Christmas Lottery

Davide Scarrà

Idea

The process of participating in the lottery begins when a user wants to purchase one or more tickets. The user contacts the contract owner, providing personal information such as name, surname, and student ID, along with the amount of money required to purchase the ticket.

Subsequently, the contract owner uses the `addParticipant` function to register the user, generating a ticket associated with the information provided by the user. This ticket contains the owner's name, surname, and student ID plus an identifier and a timestamp.

On the day of the draw, the contract owner presses the `getWinner` button, specifying the number *n* of desired winners. At this point, the contract randomly selects *n* winners from all registered participants on the blockchain. Each participant is eligible to win only one prize; the more tickets you purchase, the higher your chances of winning the first prize.

Once the draw is completed, the winner's data becomes available to the owner through the `showWinners` button, who can then announce the lucky winners of the lottery.

Additionally, the owner can also reset the participants and the winners with the `resetParticipants` and `resetWinners` functionalities, allowing the lottery to be used multiple times.

Smart contract implementation

Address: *0x5d33AD128Cf17c49DFeB711cBB006395026A5b27*

The smart contract has been implemented in Solidity using Remix IDE.

It provides three read methods:

1. `isOwner`: returns true if the caller's wallet address corresponds to the owner address, otherwise returns false.
2. `participantsCount`: returns the number of lottery participants.

3. `showWinners`: returns a list of tuples containing *firstname*, *lastname*, and *studentID* of the winners drawn from the last reset of the winners/participants.

And four write methods:

1. `addTicket` (`onlyOwner`): takes input from the owner for *firstname*, *lastname, studentID,* and *number*, and inserts [*number]* tickets associated with the person's details into the lottery.
2. `drawTicket` (`onlyOwner`): takes input a number and randomly extracts [number] tickets from the lottery.
3. `resetParticipants` (`onlyOwner`): resets the participants (and also the winners) in the lottery to 0.
4. `resetWinners` (`onlyOwner`): reset only the winners to 0.

The competitors' tickets are stored in the following structure:

```
struct Ticket {
  uint256 timestamp;
  uint256 identifier;
  TicketOwner ticketOwner;
}
```

Where the owner's details are stored in the following structure:

```
struct TicketOwner {
  string firstname;
  string lastname;
  string studentID;
}
```

Sold tickets are stored in the dynamic array `soldTickets` containing data of type `Ticket`.

The details of the drawn winners are stored in the dynamic array `winners` containing data of type `TicketOwner`.

There is also a mapping to keep track of participants:

```
mapping(string => TicketOwner) private participants;
```

This allows for a quick check to see if a customer already had a ticket when adding a new ticket and not increment the number of participants accordingly.

```
function addParticipant(string memory _firstname,
                                                string memory
                                                string memory
    // if the participant is not already in the list of parti
    if (bytes(participants[_studentID].studentID).length == 0
        // add the participant to the mapping of participants
                // with the studentID as key
        participants[_studentID] = TicketOwner(_firstname,


        participantsCount++;
    }
}
```

There is also a mapping for verifying whether a participant has already won a prize:

```
mapping(string => bool) private winnersMap;
```

This mapping enables a quick check to determine if a randomly selected participant in the winner selection has already won a prize:

```
...
// check if the winner is not already in the list of winners
if (!winnersMap[winner]) {
  // if not, add the winner to the list of winners
  winnersMap[winner] = true;
  return soldTickets[randomIndex].ticketOwner;
}
...
```

The number of participants is stored in a public variable so that anyone can see it.

```
uint public participantsCount;
```

I created two types of events:

```
event TicketAdded(uint256 _timestamp, string _studentID,
                  string _firstname, string _lastname);
event WinnerDrawn(string _firstname, string _lastname,
                  string _studentID);
```

The `TicketAdded` event will be emitted when each ticket is added, so if a user buys three tickets, the event will be emitted three times.

The `WinnerDrawn` event will be emitted when the owner draws the winner(s) and will be emitted for each winner, showing their *firstname*, *lastname*, and *studentID*.

There are also 3 modifiers:

```
// check if the caller is the contract owner
modifier onlyOwner() {
    require(msg.sender == contractOwner, "Only the contract o
    _; // Continue with the function execution if the modifie
}

// check if the string is not empty
modifier nonEmptyString(string memory _str) {
    require(bytes(_str).length > 0, "Empty string not allowed
    _; // Continue with the function execution if the modifie
}

// check if the number is not zero
modifier positiveNum(uint256 _num) {
    require(_num > 0, "Zero value not allowed");
    _; // Continue with the function execution if the modifie
}
```

The `addTicket` function increments the number of participants if the customer has not yet purchased any tickets and inserts the newly purchased tickets into the `soldTickets` array.

The `drawTicket` function first checks if the number of sold tickets is greater than 0. If so, it randomly extracts the desired number of tickets from the array of sold tickets and inserts the owner(s) of that ticket(s) into the winners' array.

This function relies on the `getAWinner` function, which extracts and returns the owner of a winning ticket (if not already in the winners' array), and the `randomNumber` function, which generates a random number between 0 and the number of sold tickets.

The `getAWinner` function consists of a while loop that utilizes the `randomNumber` function to obtain a random number. This random number is then used to retrieve a participant from the array of sold tickets. If the retrieved participant has not won yet, they are returned as the winner; otherwise, the procedure is repeated.

Initially, I implemented the `getAWinner` function in a recursive fashion. However, I read that this is not a good practice as it makes it difficult to estimate the gas consumption of the function.

```
// generate a random number between 0 and tickets.length
function randomNumber() private returns (uint256) {
    uint256 random = uint256(keccak256(abi.encodePacked(block
                                                        b
    incr++;
    return random % soldTickets.length;
}

function getAWinner() private returns (TicketOwner memory) {
    require(participantsCount > winners.length, "No more tick

    uint256 attempts = 0;
    while (attempts < soldTickets.length) {
        uint256 randomIndex = randomNumber();

        // get the winner studentID from the list of sold tic
        string memory winner = soldTickets[randomIndex].ticke

        // check if the winner is already in the list of winn
        if (!winnersMap[winner]) {
            // if not, add the winner to the list of winners
            winnersMap[winner] = true;
            return soldTickets[randomIndex].ticketOwner;
        }
        // if the winner is already in the list of winners, t
        attempts++;
    }
```

```
    // this should never happen because the number of partici
    // than the number of winners so there should always be a
    revert("Failed to find a winner");
}
```

Front-end

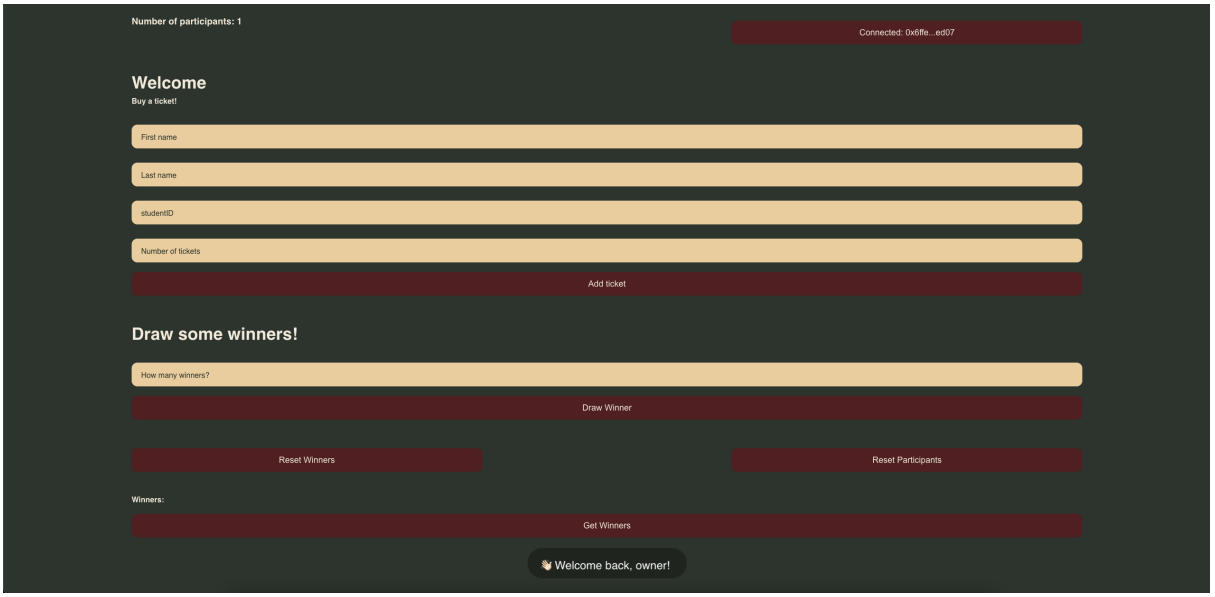
I implemented the front-end using ReactJS, using the Web3 library to handle communication with the smart contract and Infura as a provider.

It offers two different interfaces based on whether the connected wallet is that of the owner or not.

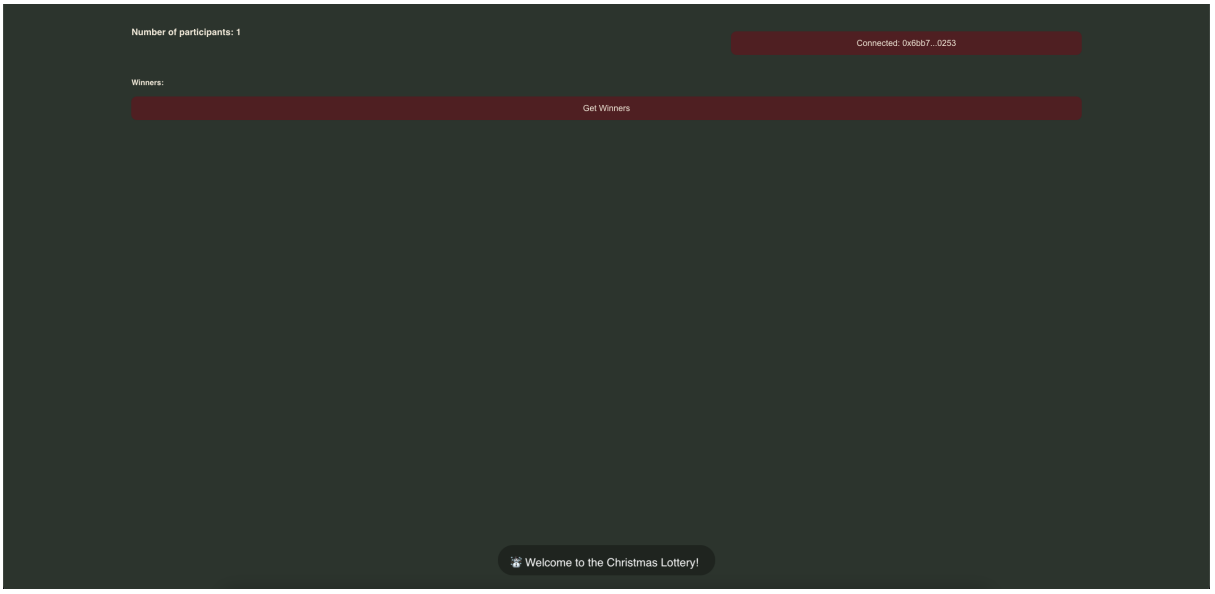
If the wallet belongs to the owner, the site will display all components:

1. The number of participants (refreshed on each page reload)
2. The button to connect/view the wallet
3. The form to add new tickets to the lottery
4. The form to draw winners
5. The button to reset participants
6. The button to reset winners
7. The button to view the winners drawn from the last reset
8. The status bar

If the connected wallet does not belong to the owner, the site will only show components 1, 2, 6, and 8.



Website View for Contract Owner



Website view for a guest

Project Structure Overview

the `App()` function, inside the */App.js* file, is the root component of the application.

Here all the components are rendered and is checked if the connected wallet belongs to the contract owner, adjusting the displayed components accordingly.

To determine ownership, the `isOwner` method within the smart contract is invoked through the `checkIsOwner` function. This check occurs on each page reload or whenever the connected wallet changes, leveraging the `useEffect` hook.

Moreover, within `App.js`, there's an attempt to implement an event listener for the smart contract event `WinnerDrawn` but I encountered difficulties in getting it to work as intended.

Every component rendered by the `App()` function is located in the */components* directory. Each component has its own folder, including a JavaScript file for its logic and a CSS file for its specific styles. The components' logic is straightforward, typically involving the HTML, a call to the corresponding contract methods, an update of the status bar, and a return of the data provided by the smart contract method. Therefore, I won't delve into the details of each component in this report.

All interactions with the methods exposed by the contract are encapsulated within the */utils/contract.js* file. Meanwhile, the */utils/wallet.js* file encompasses all the logic for connecting a wallet to the website.

Connection to MetaMask Wallet

In order to establish a connection with the MetaMask wallet, I followed the Web3 tutorial available at Web3 University. The entire connection logic is encapsulated within the */utils/wallet.js* file.

When a user clicks the "Connect Wallet" button, the `connectWallet` function is invoked. This function utilizes the APIs injected by MetaMask to initiate a connection.

It requests the user to grant access to their Ethereum accounts through the `eth_requestAccounts` method, in other words it's asking the user to connect or unlock their MetaMask wallet and share their Ethereum accounts.

Subsequently, it retrieves the first address from the array of addresses connected to MetaMask and returns it.

If the user does not have MetaMask installed in their browser, an informative error message is displayed, encouraging the user to download MetaMask.

```
export const connectWallet = async () => {
    // checks if window.ethereum is installed in browser
    if (window.ethereum) {
        try {
            // try to connect to Metamask. Calling this funct
            const addressArray = await window.ethereum.reques
                method: "eth_requestAccounts",
            });
            const obj = {
```

```
                // take the first address in the array of add
                address: addressArray[0],
            };
            return obj;
            // if user denies access to their Metamask accoun
        } catch (err) {
            return {
                address: "",
                status: "😢 " + err.message,
            };
        }
    } else {
        return {
            address: "",
            status: (
                <span>
                    <p>
                        {" "}
                        🦊{" "}
                        <a target="_blank" href={`https://met
                            {INST_METAMASK}
                        </a>
                    </p>
                </span>
            ),
        };
    }
};
```

To maintain user connectivity even after a page reload, the `getCurrentWalletConnected` function is called within the `useEffect` of the `walletConnection` component. This function employs the `eth_accounts` method to request an array of Ethereum addresses authorized by the user to be accessed by the app and then the first address is taken. If the user hasn't authorized any address, an error message is displayed.

```
export const getCurrentWalletConnected = async () => {
    if (window.ethereum) {
```

```
        try {
            const addressArray = await window.ethereum.reques
                method: "eth_accounts",
            });
            // if Metamask is connected then take the first a
            if (addressArray.length > 0) {
                return {
                    address: addressArray[0]
                };
            } else {
                return {
                    address: "",
                    status: "🦊 Connect to Metamask using the
                };
            }
        } catch (err) {
            return {
                address: "",
                status: "😢 " + err.message,
            };
        }
    } else {
        return {
            address: "",
            status: (
                <span>
                    <p>
                        {" "}
                        🦊{" "}
                        <a target="_blank" href={`https://met
                            {INST_METAMASK}
                        </a>
                    </p>
                </span>
            ),
        };
    }
};
```

This implementation ensures a seamless user experience by maintaining the connection even after a page reload.

The `walletListener` function within the `walletConnection` component captures changes in the connected wallet. In the event of a user changing the connected wallet, it is automatically reflected to the page.

```
function walletListener() {
    if (window.ethereum) {
        window.ethereum.on("accountsChanged", (accounts) => {
            if (accounts.length > 0) {
                setWallet(accounts[0]);
            } else {
                setWallet("");
                setStatus("🦊 Connect to Metamask using the \
            }
        });
    } else {
        setStatus(
            <p>
                {" "}
                🦊{" "}
                <a target="_blank" href={`https://metamask.io
                    You must install Metamask in your browser
                </a>
            </p>
        );
    }
}
```

The function registers an event listener for the `accountsChanged` event. This event is emitted by MetaMask when there is a change in the connected Ethereum accounts.

When a wallet is connected, the Connect Wallet button displays the first and last characters of the connected address to the user.

Interraction with the contract

Inside the */utils/contract.js* there is all the logic for calling the smart contract functions previously listed.

Here the ABIs are imported, and the address where the smart contract is deployed is defined.

All writing methods utilize the `sendTransaction` function by providing it with the method to call, the parameters to pass, and the wallet of the transaction initiator. This decreases the boilerplate code.

```
const sendTransaction = async (method, parameters, walletAddr
    try {
        //set up transaction parameters
        const transactionParameters = {
            to: contractAddress, // Required except during co
            from: walletAddress, // must match user's active
            data: christmas_lottery_contract.methods[method](
        };
        //sign the transaction
        const txHash = await window.ethereum.request({
            method: "eth_sendTransaction",
            params: [transactionParameters],
        });
        // show a message to the user that the transaction ha
        return {
            status: (
                <span>
                    ✅{" "}
                    <a target="_blank" href={`https://sepolia
                        View the status of your transaction o
                    </a>
                </span>
            ),
        };
    } catch (error) {
        // in case of an error, show a message to the user
        // a possible error is that the user rejected the tra
        console.error(`Error during the transaction ${method}
        return {
            status: "😢 " + error.message,
        };
```

```
    }
};
```

The function prepares the parameters for the Ethereum transaction, including the target contract address, the wallet address initiating the transaction, and the encoded data representing the specific method and its parameters.

It then signs the transaction using the `eth_sendTransaction` method provided by MetaMask, which triggers the user to confirm the transaction in their wallet.

If the transaction is successful, it returns a message with a success status, providing a link for the user to view the transaction details on Etherscan. In case of an error, it returns an error message, which could occur for example if the user rejects the transaction or due to other transaction-related issues.

Also this function was implemented following the official Web3 University guide.

Additionally, at the bottom of the screen, the site displays a status bar where informative or error messages can be printed:

- 🦊 Connect to Metamask using the “Connect Wallet” button: prompts the user to connect a wallet.
- 🦊 You must install Metamask in your browser.: If the user does not have Metamask installed.
- 💡 Connect your Metamask wallet to play with the lottery: prompts the user to connect a wallet if they clicked the button to view the winners (the only visible button if an account is not connected).
- 👋🏻 Welcome back, owner!: displayed when connected with the owner's wallet.
- ☃️ Welcome to the Christmas Lottery!: displayed when connected with a wallet that does not belong to the owner.
- ✅ View the status of your transaction on Etherscan!: displayed when interacting with write methods of the contract.
- 😥 *<error message>*: if something went wrong.
- 😥 No winners yet!: If the user attempts to view the winners, and no winners have been drawn yet.

Issues Encountered

I observed that if the condition of a `require()` is not met, Metamask notifies the user that there might be an error in the contract and prevents the transaction from being accepted.

This occurred when attempting to draw a number of tickets greater than the possible winners. I resolved the issue by implementing a JavaScript check that prevents the user from drawing an excessively high number of winners.

As mentioned earlier, I encountered challenges while working on the event listener intended to capture WinnerDrawn events. Despite attempting various implementations, the event listener appears to ignore the events emitted by the smart contract.

```
function eventListener() {
  christmas_lottery_contract.events["WinnerDrawn"]({}, (error
    if (!error) {
      // Handle the event data
      const eventData = event.returnValues;
      setStatus(
        `🎉 Congratulations to the winner: ${eventData.firstn
      );
    } else {
      console.error('Error listening to the event:', error);
      setStatus("😢 Some error occurred while listening to th
    }
  });
}
```