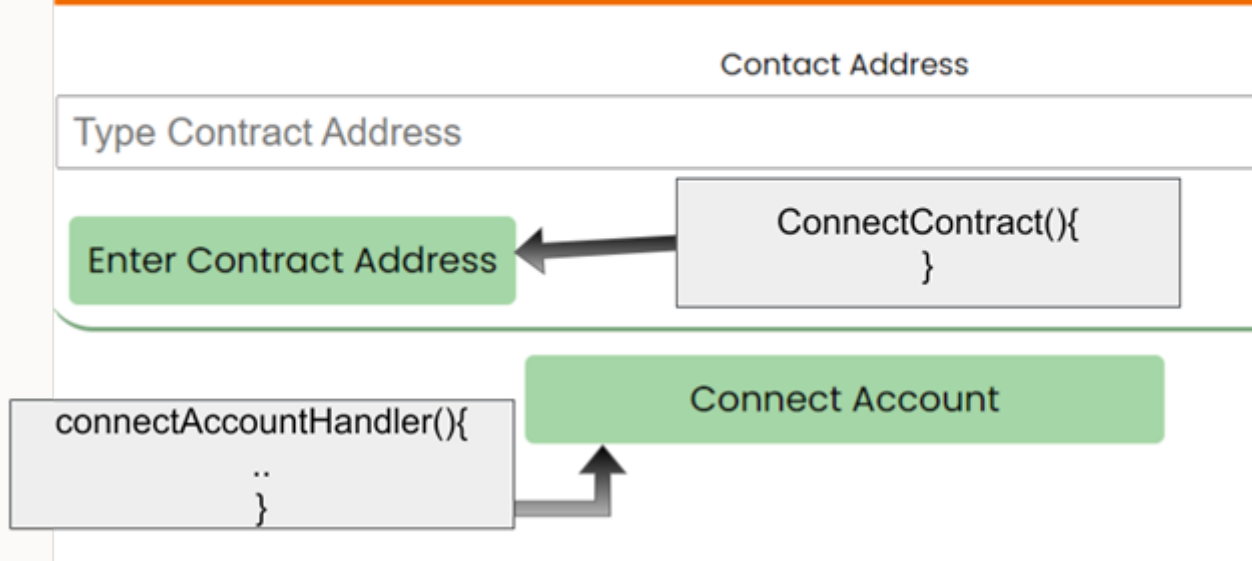


Figure 10: GUI of player to connect account and to connect to a Lottery contract

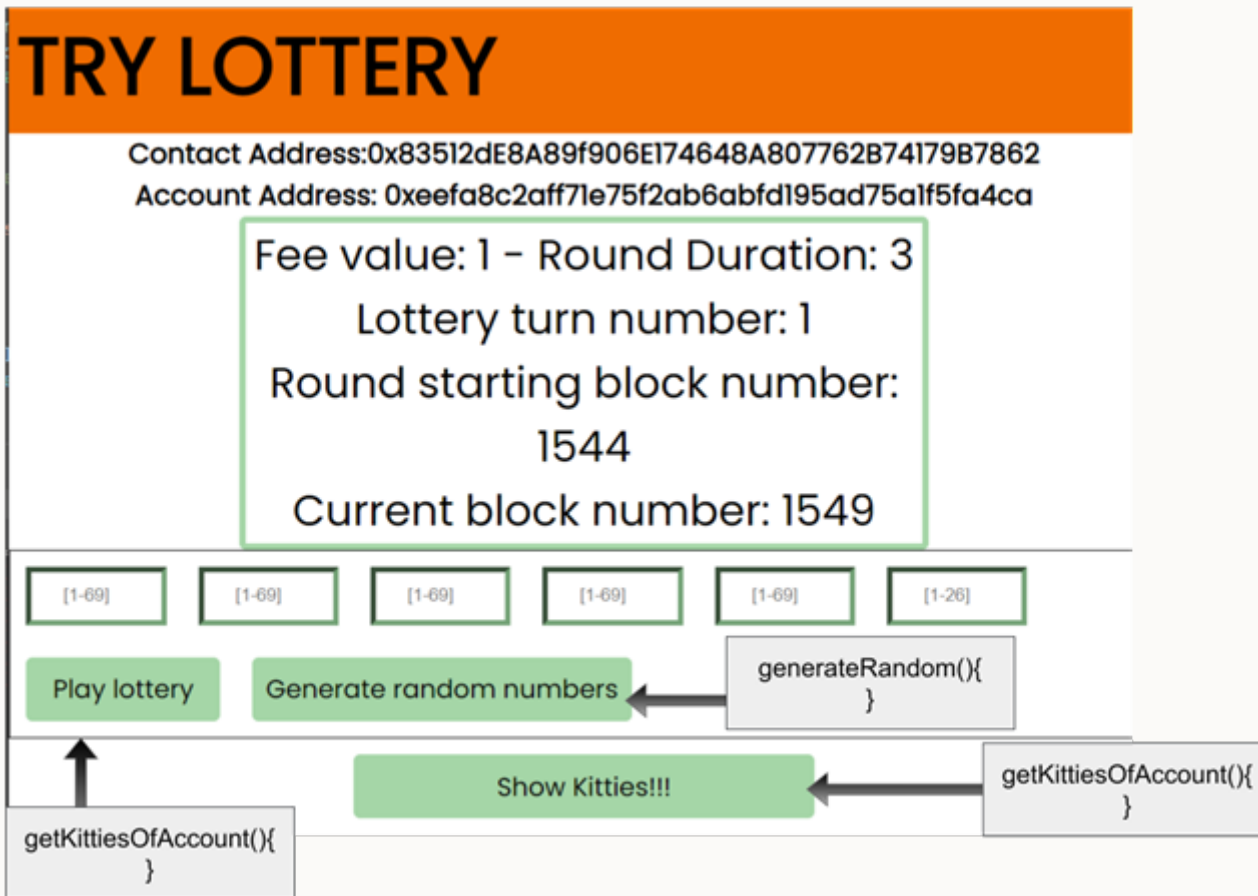


Figure 11: GUI of player to buy tickets or see the list of owned ones

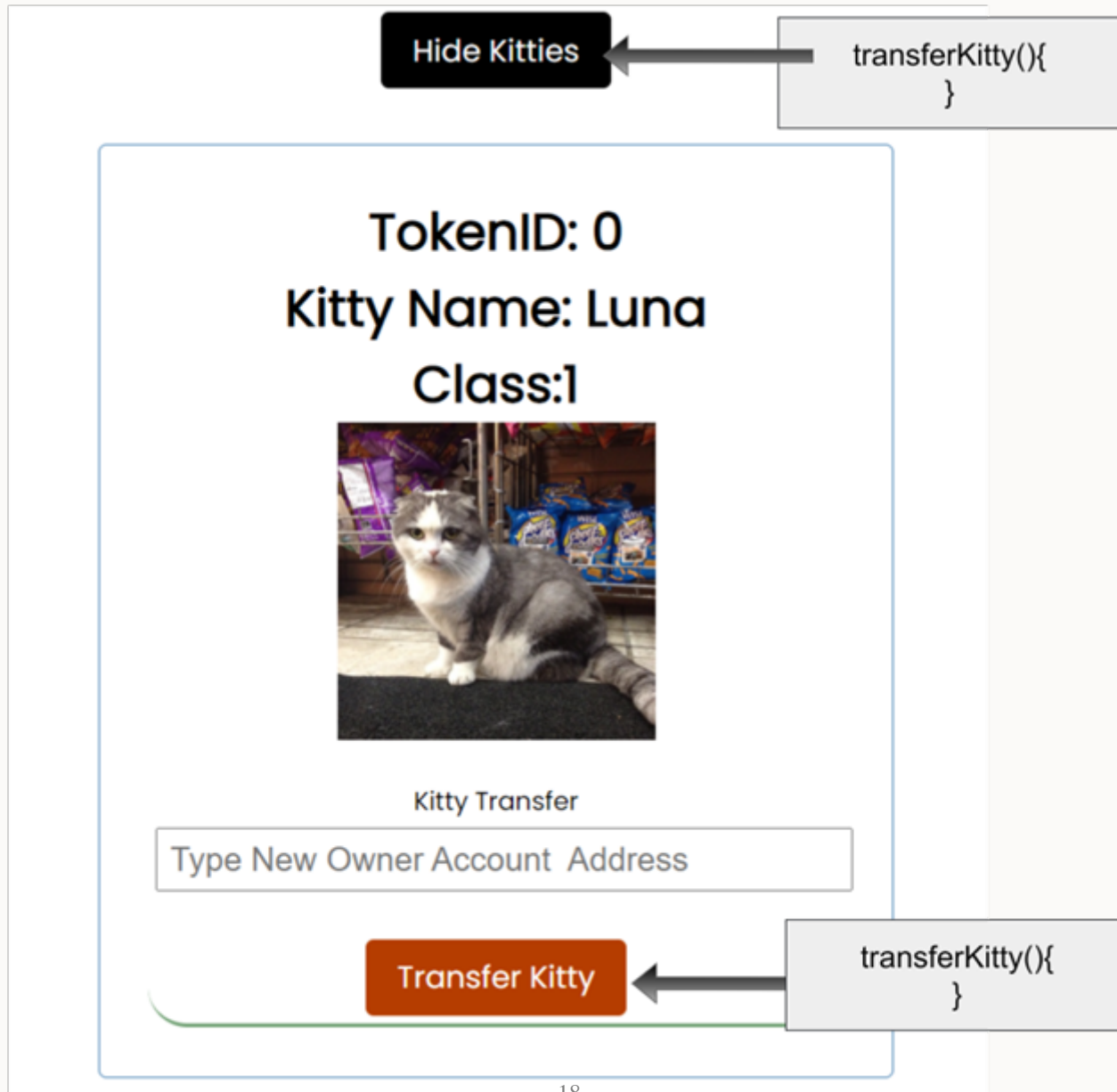


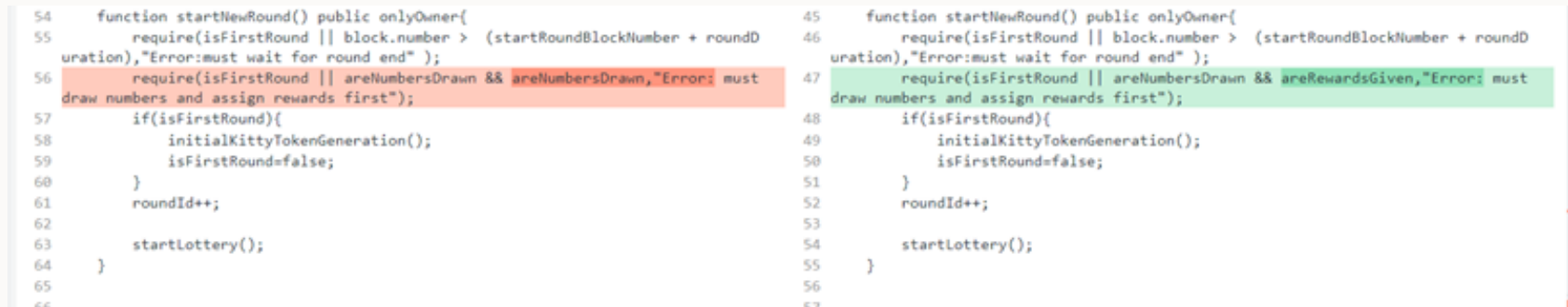
Figure 12: GUI of player displaying kitties and a way to transfer them

4 Smart Contract fix

The Smart Contract included in this project has three major differences in respect to the mid-term delivered one.

4.1 Fix 1

In a requirement of the method `startNewRound` there is a check on the boolean `areNumbersDraw` twice, instead of checking the variable `areRewardsGiven` and checking that in the previous round rewards were given. The fix can be seen in figure 13



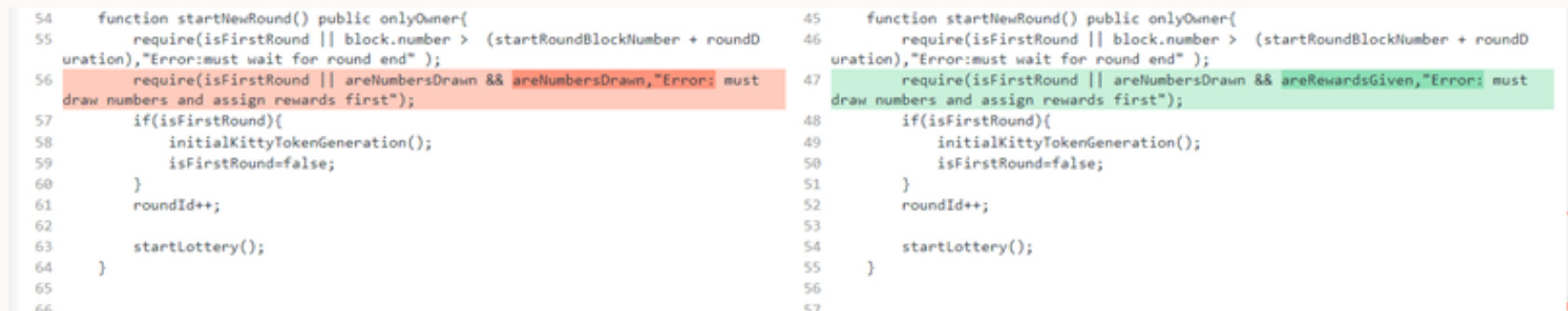
```
54  function startNewRound() public onlyOwner{
55      require(isFirstRound || block.number > (startRoundBlockNumber + roundD
uration),"Error:must wait for round end" );
56      require(isFirstRound || areNumbersDrawn && areNumbersDrawn,"Error: must
draw numbers and assign rewards first");
57      if(isFirstRound){
58          initialKittyTokenGeneration();
59          isFirstRound=false;
60      }
61      roundId++;
62
63      startLottery();
64  }
65
66
```

```
45  function startNewRound() public onlyOwner{
46      require(isFirstRound || block.number > (startRoundBlockNumber + roundD
uration),"Error:must wait for round end" );
47      require(isFirstRound || areNumbersDrawn && areRewardsGiven,"Error: must
draw numbers and assign rewards first");
48      if(isFirstRound){
49          initialKittyTokenGeneration();
50          isFirstRound=false;
51      }
52      roundId++;
53
54      startLottery();
55  }
56
57
```

Figure 13: StartNewRound function changes, on the left the old code, on the right the new code

4.2 Fix 2

The lottery operator can define the address to which send the fees collected for a round when the prizes are given, this was not considered in the final term code, where the fees were given to the lottery operator's address instead. This has been fixed by passing an address to the givePrizes function instead, as can be seen in figure 14



```
54 function startNewRound() public onlyOwner{
55     require(isFirstRound || block.number > (startRoundBlockNumber + roundDuration), "Error: must wait for round end" );
56     require(isFirstRound || areNumbersDrawn && areNumbersDrawn, "Error: must draw numbers and assign rewards first");
57     if(isFirstRound){
58         initialKittyTokenGeneration();
59         isFirstRound=false;
60     }
61     roundId++;
62
63     startLottery();
64 }
65
45 function startNewRound() public onlyOwner{
46     require(isFirstRound || block.number > (startRoundBlockNumber + roundDuration), "Error: must wait for round end" );
47     require(isFirstRound || areNumbersDrawn && areRewardsGiven, "Error: must draw numbers and assign rewards first");
48     if(isFirstRound){
49         initialKittyTokenGeneration();
50         isFirstRound=false;
51     }
52     roundId++;
53
54     startLottery();
55 }
56
57
```

Figure 14: Excerpt of givePrizes function changes, on the left the old code, on the right the new code

4.3 Fix 3

The final term code included an Interface, that should have allowed another Smart Contract to call a function eventCreationEmit, not implemented, that should have allowed creating a createdContract event with the parameters specified, after the creation of a Lottery contract from the other Smart contract. As after fix 2, the Lottery contract code would result in being too big to be deployed respecting the block gas limit of Ganache if

compiled without the Enable Optimization option, the interface has been removed. This can be seen in figure 15



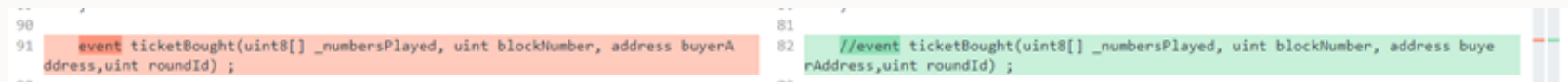
```
4 interface EventCreation {
5     function eventCreationEmit(address _lotteryAddress,address _lotteryCreator,u
6     int64 _fee, uint8 _roundDuration) external;
7 }
8
9
10 /// @title Try-An NFT lottery
```

```
4
5 /// @title Try-An NFT lottery
```

Figure 15: Excerpt the contract interface declaration, the old code on the left includes it, the new code on the right does not

4.4 Fix 4

After fix 2 and even with fix 3, the contract constructor function would exceed the block gas limit. To fix this the event ticketBought is commented out. It was not required by the specification and was triggered every time a ticket was bought. The change to the declaration can be seen in figure 16.



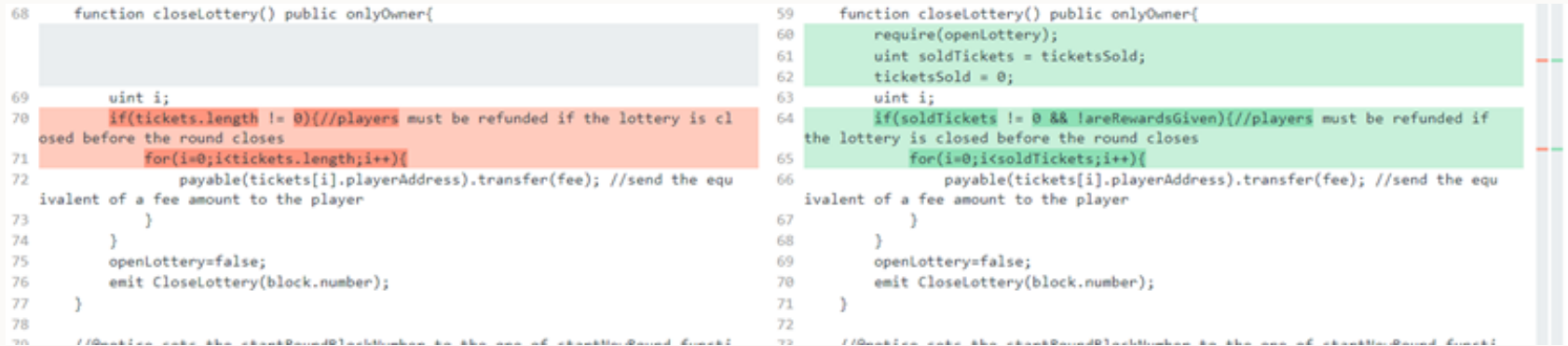
```
90
91 event ticketBought(uint8[] _numbersPlayed, uint blockNumber, address buyerA
92 address,uint roundId) ;
```

```
81
82 //event ticketBought(uint8[] _numbersPlayed, uint blockNumber, address buye
83 rAddress,uint roundId) ;
```

Figure 16: ticketBought event being commented out, the old code on the left includes it, the new code on the right comments the command

4.5 Fix 5

The lottery close operation has an issue. If triggered after the first round ends and in the following rounds the number of tickets bought in the current round is less than in the previous round. This happened because before closing the lottery the array holding the ticket objects' length was checked, and it does not shrink in size after a round. To solve this, the number of tickets to issue a refund is counted using the value of the variable holding the number of tickets bought in the current round. Also, a check to see if the lottery was not already closed is included. The changes can be seen in figure 17.



```
68 function closeLottery() public onlyOwner{
69     uint i;
70     if(tickets.length != 0){//players must be refunded if the lottery is cl
71         for(i=0;i<tickets.length;i++){
72             payable(tickets[i].playerAddress).transfer(fee); //send the equ
73             ivalent of a fee amount to the player
74         }
75         openLottery=false;
76         emit CloseLottery(block.number);
77     }
78 }
79 //Before this the closeLottery function was the one of the old version
```

```
59 function closeLottery() public onlyOwner{
60     require(openLottery);
61     uint soldTickets = ticketsSold;
62     ticketsSold = 0;
63     uint i;
64     if(soldTickets != 0 && !areRewardsGiven){//players must be refunded if
65         the lottery is closed before the round closes
66         for(i=0;i<soldTickets;i++){
67             payable(tickets[i].playerAddress).transfer(fee); //send the equ
68             ivalent of a fee amount to the player
69         }
70         openLottery=false;
71         emit CloseLottery(block.number);
72     }
73 }
74 //Before this the closeLottery function was the one of the old version
```

Figure 17: CloseLottery function changes, the old code on the left, the new code on the right

Bibliography

- [1] Ethers. *Ethers.js*. URL: <https://docs.ethers.io/v5/>.
- [2] Metamask. *Ethereum Provider API*. URL: <https://docs.metamask.io/guide/ethereum-provider.html>.