Introduction

The project required to modify the initial smart contract provided for the final term, and that implements a voting system, in two parts:

1. In the first part, instead of considering one candidate, the smart contract has to consider one or more candidates.
2. The second part consists of implementing a different way in which the souls are awarded to candidates/electors OR implementation of the soul as an ERC20 token OR considering coalitions of candidates. For the second part, I choose to implement the soul as an ERC20 token.

In the following sections, I explain the tools that were used (in particular focusing on Drizzle that was briefly mentioned during the course), the changes done to the original smart contract for the first/second part, the commands that can be issued to test/deploy the contracts and to launch the dapp. In the end, I'll present a short demo that will show the dapp usage and the main cases that can occur when declaring a mayor/tie.

Tools used

The tool used to write, execute and test the contract is Truffle and Ganache. For the ERC20 implementation, I used the OpenZeppelin library. For the dapp development and the front-end, I used Drizzle, React (with the toolchain create-react-app and Material-UI framework) and Metamask.

Truffle and Ganache

Truffle takes care of managing contracts, artifacts, deployments and testing. Ganache is used under the hood to generate a private blockchain when testing/executing the contracts in the development network.

The Truffle project delivered is divided into different directories and files:

- `/contracts`: in this directory, you'll find the Mayor contract (`mayor.sol`), the SOUToken contract (`soul.sol`) that implements the ERC20 token and the standard Migration contract (`Migrations.sol`) provided by Truffle that

implement keeps track of the migrations done when issuing the truffle deploy command.

- `/migrations`: in this directory, you'll find the `initial_migrations.js` that is provided by Truffle and only deploy the Migration contract and `mayorsoul_migrations.js` that deploy both the Mayor and Soul contract outputting in the console some information related to the addresses of both contracts, the quorum, the candidates, the escrow account address.
- `/test`: in this directory, you'll find the `test_base.js` that includes checks for all functions/state variables of the Mayor/Soul contract.
- `/client`: in this directory, you'll find all the code that is needed to run the dapp, including node dependencies specified into `package.json`.
- `truffle_config.js`: this is the standard configuration file, some parts are modified with respect to the original one:
    - The port of the `development` network is setted as 9545.
    - The directory that will contain the artifact files is moved into the client directory to be accessed by Drizzle. This is done by adding a new section called `contracts_build_directory`.
    - The usage of a plugin, called `solidity-coverage` and specified in the section `plugins`, to assess the lines/branches covered by `test_base.js`.
- `package.json` and `package-lock.json`: define the dependencies needed to deploy, test and execute the contracts.

Drizzle

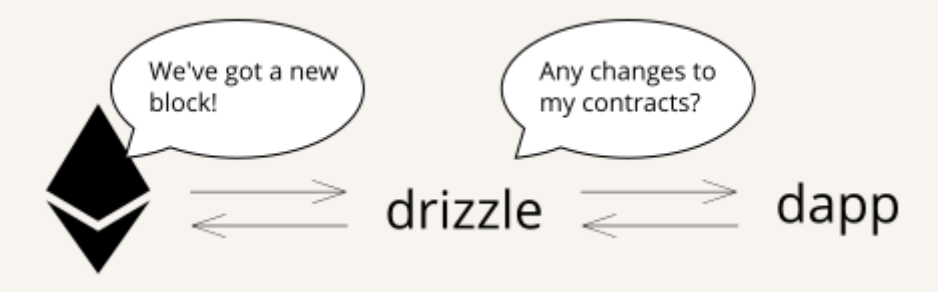
Drizzle is a collection of front-end libraries that make writing dapp front-ends easier and more predictable. In particular, Drizzle synchronizes the contract data and the related transactions using a Redux store and offers the methods `cacheSend` and `cacheCall` that abstracts all the calls to `web3` library.

A Redux store is used mostly for application state management by maintaining the state of an entire application in a single immutable state tree (object), which can't be changed directly. When something changes, a new object is created (using actions and reducers).

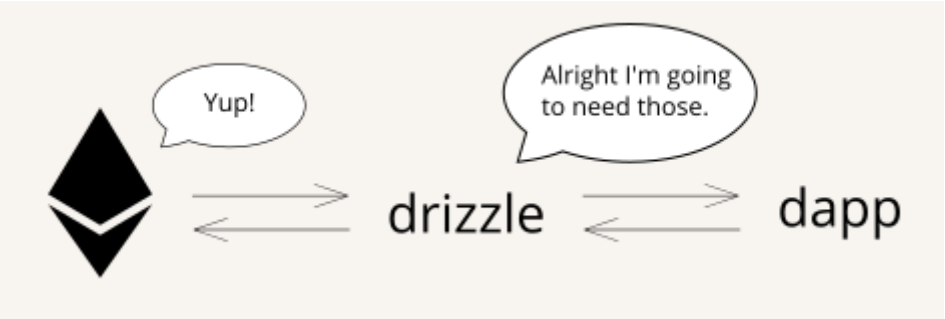
1) Once initialized, Drizzle instantiates `web3` and the contracts specified in the `drizzleOptions` (defined in the file `/client/src/index.js`), then observes the chain by subscribing to new block headers.

2) Drizzle keeps track of contract calls so it knows what data to synchronize.

3) When a new block header comes in, Drizzle checks that the block isn't pending, then goes through the transactions looking to see if any of them touched the contracts specified before.



4) If they did, then Drizzle replay the calls already in the store to refresh any potentially altered data. If they didn't, the dapp will continue to work with the stored data.



Drizzle also allows to track events (defined in the section `events` of `drizzleOptions`) and transaction errors. This information will be displayed in the dapp by using a notify mechanism based on `react-toastify`.

Drizzle is also modular, it allows the use of other packages (for example to interact with React applications). In this project the components of Drizzle that were used are `drizzle-store` and `drizzle-react` (provides `DrizzleProvider` `DrizzleConsumer` and other components to make it easier to connect Drizzle with the React application).

First part of the project

- To keep track of all the votes and souls that a candidate has received, I introduced a new data structure called `AccumulatedVote`. To represent a tie situation, in which there are two or more candidates that have the same amount of votes, I replaced the event `Sayonara` with a new event called `Tie` that will have as a parameter an address that will be instantiated in `mayor_or_sayonara` with the escrow account. Each `AccumulatedVote` will be stored in a `mapping(address => AccumulatedVote)` called `candidates_info`. As I cannot retrieve the addresses of the candidates from this mapping, I introduced an array of addresses called `candidates`. In the `Refund` data structure, instead of considering a doblon (`bool`) I consider the symbol of the candidate that is voted but lost the election.
- The original `constructor` function is modified, it receives an array of addresses instead of the address of one candidate and, for each of the candidates, a new `AccumulatedVote` is created and stored in the mapping.

- The function `compute_envelope` is modified including a new require condition that checks that the candidate specified in the envelope exists.
- The function `cast_envelope` is not modified.
- The function `open_envelope` is modified by adding the operations that modify the state of `AccumulatedVote` for that particular symbol (candidate).
- The function `mayor_or_sayonara` is completely changed. To implement the logic that will declare the mayor, I saved the `AccumulatedVote` of the first candidate and, I used a for loop starting from the second candidate that, checks (using the respective `AccumulatedVote`):
    - If the candidate considered has more souls (or if the souls are equal we compare the votes) there may be a winner (bool variable `_winner` set to true, `_mayor` address set and amount of souls that he should be receiving assigned to the variable `_soulsToMayor`).
    - If both souls and votes are equal I keep track that there may not be a winner and update the bool variable `_winner` to false.
    - In both cases I update a variable that contains all the souls that should be transferred to the escrow in case of a tie.

  After the for loop I simply check the variable `_winner` to see if there will be a mayor or a tie:
    - If it is true then we have a new mayor, in this case I implement the same for loop used in the final term to refund all the loser voters. Then I transfer the souls tracked in the variable `_soulsToMayor` to the winner address and emit the event `NewMayor` with the winner's address.
    - If it is false then we have a tie situation. In this case, all the souls tracked in the variable `_soulsToEscrow` are sent to the escrow account and emit the event `Tie` with the escrow address.

Second part of the project

For the second part of the project, I decided to implement the souls as an ERC20 token called SOU. The implementation used is provided by the `@openzeppelin/contracts` library. With this library, it is possible to implement our token contract (SOUToken) by simply extending the ERC20 contract provided by OpenZeppelin. The token contract is created independently from the Mayor contract to achieve the best flexibility and allow the transfer of tokens between the users.

The main functions provided in the SOUToken smart contract are:

- The `constructor` simply initialises the main information of the token: the name (Soul) and the symbol (SOU). The decimals are not modified and by default are set to 18.
- The `mint` receives the address on which to deliver the tokens (100) that will be created. To check if that address has already received the tokens I used a mechanism similar to the one used in the `cast_envelope` function to keep track of the counter of envelopes cast.

The functions of the Mayor contract are kept basically the same:

- The `constructor` is modified introducing a new parameter (`_token`) that is the reference to the token smart contract.
- The function `cast_envelope` includes the calling to the mint operation to send the voters some tokens that will be used in the `open_envelope`.
- The function `open_envelope` is modified by including the `transferFrom` operation to transfer the Soul tokens from the voter address to the Mayor contract address. To use `transferFrom` it is important that the Mayor contract has a delegation on a certain amount of tokens (this can be achieved by calling the function `approve` of the ERC20 contract).
- The function `mayor_or_sayonara` includes all the transfer operations that now are called from the SOUToken smart contract.

Commands

First of all, we need to install all the dependencies:

- To install Truffle simply issue the command `(sudo) npm install -g truffle`.
- If you want to deploy/test the contracts just use the command `npm install` in the directory `/project`.
- If you also want to execute the dapp, issue another `npm install` command in the directory `/project/client`.

To test the contract and get the coverage (both in CLI and with a simple web based GUI) issue the command `truffle run coverage` (the plugin takes care of deploying the Ganache server).

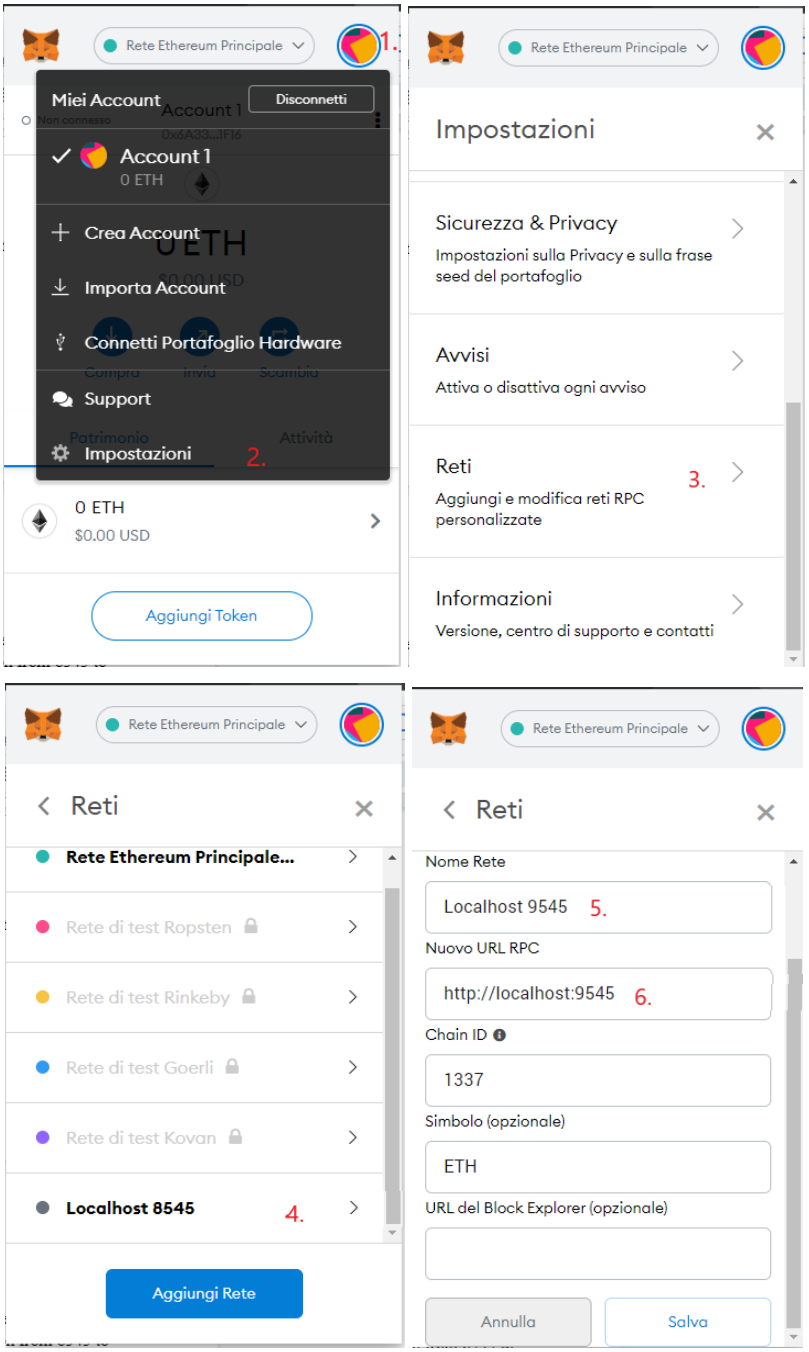
To execute the dapp, first migrate the contracts using the following commands:

- `truffle develop`: this command spawns a local development blockchain and allows you to interact with contracts via the command line. Additionally, many Truffle commands are available within the console. In particular it creates the standard development accounts/private keys that will be needed to import in Metamask to interact with the dapp.
- Now simply issue the `migrate` command that will deploy all the three contracts into the development blockchain.

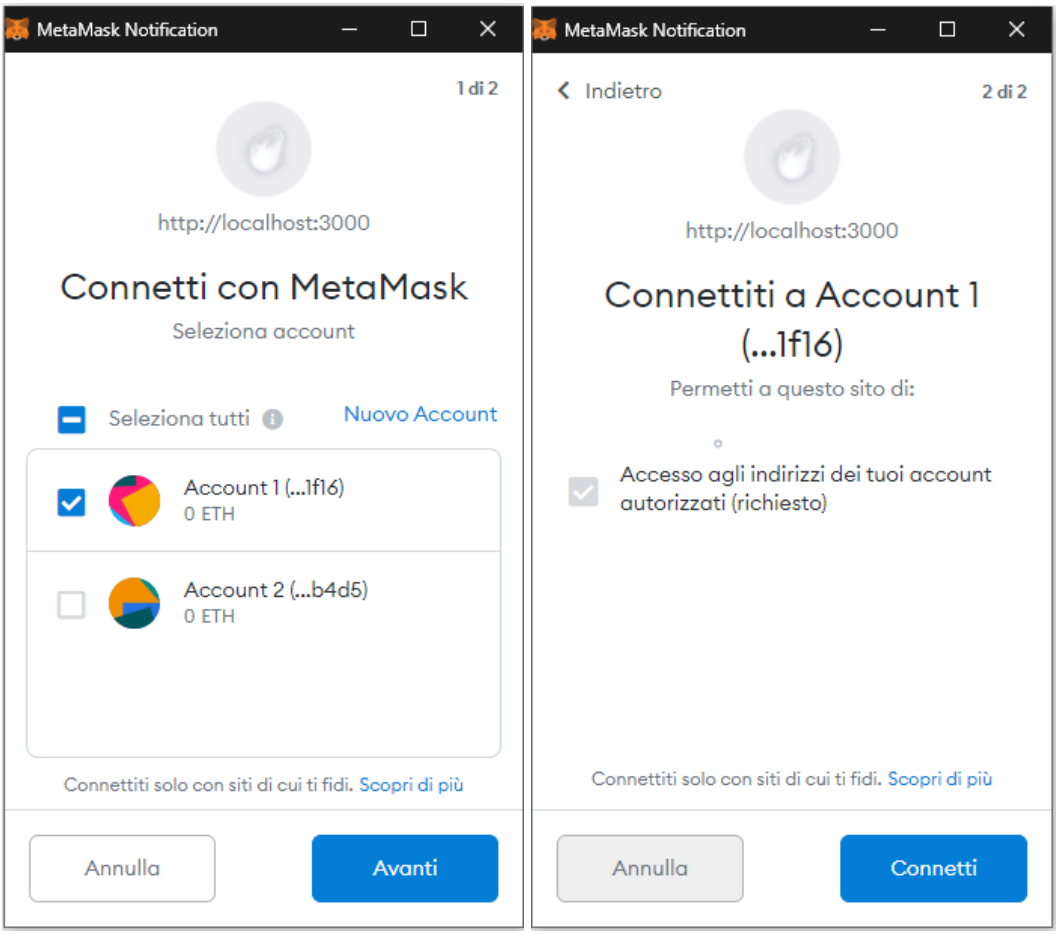
Now go to the directory `/project/client` and issue the command `npm start`. This command will run the dapp in development mode.

Demo

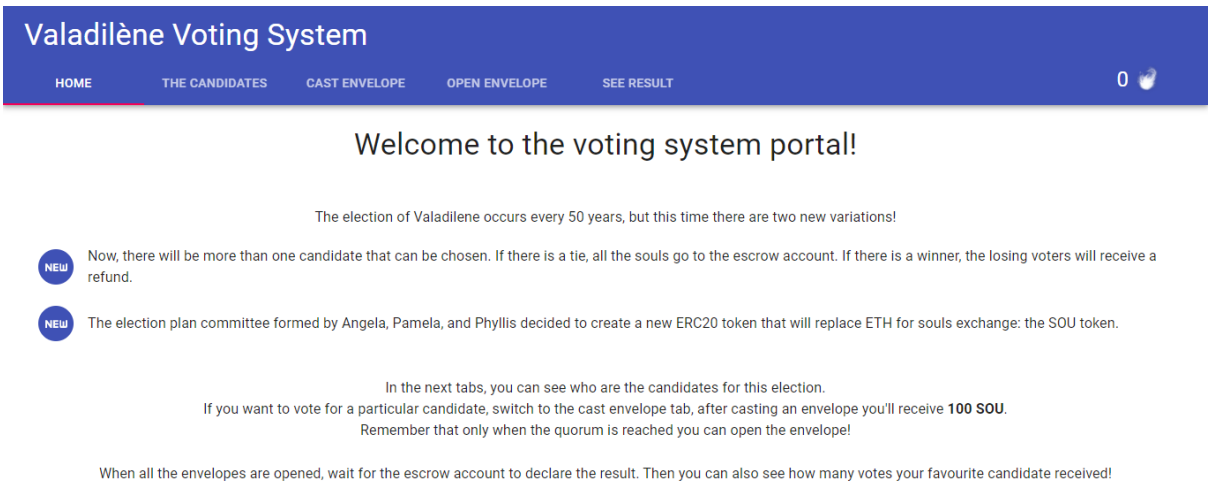
For this demo I used Metamask, the contract will be deployed using the commands specified before with a quorum of 4 voters and 3 candidates (first three addresses of `accounts` array). The voter addresses that I used are the first three and the last one (escrow address). After deploying the contract, we have to change the port of the local network in the metamask extension from 8545 to 9545.



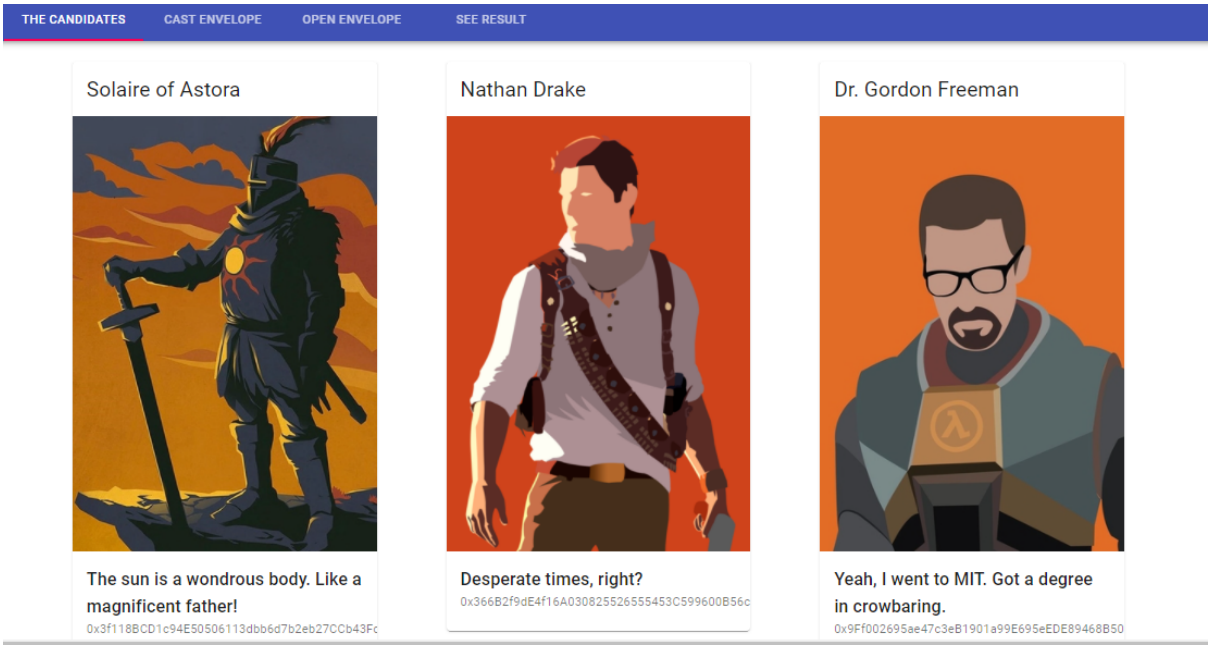
Now run `npm start` inside the `/project/client` directory and wait until the development server is ready. If Metamask is correctly installed, the dapp will not load until you login with Metamask.



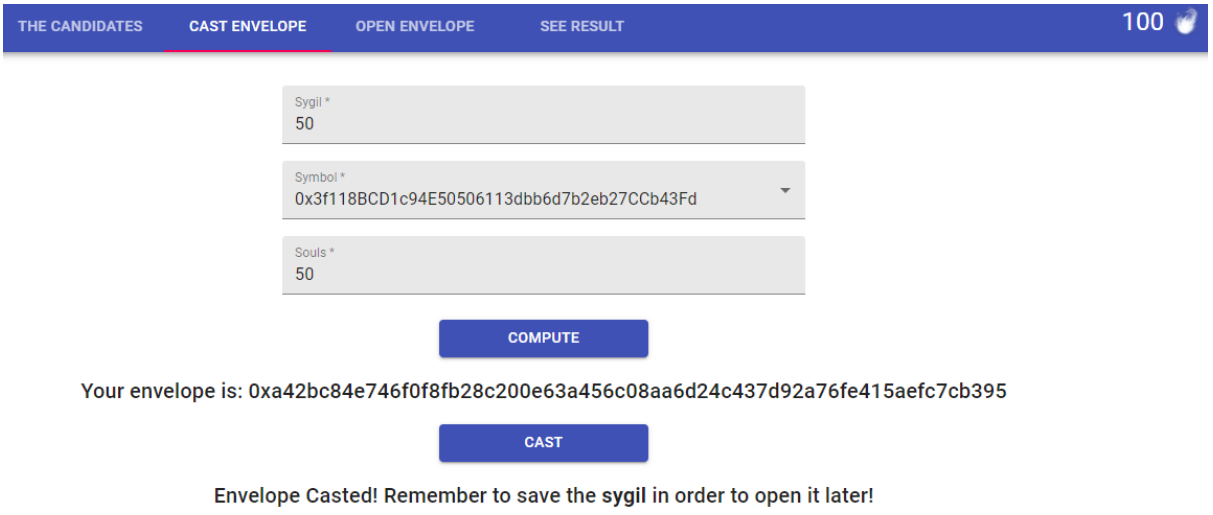
After the login, the dapp home page will be displayed. We can import the test accounts generated by Ganache (for this demo I used the first three accounts and the last one that is the escrow).



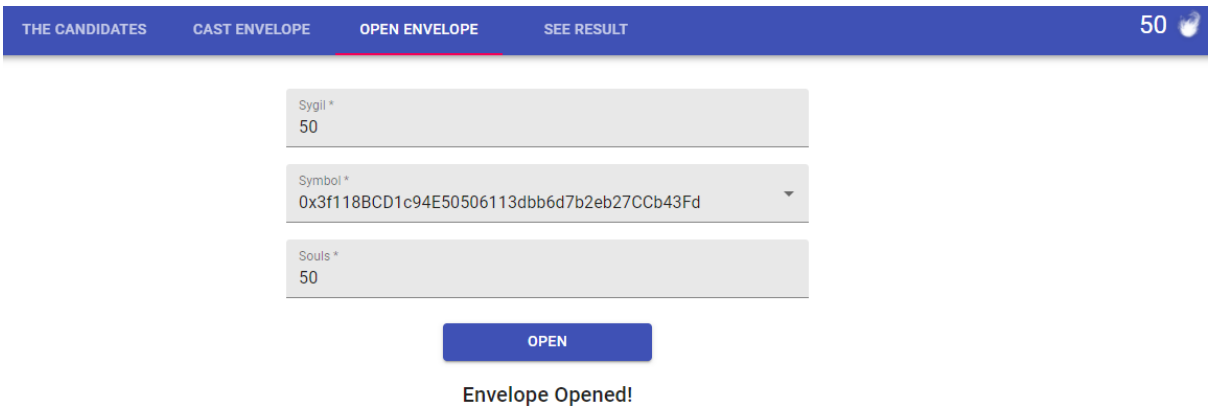
In “The candidates” tab, you can see who are the candidates for this election (name, slogan and address).



In the “Cast envelope” and “Open envelope” tab, there will be a simple form that will be used to get the data for these operations. In the following picture, there is a screen of a successfully casted envelope. As we can notice, the soul tokens displayed are updated (the soul tokens will be also shown in the Metamask extension).

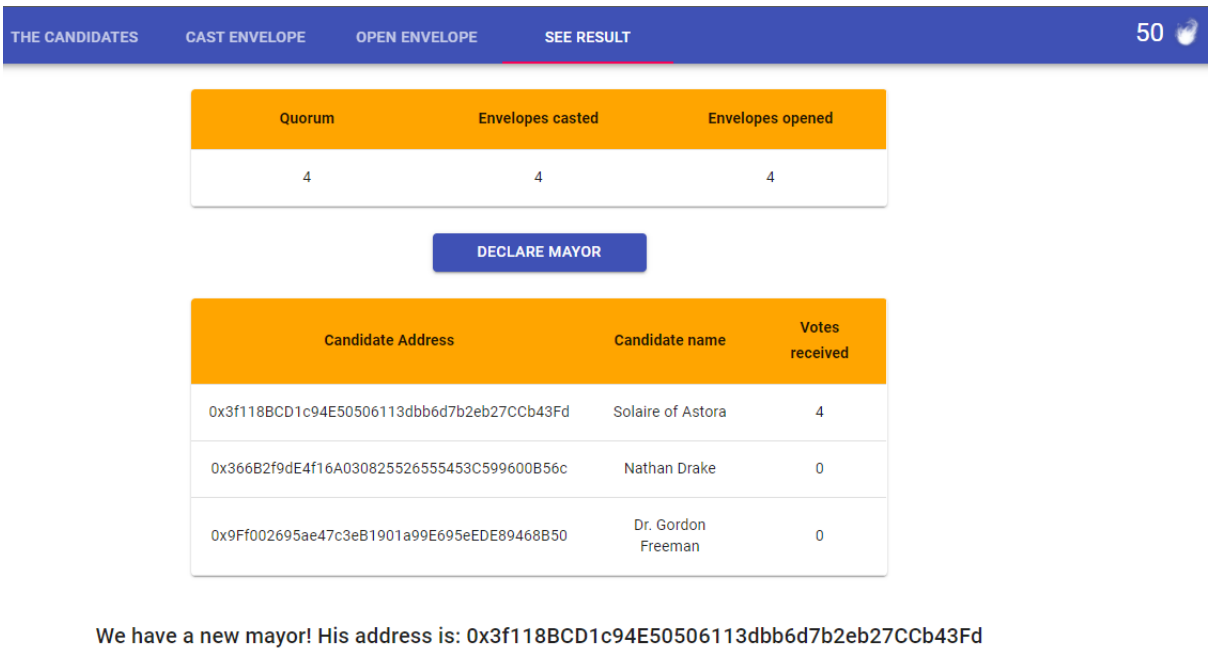


In the following picture, there is a screen of a successful open envelope. The souls are transferred from the voter account to the mayor contract address.



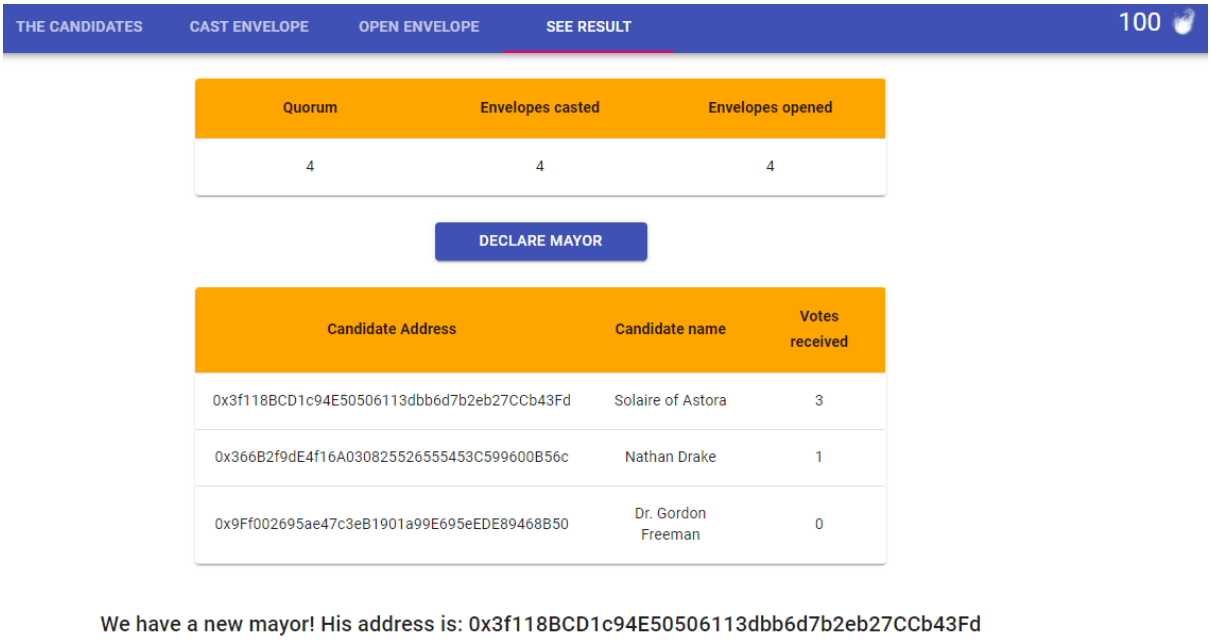
In the following picture, there is a screen of the "See result" tab. In this tab there will be displayed two tables:

- The first one is immediately available and lists the quorum, cast and opened envelopes. If the account currently selected in metamask is the escrow account, then there will also be a button that enables you to call `mayor_or_sayonara`.
- After this function is completed successfully, another table will be available to everyone that contains the number of votes each candidate has received and a simple paragraph that displays the address of the new mayor or the address of the escrow account if there is a tie.



New mayor (declared by souls)

Now, let’s suppose that 3 voters vote for the first candidate and the last one for the second (each voter uses 50 soul tokens). The results will be exactly the same as the following. In this case, the account currently selected is the escrow and in fact, it does receive back the soul tokens invested.

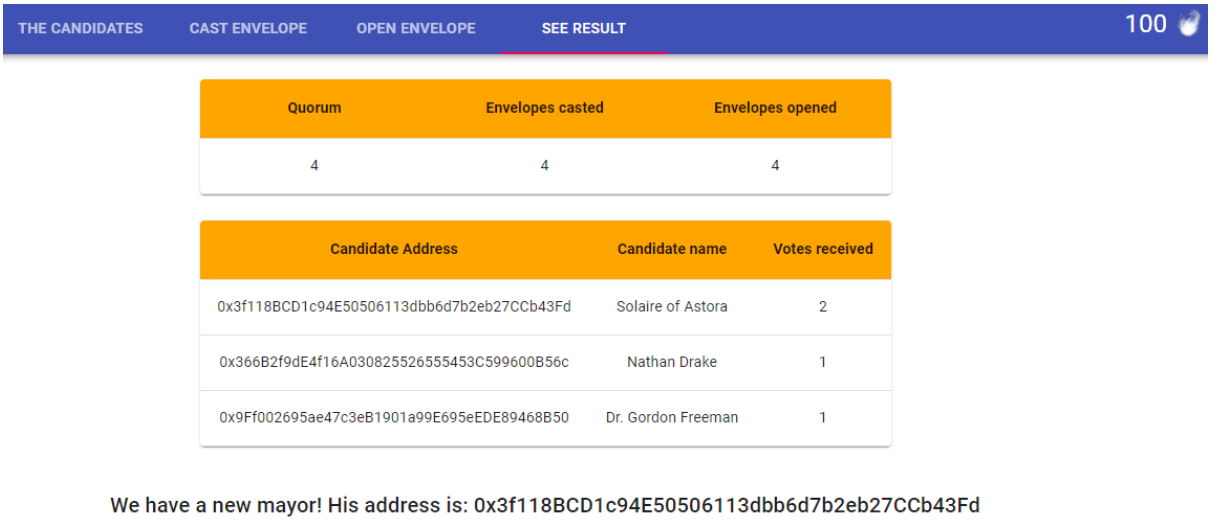


New mayor (declared by votes)

This case happens when two or more mayors have received (in the envelope) an equal amount of soul tokens but one of them has one or more votes than the others. Let's suppose that:

- The first two voters vote for the first candidate with 50 tokens each.
- The third voter votes for the second with 100 tokens.
- The fourth voter votes for the third with 100 tokens.

The results will be as follows, the account selected is of the third voter. The voter will receive back the tokens.



Tie

This happens when 2 or more candidates receive the same amount of souls and votes. In this case, there will be no mayor and all the souls will be transferred to the escrow account. Let's suppose that the first two voters vote for the first candidate and the other two for the second (in all cases the tokens will be equal to 100). The result will be as follows (from the escrow account perspective). The escrow will receive all the soul tokens.