P2PBC21

Final Project

Professor Laura Ricci

Alex Colucci **617783**

Contents

1 Introduction

In this final project, it is proposed an improvement over the solution of the final term. The smart contract has been generalized for multiple candidates and the Soul has been implemented as an ERC20 token. Afterwards, a decentralized application has been developed with Svelte and Web3.js. Moreover, to ensure a full decentralization, both in terms of backend and frontend, the DApp has been deployed over the IPFS network.
The following sections describe the new feature and changes over the contracts, with an explanation of the rationale. In addition, is explained the rationale and given a brief description of the stack for the DApp. Finally, is reported a demo scenario and instructions to run this solution.

2 Smart Contracts

The Smart Contracts that have been used are `Mayor.sol` for the election logic and `SULToken.sol` for using the Soul as an ERC20 token.

2.1 Mayor

The contract, compared to the solution proposed in the final term, has been enriched in order to support the following functionalities:

1. Generalize the previous contract for more than one candidate.
2. Soul implemented as an ERC20 token, the proposal of Angela. This is possible thanks to a **new registration** phase, during which the voter registers himself/herself and receives 100 Soul from the contract. A more in depth explanation about the rational is proposed in section 2.2 SUL Token.

Let's analyse and compare it with the one of the final term, by having an overview per struct, events, modifiers, attributes, and methods.

2.1.1 Structs

The `structs` reported below are the one that have been added or edited.

Candidate substitutes the old variables `naySoul` and `yaySoul` used to keep track of the amount of Soul for a candidate. With the new struct, it's possible to keep track of the number of Soul and the number of votes for a specific candidate. Used to support functionality (1).

Refund as before, used to store data for the refund, still contains the amount of Soul that a voter decides to stake, but it does not contain the boolean doblon any more. In its place there is the candidate address that has been voted. Used to support functionality (1).

Conditions as before, used to keep track of the information about the election (quorum, envelopes casted, envelopes opened, outcome announced), but now contains the address of the winner if any. Used to support functionality (1).

2.1.2 Events

The `events` reported below are the one that have been added or edited.

EnvelopeOpen as before, but now does not contain any more whether the candidate has been voted or not, but it contains the address of who has been voted. Used to support functionality (1).

Registered is a new event emitted during the voter registration phase. More in detail, after the contract sends the Soul to the voter. Used to support functionality (2).

2.1.3 Modifiers

The `modifiers` reported below are the one that have been added or edited.

canRegister is a new modifier defined in order to avoid that a voter can register more than one time, and hence receive more than 100 Soul. Used to support functionality (2).

canVote as before, a voter can vote only if the quorum has not been reached. But now, the constraint to register first has been added. Used to ensure that only a voter with the 100 Soul can actually vote. Used to support functionality (2).

2.1.4 Attributes

The `attributes` reported below are the one that have been added or edited.

SULToken is a new contract that implements the ERC20 token by exploiting the implementations of the OpenZeppelin library, for secure smart contract development. The `Mayor` contracts references the `SULToken` contract address, that needs to be deployed beforehand. In this way, it's possible to have control over it and having flexibility in terms of new possible requirements and constraints, hence ensuring future-proof. Used to support functionality (2).

candidates substitutes the previous address of the only candidate with an array of candidate addresses. Used to support functionality (1).

candidates_state is a new mapping from a candidate address to its state, that is the number of soul and votes received. Used to support functionality (1).

registered is a new mapping from a voter address to a value that states whether is registered or not. In other words, if the person received the Soul in order to express the preference for a candidate. Used to support functionality (2).

2.1.5 Methods

The `methods` reported below are the one that have been added or edited.

constructor as before, takes in input the escrow and the quorum, but now the candidate address has been substituted with an array of candidate addresses to support functionality (1). Moreover, the address of the `SULToken` contracts needs to be passed as an input, to support functionality (2). Compared to the previous solution, is added the initialization of the `candidates_state`.

register it's a new function called only from the external and before casting the envelope. It ensures that the voter register himself to the election in order to receive the ERC20 SUL token used to express the vote. Used to support functionality (2).

cast_envelope as before, but now it's optimized with the external visibility mark in place of the public one. This ensures better performances and less gas consumption, given the fact that Solidity for a public function copies arguments to memory, while for an external one it reads from calldata.

open_envelope as before, but now it's optimized with the external visibility mark in place of the public one. Moreover, it asks the sender for the permission to transfer the SUL tokens from the candidate address to the `Mayor` contract. This is possible thanks to the interface of the ERC20 standard that is `allowance`. If the allowance is granted, then tokens are transferred, and it updates the candidate state with the amount of tokens that the voter stake on him, and it increments by one the number of votes. Used to support functionality (1) and (2).

find_candidate_with_max_soul is a new private view function used to identify and return whether there is a winner or there is a tie situation. In the former case, it's returned even the address of the winner that is the candidate with the maximum number of Soul. In the hypothetical scenario in which there are multiple candidates with the same number of Soul, the one with more votes is the new mayor. Used to support functionality (1).

mayor_or_sayonara now it's optimized with the external visibility mark in place of the public one. In addition, it exploits `find_candidate_with_max_soul` to determine the winner and the `SULToken` contract is used to pay the mayor, the escrow or to refund the voters that lost, depending on the final result. Hence, the logic drastically changed in order to support functionality (1) and (2).

candidates_number is a new external view function used to return the number of candidates participating in the election. It's used by the decentralized application to execute the proper number of calls to get the addresses of the candidates and the mapping `candidate_state`. Used to support functionality (1).

2.2 SUL Token

The contract defines the specification for the SUL token by exploiting the Open Zeppelin ERC20 APIs. This ensures a stable and secure implementation of the standard. The SUL token is defined with zero decimals and when the contract is initialized through the `constructor`, an initial supply must be provided. This is a safe assumption to make, because in any electoral system, the number of voters is known a priori. Thanks to the ERC20 it's possible to mint, transfer tokens and to know the token balance of a specific address and to make an allowance to let a contract execute a transfer of tokens. These are the functionalities used in the Mayor contract 2.1.

3 Decentralized Application

The DApp that acts as the frontend for the contracts deployed over the blockchain, has been implemented and deployed by using the following stack:

- Svelte, for the user interface. This allows to manage the state of the DApp and ensures a fast application compared to other frameworks, thanks to the compilation at build time.
- Semantic UI, a CSS framework for the design.
- Fleek a suite of tools for the deployment of the DApp over IPFS to ensure a full decentralization. Fleek ensures high quality performances, resilience, and high availability. Moreover, a continuous deployment is possible, this allows for sudden changes in cases of malfunctioning during the elections. Finally, it solves the problem related to clients that only have HTTP, by offering a gateway service for IPFS.

3.1 Demo

Let's describe a demo with 3 candidates and a quorum of 2. To replicate it, follow the instructions in section 4.2. A web3 provider is required, such as Metamask, that needs to be connected over the `localhost:8545` network and the 10 demo accounts of ganache must be imported by using the following mnemonic key: `never ceiling bulb miss soon drop feel clarify easy night tank door`.
The candidates are the first three accounts (i.e., 1, 2, 3) provided by ganache. All the others are just voters. The election will proceed as follows:

1. **Account 3** is the third candidate and will act as a voter too. He will vote for himself with sigil 123 and 99 SUL.
2. **Account 4** is not a candidate and will vote for **Account 1** with sigil 124 and 100 SUL.

On the DApp page over `http://localhost:5000/` or `https://valadilene.on.fleek.co/` you will find a guided procedure made of different steps (Fig. 1) in order to express the preference for your favourite candidate. Moreover, the information about the state of the election is reported above (Fig. 2). In the sections below, an example of **Account 3** voting for himself is shown.

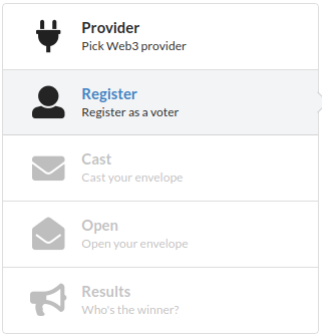

Figure 1: Guided procedure with menu.

Figure 2: Info about the state of the election.

3.1.1 Step 1: Provider

It's the step through which the DApp ask for a connection to the Web3 provider, such as Metamask (Fig. 3).

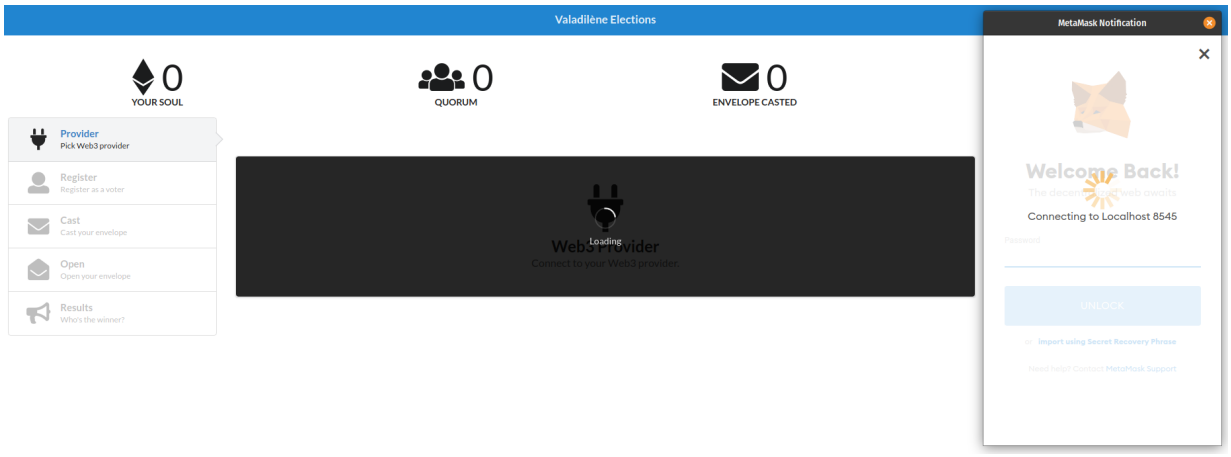

Figure 3: Web3 connection.

3.1.2 Step 2: Register

The step through which the voter will receive 100 SUL to vote. This is done by calling the function `register` and the contract `Mayor` will send the token to the voter address as in Figure 4. Each voter will do the same thing.

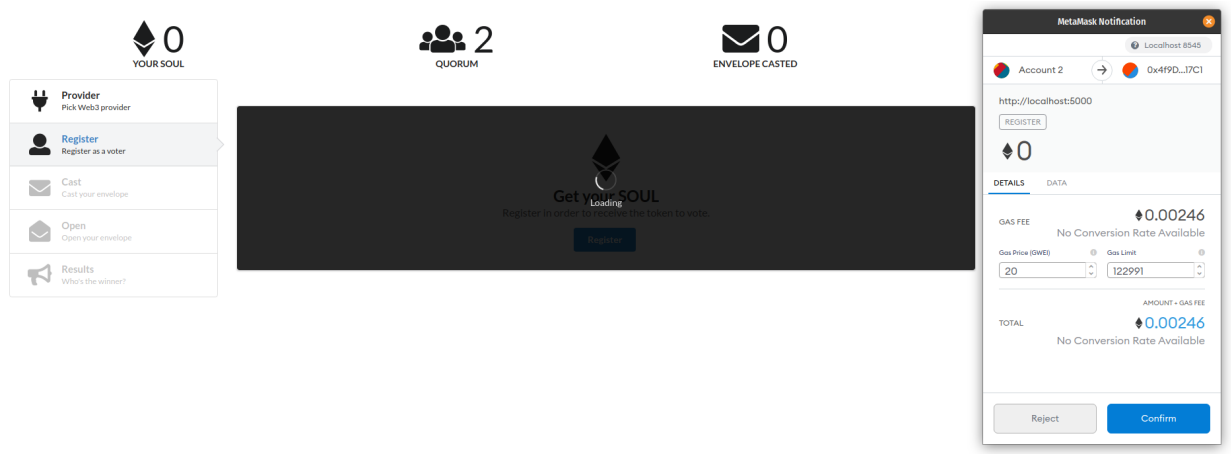

Figure 4: Registration phase.

3.1.3 Step 3: Cast

During this step, the list of candidates is shown with the addresses and random avatars. The voter can cast the envelope by hovering over the button `cast` as in Figure 5. This will open a pop-up through which the voter can write the sigil (depicted with a random image) and the amount of SUL that in this case are 123 and 99 respectively (Fig. 6). This step calls the functions `compute_envelope` and `cast_envelope`. Each voter will cast over the favourite candidate.

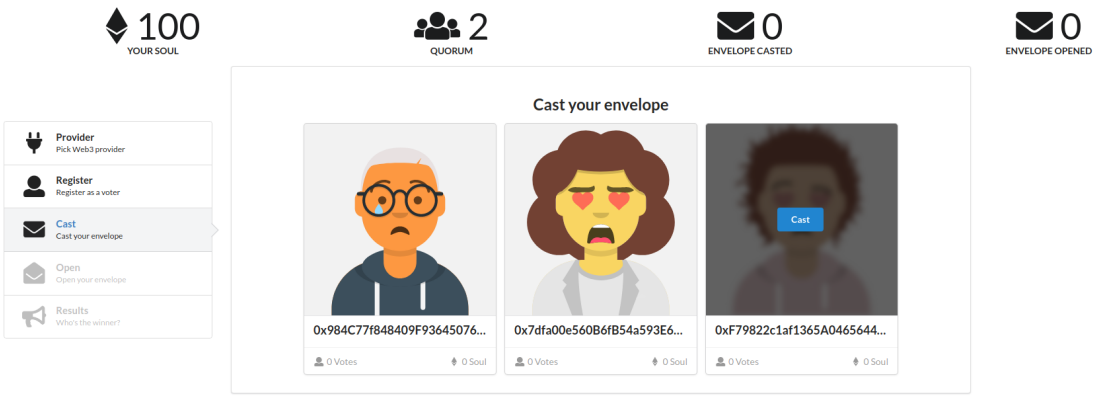

Figure 5: List of candidates that can be cast on.

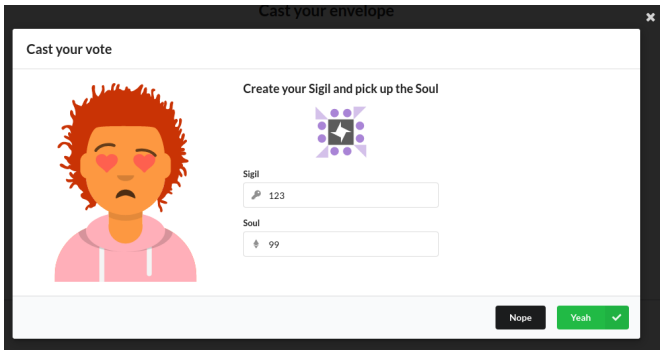

Figure 6: Cast pop-up.

3.1.4 Step 4: Open

In this phase, while all the other voters are in the previous described phases, and hence the number of envelopes casted is less than the quorum, it's not possible to open the envelope. Thus, the voter needs to wait, as in Figure 7. In the meantime, **Account 4** cast an envelope for **Account 1** with sigil 124 and 100 SUL. At this point, the open step is unlocked and **Account 3** can open the envelope for himself by following a procedure similar to what was done in Step 3 as in Figure 8. This will ask for the allowance (Fig. 9, and then it will open the envelope by calling the function `open_envelope`.
Each voter will open over the favourite candidate.

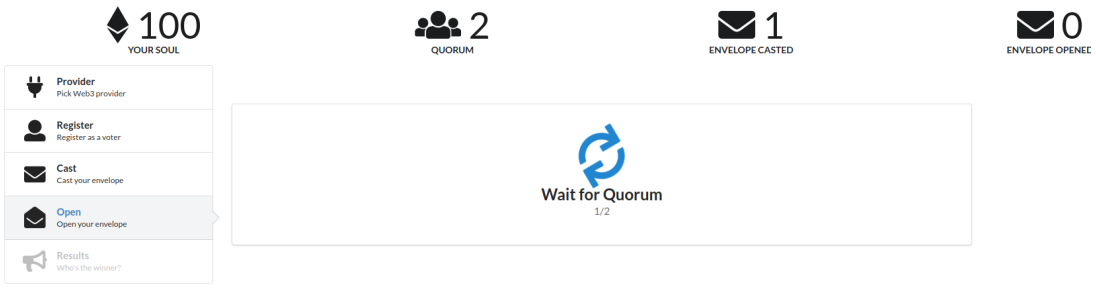

Figure 7: Wait before envelopes casted reaches the quorum.

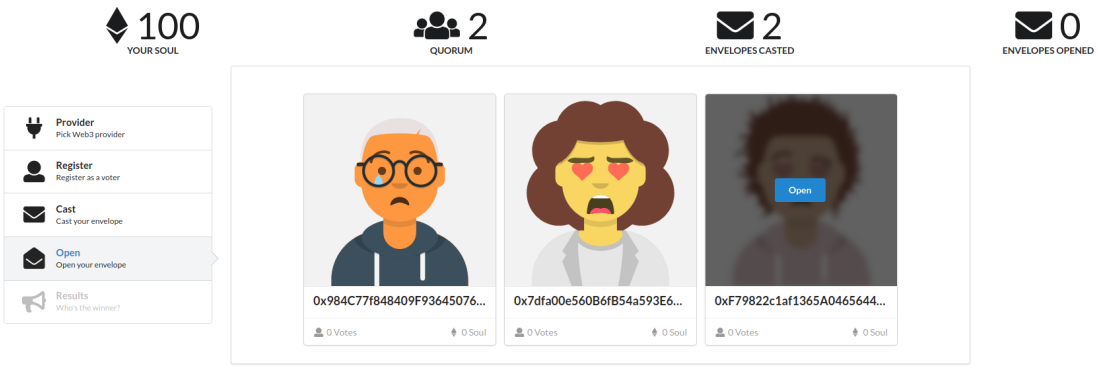

Figure 8: List of candidates on which the envelope can be opened.

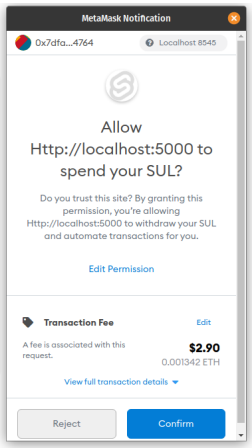

Figure 9: Allowance to let the Mayor contract receive the SUL.

3.1.5 Step 5: Results

During this step, the voter needs to wait that all the other voters open their envelope. In other words, that the number of envelopes opened reaches the quorum (Fig. 10). In the previous figure, moreover, you can see that the SUL of **Account 3** is now 1.
In the meantime, **Account 4** opens the envelope too. At this point, this voter will be in the Results step and given the fact that the quorum now is reached, the results can be computed. For this reason, the function `mayor_or_sayonara` is called (Fig. 11 automatically by **Account 4**).
When the results will be computed, everyone can see who is the winner. From the perspective of **Account 3**, the situation will be the one depicted in Figure 12. Overall, it can be summarized as following:

- Candidate **Account 1** will be the winner with 100 SUL and 1 Vote.
- Candidate **Account 2** loses with 0 SUL and 0 Votes.
- Candidate **Account 3** loses with 99 SUL and 1 Vote. He voted for himself, so now he has 100 SUL back again.
- Voter **Account 4** voted for the winner, so now has 0 SUL.

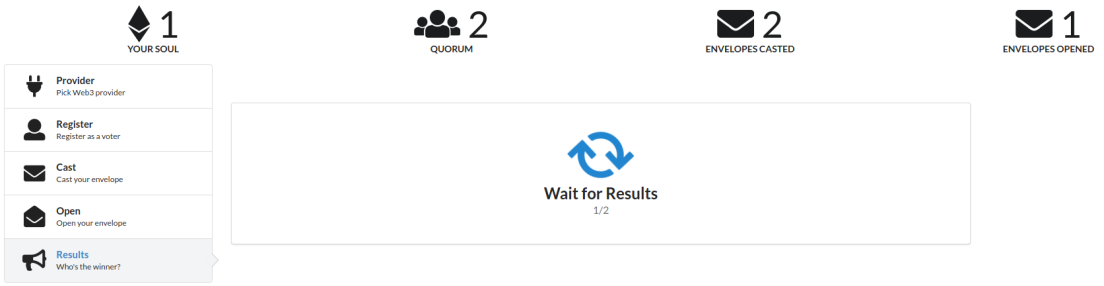

Figure 10: Wait before envelopes opened reaches the quorum.

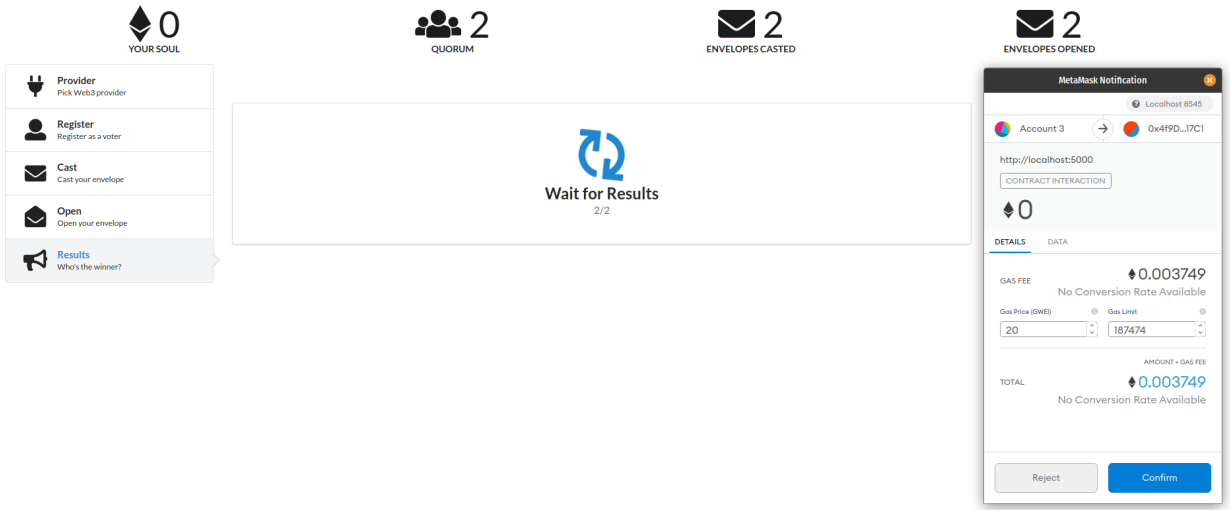

Figure 11: Results are computed by calling *mayor_or_sayonara*.

4 Structure and Instructions

4.1 Structure

The code can be found on github.com/reuseman/voting-dapp. The entire project structure is divided in two main folders, `backend/` that contains the Smart Contracts and `frontend/` that contains the implementation in Svelte of the decentralized application. Let's describe each folder more in detail.

- backend/
  - build/contracts/ contains the compiled version of the smart contracts
  - contracts/ contains the Solidity implementation of the smart contracts `Mayor.sol` and `SULToken.sol`
  - migrations/ contains the JavaScript code used for the deployment logic of the smart contracts
  - node_modules/ contains the dependencies used by the backend
  - test/ contains the test for the smart contracts
- frontend/
  - contracts/ contains the compiled version of the smart contracts and default addresses of the deployed ones
  - public/ contains the compiled code by Svelte for the DApp

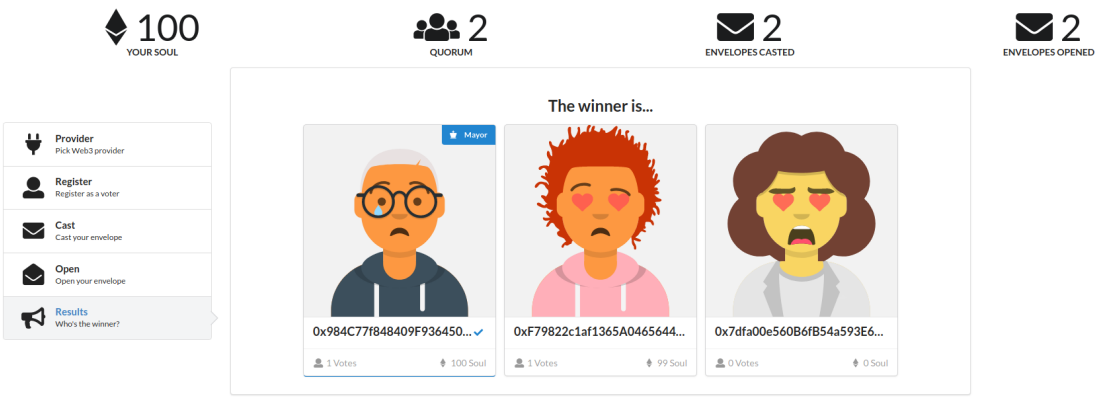

Figure 12: The final result page from the **Account 3** point of view.

- scripts/ required by Svelte
- node_modules/ contains the dependencies used by the frontend
- src/ contains the code for the DApp
  - components/ contains the Svelte components used in routes/
  - routes/ contains the routes through which the voter will pass through during the election

4.2 Instructions

After the code has been downloaded, in order to run the entire applications, the following steps must be executed in order.

1. Deploy the Smart Contracts
   (a) Go in the `backend/` folder
   (b) Install the dependencies with `npm install`
   (c) (Optional) Run the test by executing `npm run test` to ensure that everything works
   (d) In one terminal, execute the command `npm run ganache`. This will launch an instance of ganache-cli with this mnemonic: `never ceiling bulb miss soon drop feel clarify easy night tank door`
   (e) In another terminal, execute the command `npm run deploy`, to deploy the smart contracts
2. Start the DApp by using one of the following methods:
   - Go over the deployed version on IPFS at the following link: https://valadilene.on.fleek.co/. *(Note that this approach could not work if the instructions to deploy the contracts are not followed correctly or the deployment is made multiple times. Addresses of the contracts needs to be known ahead of time and hence the default addresses will not work, if deployment is made on different addresses.)*
   - Build it yourself by following these steps:
     (a) Go in the `frontend/` folder
     (b) Install the dependencies with `npm install`
     (c) In one terminal execute the command `npm run build` and then `npm run start` to launch the server that will be available on `http://localhost:5000/`