APT Project

Stefano Dessì, Andrea Demontis

Introduction

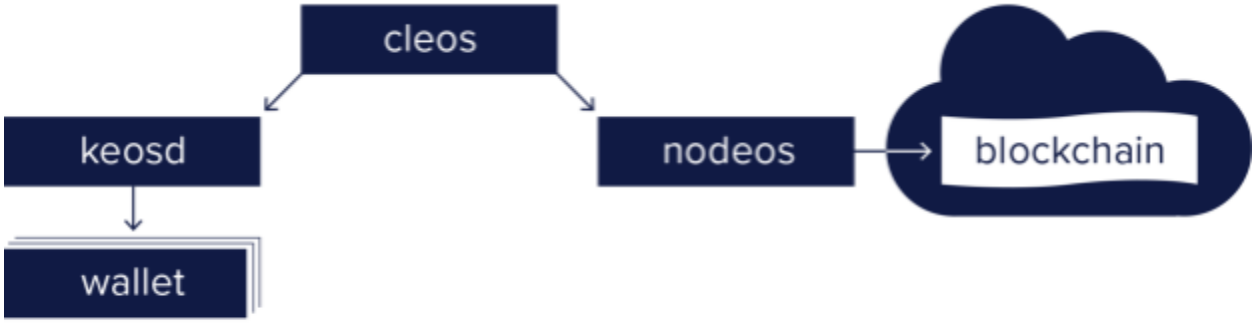
Objectives

The objective of this project is the development of a simple voting decentralized application (dApp) using the EOS.IO blockchain to provide decentralized storage of the election data (candidates and voters) and a way to interact with them (voting), for this purpose a smart contract and unit tests will be written for the dApp. The project also requires the development of the client side part of the application, a way for the end users to see the election's status and cast votes by connecting their account.
A comparison between the development processes and tools of the ethereum blockchain and the ones available with EOS.IO is made throughout the project, this project's development follows closely the one described for a similar ethereum voting dApp, this way we'll be able to highlight some of the differences in technologies, features and developing environment we have encountered.

EOS.IO Blockchain

EOS.IO is a rather new blockchain protocol that aims to solve some of the scalability issues of current blockchains and eliminating user fees.
EOS's delegated proof-of-stake consensus mechanism and overall focus on scalability, at the cost of reduced infrastructural decentralization, contributes to a characteristically very fast transaction time. During the blockchain's launch 21 block producers are decided, the users vote on who they want to produce the blocks by staking their tokens, only 21 nodes are able to produce new blocks at a time.
Since its initial release, in January 31, 2018, it very quickly rose as one of the most popular blockchain protocols, because of its recency, some features are yet to be implemented, and few complete tools exist to help development.

There are three different types of resources available in the blockchain: CPU bandwidth, Network bandwidth, and Memory (RAM).
CPU and network bandwidth are staked and locked up for a certain amount of time for use in transactions and actions, RAM is instead persistent and bought, its price changes based on supply and demand, it can also be traded and its mainly used to store data on the blockchain.
The main native token exchanged in the blockchain is EOS, it can be used to stake CPU and network bandwidth or buy and sell RAM. Staked EOS can be unstaked to receive them back (it takes 72 hours to unstake tokens).



EOSIO Architecture

cleos	cli + eos = cleos	Command line interface to interact with the blockchain and to manage wallets.
keosd	key + eos = keosd	Component that securely stores EOSIO keys in wallets.
nodeos	node + eos = nodeos	The core EOSIO node daemon that can be configured with plugins to run a node. Example uses are block production, dedicated API endpoints, and local development.
eosio-cpp		Part of eosio.cdt, it compiles C++ code to WASM and can generate ABIs.

WebAssembly (WASM) is used to execute user-generated applications and code. The EOSIO C++ toolchain is the recommended way to build smart contracts, third party toolchains are also being developed for Rust, Python, and Solidity. There's currently no official support provided for development on Windows systems.

The Application Binary Interface (ABI) is a JSON-based description on how to convert user actions between their JSON and Binary representations. Developers and users will be able to interact with a contract via JSON using the ABI. Ricardian clauses can also be specified in the ABI file, Ricardian contracts are legally binding agreements, and their use in a blockchain is a new and interesting feature of EOSIO.

Wallets are used to store private keys in an encrypted file and sign transactions. Accounts are required to interact with the blockchain, by means of transactions and actions, they can be described as a collection of authorizations and a way to identify senders or recipients in the blockchain.

Project Workflow

The project has been developed in its entirety on a linux environment, following a remote pair programming like development technique.

Libraries and programs used in this project's development:

EOSIO	1.4.2
EOSIO.CDT	1.3.2
EosFactory	2.2
Eosjs	16.0.9
Scatterjs-core	2.3.8
Scatterjs-plugin-eosjs	1.3.10
Scatter Desktop	v9.6.0

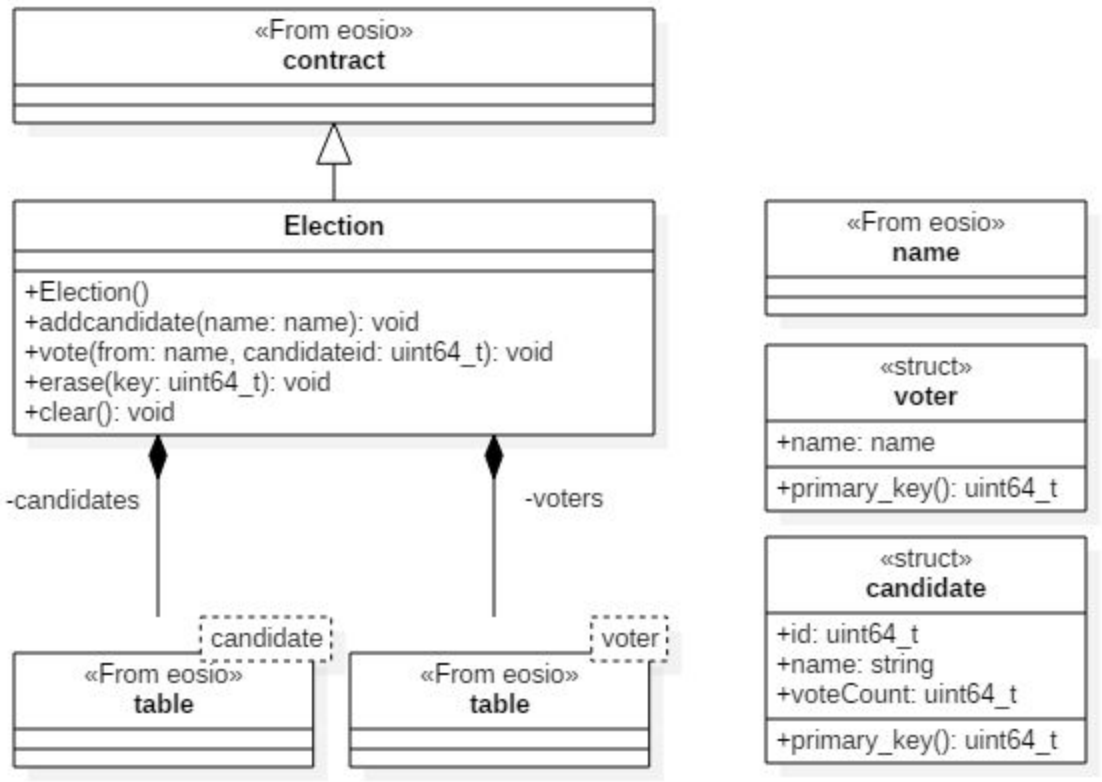
Public Testnets used:

EOS Jungle Testnet	http://dev.cryptolions.io
CryptoKylin Testnet	https://tools.cryptokylin.io

The project has been developed on a Ubuntu 18 system with Atom editor as the only IDE, the contract has also been deployed on the local node, tests are also done on the local node. The UML diagrams have been drawn using StarUML.
The full and complete code has been uploaded on GitHub on this repository: https://github.com/StefanoDessi/EOS-Election

UML Diagrams

Class diagram

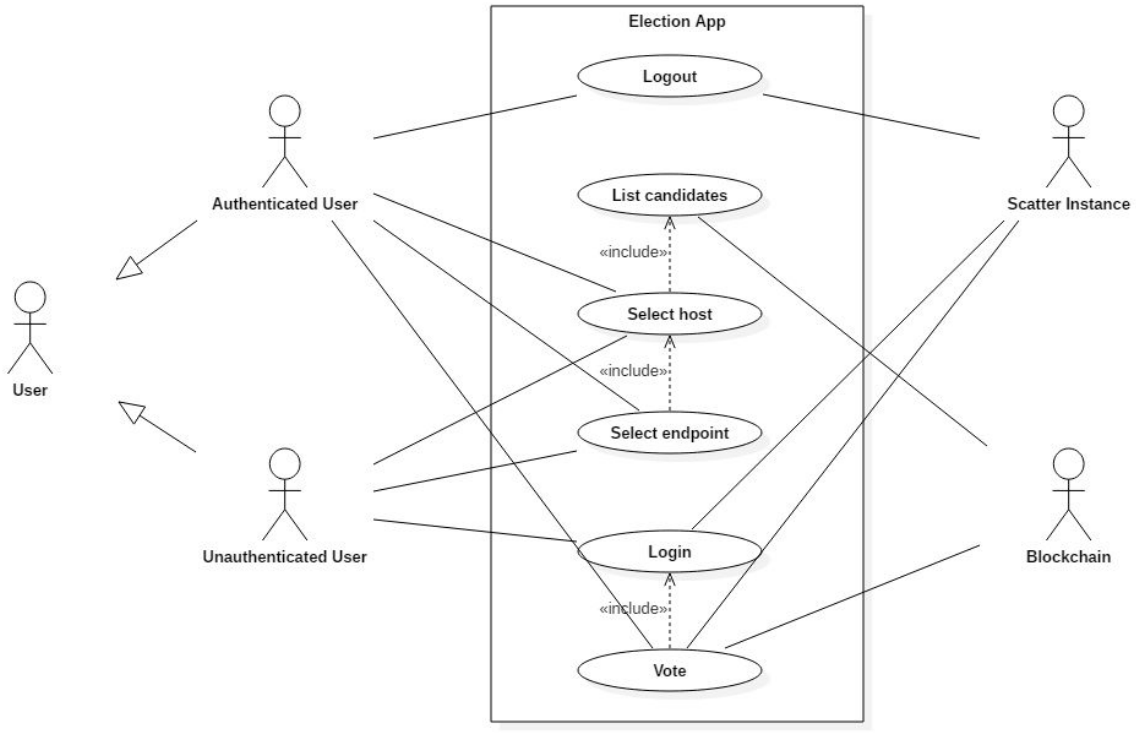


Class diagram describing the election smart contract.

Election is a c++ class representing our smart contract, its parent class is eosio::contract. In the context of our project the most important operations it defines are addcandidate and vote. The data for our election (candidates and voters) is saved in a eosio::table format, the kind of data we save is defined in the structs, eosio::name is a type that identifies accounts.

Additional information on the smart contract will be provided further in this document.

Use case diagram



Use case diagram for the Election app.

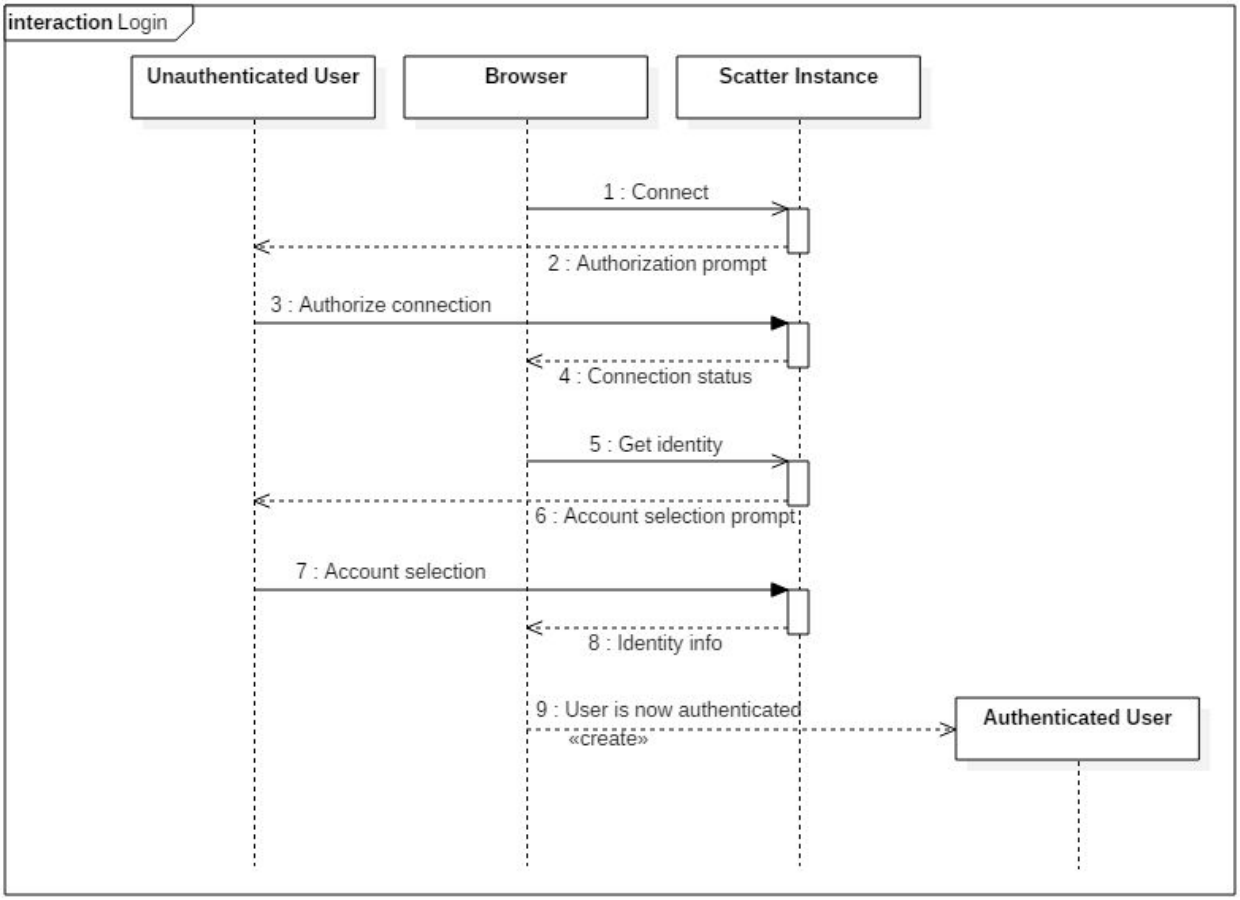
Users can be authenticated or unauthenticated, authentication is handled by the login and logout actions, they are done using the Scatter application.

Users can select the election host account and the endpoint (it specifies the blockchain to connect to), information on the election is then displayed to all types of users, by connecting to the blockchain.

Authenticated users can cast their votes on the election, if they didn't vote already, Scatter is used to sign the transactions, which will then be pushed to the blockchain.

Sequence diagrams

Login

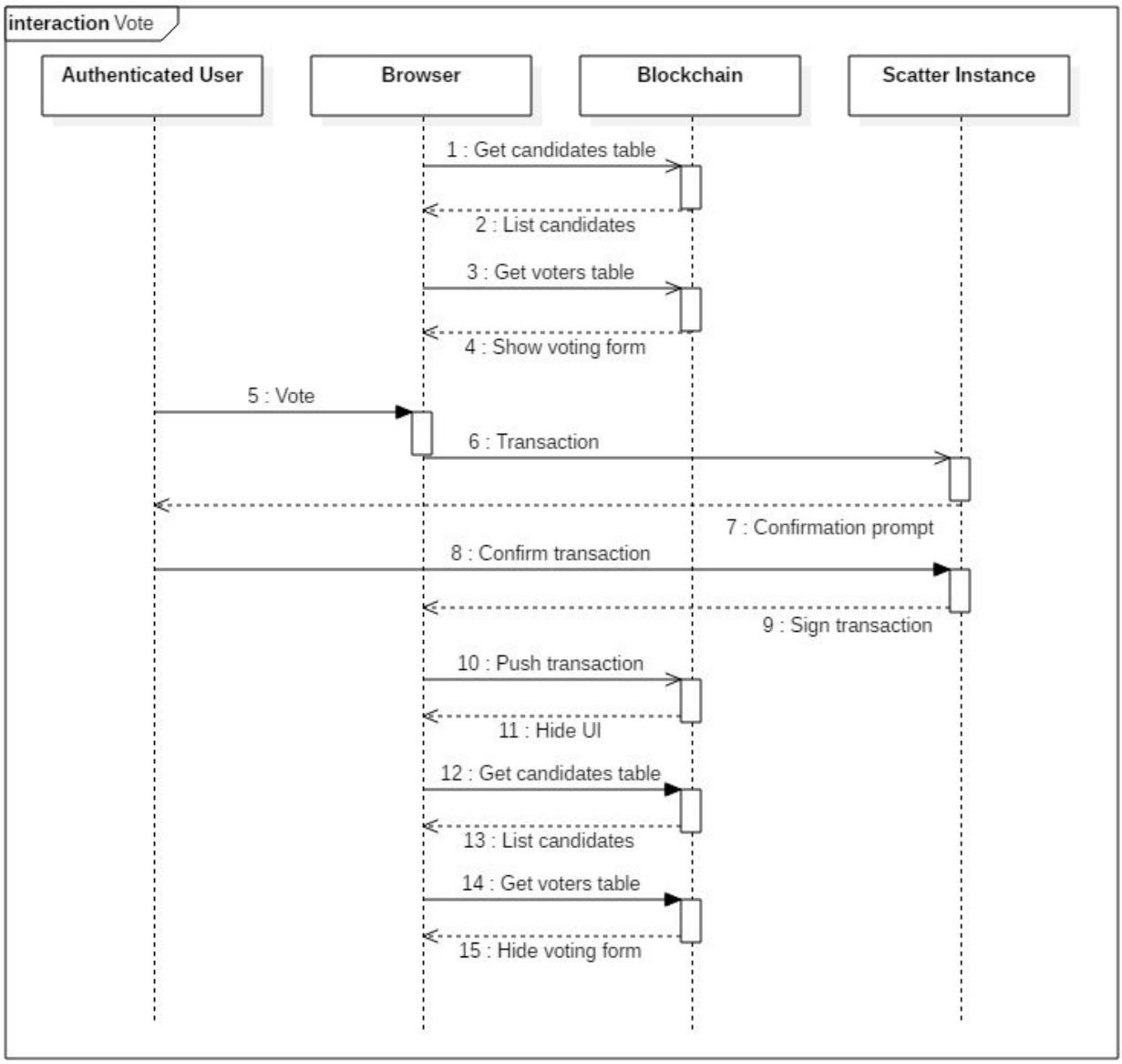


Sequence diagram for the login action in the Election app.

When an unauthenticated user loads the page the browser will try to connect to a Scatter instance, if such instance exists an authorization will need to be provided by the user.

When the connection to Scatter is established successfully the browser will try to get an account, Scatter will show the accounts linked to that particular blockchain and the user can then select one, information on the selected account will be sent to the browser and the user will be authenticated.

Vote



Sequence diagram for the vote action in the Election app.

The browser will list the candidates and show the voting form, assuming the authenticated user didn't vote already. When the user decides to vote, the transaction is sent to the connected Scatter instance to be signed, a prompt asking for a confirmation will be shown to the user, the prompt will contain important information regarding the transaction. After the browser receives the signed transaction it is then sent to the EOS blockchain and the user interface is hidden, waiting for an update. The candidates are listed again and the voting form is hidden because the vote has been casted.

Contract development

Environment

A Docker image is provided by the official tutorial, Docker provides the ability to package and run an application in a loosely isolated environment called a container, the applications run directly within the host machine's kernel. The provided Docker container contains an Ubuntu image with the compiled EOSIO software, but, because interaction with the Docker container is not supported in EosFactory, a framework we'll use for testing later, we decided to directly build and install EOSIO 1.4.2 from source.

The official EOS.IO documentation has been followed to write and deploy the smart contract, initial drafts were deployed on local nodes while final ones were deployed on several public testnets, in order to test the project's functionality on a more realistic setting and have several examples ready.
EOSIO.CDT (EOSIO contract development kit), a toolchain for WebAssembly (WASM), has been used to build the smart contract as it is featured on the official tutorial. The toolchain is built around Clang 7 and contains EOSIO specific optimization.

The language used to write the contract is c++, the input will consists of an header file, defining data types and function interfaces, and a cpp file, which will implement the functions, the output of the build process consists in the previously described ABI and WASM files.

The corresponding ethereum dApp uses the Truffle framework to build the application using Solidity as the main programming language, the already mentioned EosFactory is a possible counterpart to Truffle in this regard, we mainly used it for testing because of its nature, finding it easier to interact with the local node by traditional cleos commands when we needed to perform more obscure system actions.

To deploy the contract to public testnets we had to create an host account using their procedures, in order to receive tokens and buy RAM for storage, then we additionally created several accounts so we could vote with them, we needed to receive tokens once again and stake them to get CPU and NET, in order to cast the votes.

Data persistence

EOSIO provides a set of services and interfaces that enable contract developers to persist state across actions, basically, a Multi-Index database API to enable storing data in the blockchain using smart contracts, patterned after Boost Multi-Index Containers, called eosio::multi_index.

The primary key of the object in the eosio::multi_index container must be a unique unsigned 64-bit integer, which can be derived from the data of the stored type, the objects inside the container are sorted using this primary key in ascending order, secondary indexes can also be defined to order the table differently, a huge advantage compared to some other key/value only blockchains, bi-directional iterators for these indexes can then be retrieved.

We'll first have to define the data structure of our's to-be stored objects using a struct or class, along with a function to retrieve the primary key, if we wanted to define secondary indexes we would need to define key extractor functions for them too.

Table code

```
// Candidates table, stores name and vote count of candidates, assigns
unique id
struct [[eosio::table]] candidate
{
    uint64_t id;
    std::string name;
    uint64_t voteCount;

    uint64_t primary_key() const { return id;}
};
typedef eosio::multi_index<"candidates"_n, candidate> candidate_index;

// Voters table, store names to prevent accounts to vote more than once
struct [[eosio::table]] voter
{
    eosio::name name;

    uint64_t primary_key() const { return name.value; }
};
typedef eosio::multi_index<"voters"_n, voter> voter_index;
```

We decided to store the id, name and current vote count of each candidate in the candidates table, and only store the names of the accounts who already casted their votes, to prevent them from voting more than once.

The ethereum contract used solidity mappings instead, storing the same kind of data but using a key/value hash table like structure.

Add candidate function code

```
/**
 * Adds a candidate to the candidate table, the id is generated using
 * the first available primary key and the voteCount is initialized at 0
 * @param name Name of new candidate
 */
void election::addcandidate(std::string name)
{
  // Require deployer of contract's authority to add candidates
  require_auth(_self);

  // Declare index to access the candidates table
  candidate_index candidates(_code, _code.value);

  // Emplace a new row on the table
  candidates.emplace(_self, [&]( auto& row )
  {
    row.id = candidates.available_primary_key(); //get first available id
    row.name = name; //use the provided name for the candidate
    row.voteCount = 0; //initialize voteCount at 0
  });
}
```

The function to add a candidate uses the emplace method, and adds a row to the candidates table, generating a unique ID (getting the first unused one) and initializing the vote count at 0.

In the ethereum contract the addCandidate function generates a new key/value pair on the candidates mapping and increments the candidates count, we don't need to use a counter for candidates in our smart contract.

Casting votes

Casting a vote requires information on the user who is casting the vote and the ID of the voted candidate.

Before modifying any entry on our stored tables, we need to verify whether the voting procedure is allowed to start or not:

- An authority check is done on the provided account name, assuring that the account who issued the transaction to vote has the correct authority to do so.
- A search on the voters table is done to verify that the account didn't vote before, using the find method, an assert is used to exit the program in case the account is trying to vote twice, with an appropriate error message.
- The provided candidate ID is then checked for its validity, with a search on the candidates table to make sure a candidate with that ID exists, if the search concludes without finding a corresponding entry the assert will exit the program with a custom error message.

The eosio_assert function is used to check if certain conditions are met, and an error message can be specified in case of assertion failure.

If all checks passed with success at this point the voting procedure can start, it consists on:

- Storing the voter's account in the voters table, using the emplace method.
- Updating the candidate's entry in the table to increment the vote count, using the modify method.

Voting function code

```
/**
 * Casts a vote from the specified account to one of the candidates
 * @param from        Voting account's name
 * @param candidateid ID of the voted candidate
 */
void election::vote(name from, uint64_t candidateid)
{
  // Confirm account identity
  require_auth(from);

  // Declare indexes to interact with the tables
  candidate_index candidates(_self, _code.value);
```

```
  voter_index voters(_self, _code.value);

  // Check if user already voted
  auto voter_iterator = voters.find(from.value);
  eosio_assert(voter_iterator == voters.end(), "User already voted");

  // Find candidate row
  auto candidate_iterator = candidates.find(candidateid);
  eosio_assert(candidate_iterator != candidates.end(), "Candidate does not
exist");

  // Add voter to table
  voters.emplace(_self, [&]( auto& row )
  {
    row.name = from;
  });

  // Increment candidate vote count
  candidates.modify(candidate_iterator, get_self(), [&](auto& c) {
            c.voteCount = c.voteCount + 1;
        });
}
```

The ethereum counterpart uses the Solidity function require() in order to check if the address casting the vote is already present in the voters mapping and if the ID is valid, this particular function is used because it will refund the remaining gas when returned, unlike the assert function.

Watching events

In the ethereum dApp, an event is triggered whenever a vote is cast, in order to update the client-side application's user interface in real time by listening to the event and make the necessary changes. This is possible because in the ethereum blockchain transactions have logs to keep track of fired events.

A similar functionality is not yet supported in the EOS blockchain, even though it's supposedly being developed, this means that to update our dApp's user interface we have to choose a different way, meaning we don't need to modify our smart contract further.

Additional functions

To manage the contract, do more testing and get familiar with writing EOS smart contracts we added some additional functions to cleanup the tables, even though they aren't used in the scope of the project. They are management functions that require the contract's host authority to be used.

Erase candidate function code

```
/**
 * Erase candidate function
 * @param key ID of candidate to be erased from the table
 */
void election::erase(uint64_t key)
{
  require_auth(_self);
  candidate_index candidates(_self, _code.value);
  auto iterator = candidates.find(key);
  eosio_assert(iterator != candidates.end(), "Record does not exist");
  candidates.erase(iterator);
}
```

It searches for the row in the candidates table which contains the provided ID and, if it finds it, it uses the erase method on the iterator to delete the entry from the table.

Clear function code

```
/**
 * Clears both candidates and voters tables
 */
void election::clear()
{
  require_auth(_self);

  candidate_index candidates(_self, _code.value);
  voter_index voters(_self, _code.value);

  std::vector<uint64_t> candidateKeys;
  for(auto& item : candidates)
  {
    candidateKeys.push_back(item.id);
```

```
  }
  for (uint64_t key : candidateKeys)
  {
      auto itr = candidates.find(key);
      if (itr != candidates.end())
      {
          candidates.erase(itr);
      }
  }

  std::vector<uint64_t> voterKeys;
  for(auto& item : voters)
  {
    voterKeys.push_back(item.name.value);
  }
  for (auto& key : voterKeys)
  {
      auto itr = voters.find(key);
      if (itr != voters.end())
      {
          voters.erase(itr);
      }
  }
}
```

It retrieves all of the candidates table IDs, storing them in a vector, then for each one of them it calls the erase method. The same procedure is then applied to the voters table, storing the unsigned 64-bit integer corresponding to the names of the voters into the vector and then erasing them one by one.

Testing

Environment

EosFactory has been used to develop tests for the smart contract, it's a Python3-based framework used to develop and test EOS smart contracts. It's meant to be the EOS equivalent of Truffle and Ganache. It's object-oriented, in order to allow to keep track of the state by keeping references to contracts and accounts, it also provides aliases for account names.

EosFactory helps automate some tasks in the form of python scripts, a much needed setting for testing. Some of the tasks that can be automated are: setting up and tearing down the local testnet, building and deploying the contract, creating the accounts and the executing the transactions.

Testing is done interacting with the local testnet's blockchain using EosFactory's functions and using python's own unit testing framework.

EosFactory requires EOS to be directly installed in the machine and doesn't offer support for Docker containers. A few changes have been made to our EosFactory installation in order to support the use of EOSIO.CDT.

Tests

```
def setUpClass(cls):
      SCENARIO('''
      Create a contract from template, then build and deploy it.
      ''')
      reset()
      create_master_account("master")

      COMMENT('''
      Create test accounts:
      ''')
      create_account("alice", master)
      create_account("carol", master)
```

In the setUpClass method the local node is reset, in order to have a clean state before performing the tests, and the test accounts to be used, along with the master account, are created.

Building and deploying contract

```
def test_01(self):
      COMMENT('''
      Create, build and deploy the contract:
      ''')

      create_account("host", master)
      contract = Contract(host, "election")
      contract.build()
      contract.deploy()
```

The host account for the election is created, and the contract itself is built and deployed, errors on the code of the smart contract and EosFactory's configuration will make this test fail.

Add candidate test

```
def test_02(self):
      COMMENT('''
      Test add candidate:
      ''')
      t = host.table("candidates", host)
      self.assertEqual(len(t.json["rows"]), 0)
      host.push_action(
            "addcandidate", {"name": "Bob"}, permission=(host,
Permission.ACTIVE))

      t = host.table("candidates", host)
      self.assertEqual(t.json["rows"][0]["id"], 0)
      self.assertEqual(t.json["rows"][0]["name"], "Bob")
      self.assertEqual(t.json["rows"][0]["voteCount"], 0)
      self.assertEqual(len(t.json["rows"]), 1)

      COMMENT('''
      WARNING: This action should fail because only the host can add
candidates
```

```
      ''')
      with self.assertRaises(MissingRequiredAuthorityError):
            host.push_action(
                  "addcandidate", {"name": "Wally"}, permission=(carol,
Permission.ACTIVE))
      self.assertEqual(len(t.json["rows"]), 1)
```

To test the functionality of the addcandidate function we first get the candidates table and make sure it's empty at the beginning, then we add a candidate "Bob", using the host account's permissions.
After performing this action we get the table again and verify that each value of the added row is at it should be, checking that only one row is present in the table after the function is called.
Lastly, we verify that only the host account can add candidates, we make sure that trying to add a candidate with carol's account is not permitted, raising a MissingRequiredAuthorityError, which is caught using assertRaises, we then check that the failed action didn't modify anything, with the table's length still being one.

Cast vote tests

```
def test_03(self):
      COMMENT('''
      Test add vote:
      ''')
      t = host.table("voters", host)
      self.assertEqual(len(t.json["rows"]), 0)

      COMMENT('''
      WARNING: This action should fail due to authority mismatch!
      ''')
      with self.assertRaises(MissingRequiredAuthorityError):
            host.push_action(
                  "vote", {"from": alice, "candidateid": 0},
permission=(carol, Permission.ACTIVE))
      t = host.table("voters", host)
      self.assertEqual(len(t.json["rows"]), 0)

      host.push_action(
            "vote", {"from": alice, "candidateid": 0}, permission=(alice,
Permission.ACTIVE))
```

```
      t = host.table("candidates", host)
      self.assertEqual(t.json["rows"][0]["id"], 0)
      self.assertEqual(t.json["rows"][0]["voteCount"], 1)

      t = host.table("voters", host)
      self.assertEqual(t.json["rows"][0]["name"], alice.name)
      self.assertEqual(len(t.json["rows"]), 1)

      COMMENT('''
      WARNING: This action should fail because alice already voted!
      ''')
      with self.assertRaises(Error):
            host.push_action(
                  "vote", {"from": alice, "candidateid": 0},
permission=(alice, Permission.ACTIVE))
      COMMENT('''
      WARNING: This action should fail because the candidate doesn't exists
      ''')
      with self.assertRaises(Error):
            host.push_action(
                  "vote", {"from": carol, "candidateid": 1},
permission=(carol, Permission.ACTIVE))
```

We make sure that the voters table is empty at the start of the test, we check every scenario when we assume the transaction should fail.
We try to vote as Alice using Carol's identity, making sure it results in a raised MissingRequiredAuthorityError.
Afterwards we cast a legitimate vote as Alice for the candidate with ID: 0 (Bob), we make sure that Bob's row in the candidates table is updated with an incremented voteCount, then we check the previously empty voters table to make sure one row, and only one, is added containing Alice's name.
The test then tries to vote again as alice and to vote for for an invalid candidate, both actions should raise errors.

WebAPP development

Environment

Our application primarily uses eosjs to interface with the blockchain, eosjs is a general purpose javascript library for the EOS blockchain, it provides APIs to fully interact with the blockchain, by creating accounts, reading blocks, pushing transactions, even compiling and deploying contracts if necessary. It's the EOS equivalent of web3.js. We need a way to provide private keys securely to sign transactions, to do this eosjs is not enough and we'll also need Scatter.

Scatter stores private keys and provides a way to sign transactions securely without requiring a direct input of private keys into the browser. Scatter is meant to work with a variety of blockchains, so it requires an additional plugin to specifically work with eos and interface with eosjs.
The Scatter application is available as a desktop application, mobile application or even as a browser plugin. It will be the EOS equivalent of Metamask.

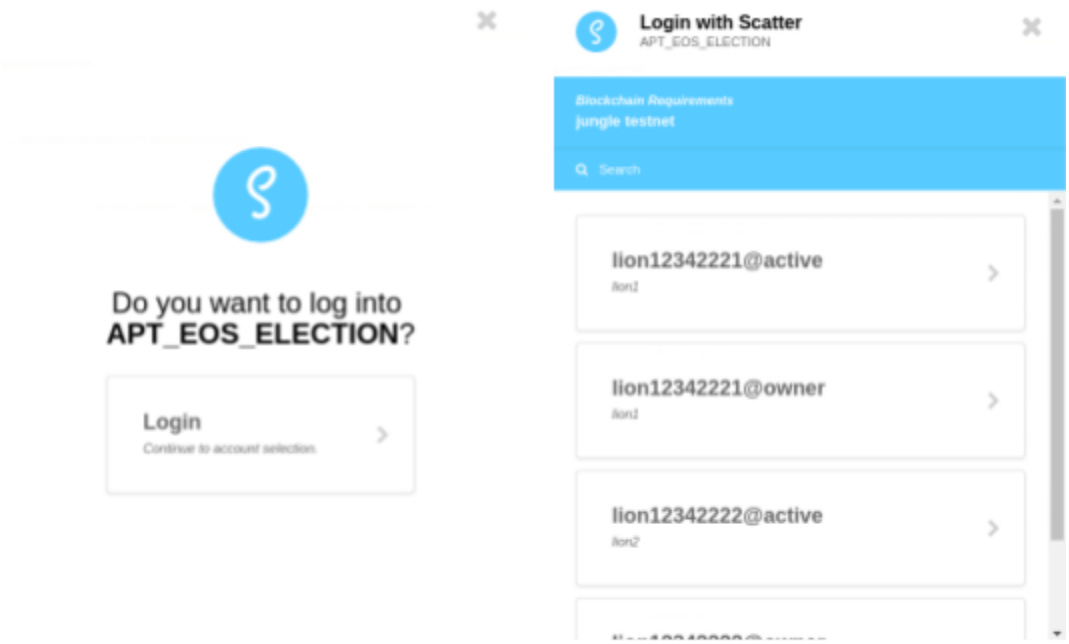
The front end of the application is very similar to the ethereum counterpart, the html part only differences are in the used libraries and the addition of a few forms to select the endpoint and host and a logout button.

Account connection

```
init: function()
  {
    ScatterJS.scatter.connect("APT_EOS_ELECTION").then(connected => {

    // User does not have Scatter Desktop, Mobile or Classic installed.
    if(!connected) return false;

    App.scatter = ScatterJS.scatter;

    network =  {
        blockchain:'eos',
        protocol:'http',
        host: App.endpoint.host,
        port: App.endpoint.port,
        chainId: App.endpoint.chainId
```

```
    }
    App.scatter_network = network;

    App.scatter.getIdentity({accounts: [network]}).then(identity => {
        if(identity.accounts && identity.accounts.length == 1)
        {
          App.account = identity.accounts[0].name;
          App.contracts.Election = App.scatter.eos(network, Eos, {})
          return App.initContract();
        }
    });
    });
}
```

Using ScatterJS we connect to the user's Scatter, specifying "APT_EOS_ELECTION" as the name of our application. Scatter needs the endpoint's data (host, port and chainID) in order connect to it. Scatter will ask the user if he wants to connect with the application, and will list accounts from the specified blockchain if there are any. After an account is chosen and the connection has been established, the application will appear in Scatter's list of connected applications, where it can be managed by the user.



Scatter prompt, allowing connection and selecting account.

We use the getIdentity() function to get the chosen account's name and have a way to connect with eos. The user's account will be used to cast votes, while the election's host account is used to retrieve information about the given election (listing candidates).

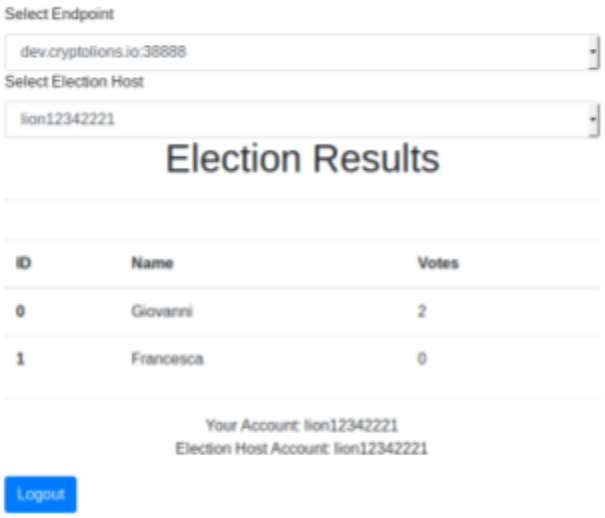
Render function and candidate listing

```
render: function()
  {
    var electionInstance;
    var loader = $("#loader");
    var content = $("#content");
    loader.show();
    content.hide();
    $("#accountAddress").html("Your Account: " + App.account + "<br>
Election Host Account: " + App.host);

    // Load contract data

App.contracts.Election.getTableRows(true,App.host,App.host,"candidates").th
en(
      function(result)
      {
        current_state = result;

        var candidatesResults = $("#candidatesResults");
        candidatesResults.empty();

        var candidatesSelect = $('#candidatesSelect');
        candidatesSelect.empty();

        rows = result.rows;
        for (var i = 0; i < rows.length; i++)
        {
          var id = rows[i].id;
          var name = rows[i].name;
          var voteCount = rows[i].voteCount;

          // Render candidate Result
          var candidateTemplate = "<tr><th>" + id + "</th><td>" + name +
"</td><td>" + voteCount + "</td></tr>";
          candidatesResults.append(candidateTemplate);
```

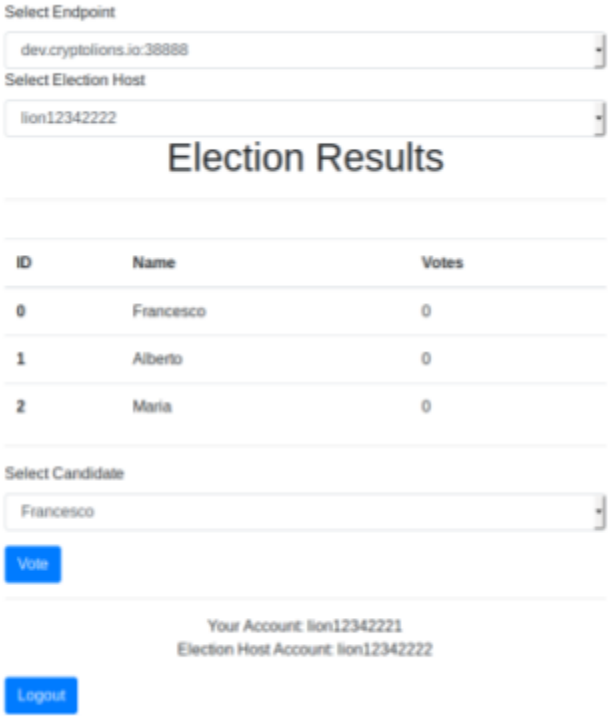
```
          // Render candidate ballot option
          var candidateOption = "<option value='" + id + "' >" + name +
"</option>"
          candidatesSelect.append(candidateOption);
        }

return(App.contracts.Election.getTableRows(true,App.host,App.host,"voters",
1,App.account,App.account+1));
      }
    ).then(function(hasVoted)
    {
      // Do not allow the user to vote if its name is found on the table
      if(hasVoted.rows.length)
      {
        $('form').hide();
      }
      else
      {
        $('form').show();
      }
      loader.hide();
      content.show();
    }).catch(function(error) {
      console.warn(error);
    });
  }
```

The render function shows the connected account and election host account names in the user interface, then it gets the candidates list from the blockchain (using the host's account, there's no need to sign this action so the private keys of the host are not needed), after emptying the current table, it adds rows for each candidate showing the id, name and vote count for each one.

Each candidate is also added as an option on the select html form responsible for casting votes, this form is hidden when the account already voted, to verify this, another request is forwarded to the blockchain, asking the row in the voters table which should contain the account's name, if no rows are returned the form is hidden.



User interface with hidden voting form.

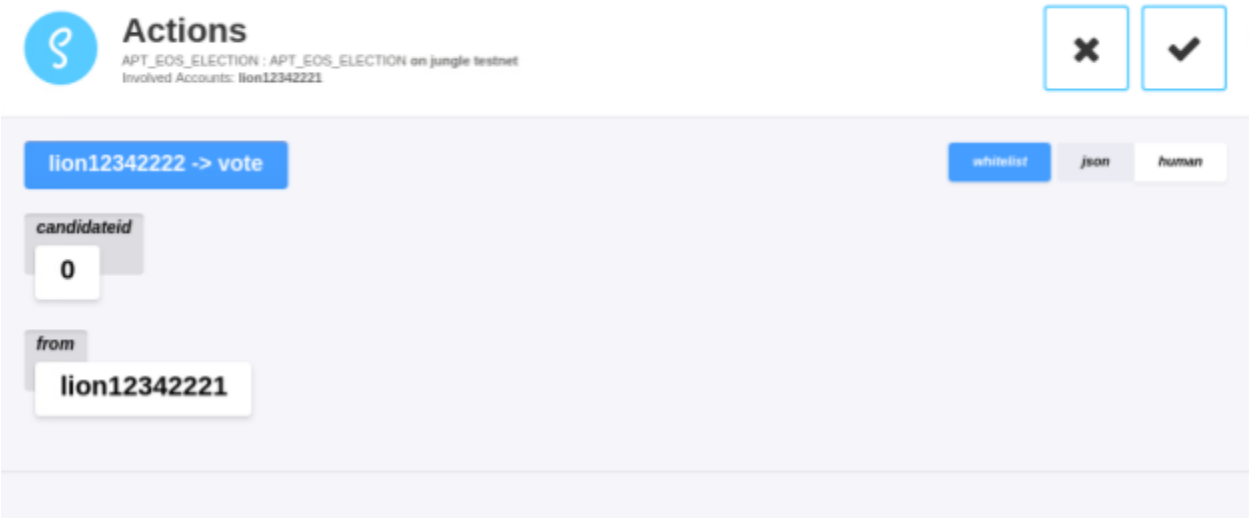


User interface with shown voting form.

Casting votes

```
castVote: function()
  {
    var candidateId = $('#candidatesSelect').val();
    App.contracts.Election.transaction(
      {
        actions: [
          {
            account: App.host,
            name: 'vote',
            authorization: [{
              actor: App.account,
              permission: 'active'
            }],
            data: {
              from: App.account,
              candidateid: candidateId
            }
          }
        ]
      }
    ).then(function(result) {
    $("#content").hide();
    $("#loader").show();
    }).catch(function(err) {
        console.error(err);
      });

  }
```

We retrieve the candidate's ID from the form, then we send a transaction using eosjs, because the voting transaction will need to be signed, Scatter will ask for the user to confirm the transaction.



Prompt asking for the user's permission to send the transaction.

Updating UI

```
listenForEvents: function()
  {
    App.updateListener = setInterval(function() {

App.contracts.Election.getTableRows(true,App.host,App.host,"candidates").th
en(
        function(result)
        {
          if(JSON.stringify(result)!=JSON.stringify(current_state))
          {
            App.render();
          }
        });
    }, 10000);
  }
```

To update the user interface we check for updates on the state of the candidates table on a set amount of time, each time we get the table we save the current state, if there are differences with the new values we retrieved, the render function is called again, updating the user interface.

The ethereum's counterpart triggers an event when votes are cast on the smart contract side, it listens to these events being called and used the watch function to update the user interface. As we already discussed, this functionality is not yet present in the EOS blockchain, so we just directly check for updates on a set amount of time.

Another possible solution could require the use of Demux. Demux uses a back-end infrastructure pattern for sourcing blockchain events in order to deterministically update queryable databases, it helps storing and retrieving indexed data and expanding on the query interface provided by EOSIO by being supported by a traditional Mongo or Postgres database, the database can then be interfaced with the application with REST.

Other differences

```
hostChange: function()
  {
    App.host = $("#hostSelect").val();
    App.render();
  },
  endpointChange: function()
  {
    App.endpoint = App.endpoints[$("#endpointSelect").val()];
    $("#hostSelect").empty();
    for(j=0;j<App.endpoint.hosts.length;j++)
    {
      $("#hostSelect").append(new Option(App.endpoint.hosts[j]));
    }
    App.host = $("#hostSelect").val();
    if(App.scatter) App.scatter.forgetIdentity();
    clearInterval(App.updateListener);
    App.init();
  },
  logout: function()
  {
    App.scatter.forgetIdentity();
    $("#content").hide();
    $("#loader").show();
    clearInterval(App.updateListener);
    App.init();
  }
```

We decided to give an option in the user interface to change the testnet and the host more easily, to show multiple elections quickly, adding two more select elements in the html file (hostSelect and endpointSelect) and the code to control the effect of changing their values, using the onchange attribute of the new select elements. We also provided a logout function, in order to disconnect from Scatter and quickly change accounts to vote with, for testing purposes.

Example structure of endpoint data:

```
endpoints: [
    {
      host: "dev.cryptolions.io",
      port: 38888,
      chainId:
"038f4b0fc8ff18a4f0842a8f0564611f6e96e8535901dd45e43ac8691a1c4dca",
      hosts: ["lion12342221","lion12342222"]
    },
    {
      host: "api.kylin.alohaeos.com",
      port:80,
      chainId:
"5fff1dae8dc8e2fc4d5b23b2c7665c97f9e9d8edf2b6485a86ba311c25639191",
      hosts: ["lion12342221"]
    },
    {
      host: "127.0.0.1",
      port: 8888,
      chainId:
'6cbecff836a9fa60da53bf97a0a180103b2e76041d4414693d11bf39e2341547',
      hosts: []
    }
```

Endpoints have hosts containing example elections we specified for each of them, changing the endpoint will also require an update to the host selector in the application (in order to list that endpoint's hosts), we'll also disconnect from Scatter and reconnect again asking for a different account. We call the render again when the election's host is changed.

Conclusion

Even though the tools that an EOS.IO developer has at its disposal are still rather new and limited because of their recency, they are being developed and expanded upon quickly. Some features of the blockchain are themselves missing and being worked on.

Recency can also be a big problem when it comes to troubleshooting issues. The depth of the ethereum developing tools and documentation built over years is a considerable advantage, but the currently available EOSIO tools still provide a way to fully develop a distributed application.

EOS.IO is increasing in popularity very rapidly, showing it posses huge potential and clever advertising focusing on it's innovative features and fast block time. As it gets more popular more tools and features will naturally become available for developers.

Solidity support has also been mentioned as a possible future feature of the EOS VM, as most smart contracts developers are familiar with Solidity this will be an important step, some other interesting features for developers (to make it easier to listen to events happening on the blockchain for example) are also in development for future releases.

Ethereum and EOSIO are currently the top two platforms for decentralized apps by market cap size, the biggest difference is the focus of Ethereum in decentralization and the focus of EOSIO in scalability. Other differences are in transaction fees, Ethereum requires to use gas while EOSIO requires to stake tokens that the user gets back, meaning no transaction fees. Developers can choose their preferred platform based on their needs (scalability, decentralization) and programming language (c++, Solidity).

Tools overview table

Ethereum	**EOS.IO**	Purpose
Truffle	EosFactory	Smart contract deployment and testing.
Metamask	Scatter	Secure transaction signing.
Web3.js	Eosjs	Javascript API library.