Voting Poll Smart Contract Manual

Blockchain based smart contract

Contents

Introduction:

What is a blockchain?

- A good analogy to understand what a blockchain is, is the bitcoin currency. Normally when you try to withdraw from the bank on your computer, you use a browser and talk to a central server over a network. All of the code of this web application lives on the central server, and all the data lives in a centralized database. The centralized database means that there is only one location where the data is stored, which makes the database more prone to attacks and if there is a failure, then the data cannot be retrieved. However, with a blockchain the data is spread out in a chain of blocks which makes the data decentralized. The blocks are also called nodes and are identifiable by a certain hash, generated by a hash algorithm. Blocks on the blockchain also have transaction history, the blockchain is a ledger which records all transactions and keeps up to date info of who owns what and how much they own. The data on the blockchain is immutable and is publicly available to everyone. Meaning, anyone can add to the blockchain because it is a public ledger. In conclusion, The blockchain is a database in itself, and the nodes talking to each other makes it a network.

Why blockchain?

- a blockchain is especially useful in an Election. Our smart contract code that we write will be stored as a node in the blockchain permanently and will govern the rules of our election. Making sure the voter can only vote once, and the right candidates are displayed, as well as authorizing the voter to be able to vote. Having a permanent transaction recorded on the public ledger of a blockchain can ensure that votes are 100% correctly tallied. We don’t have to worry about a server being hacked because the blockchain is a decentralized server with tons of nodes and each node contains a copy of the permanent list of transactions.

System Diagram:

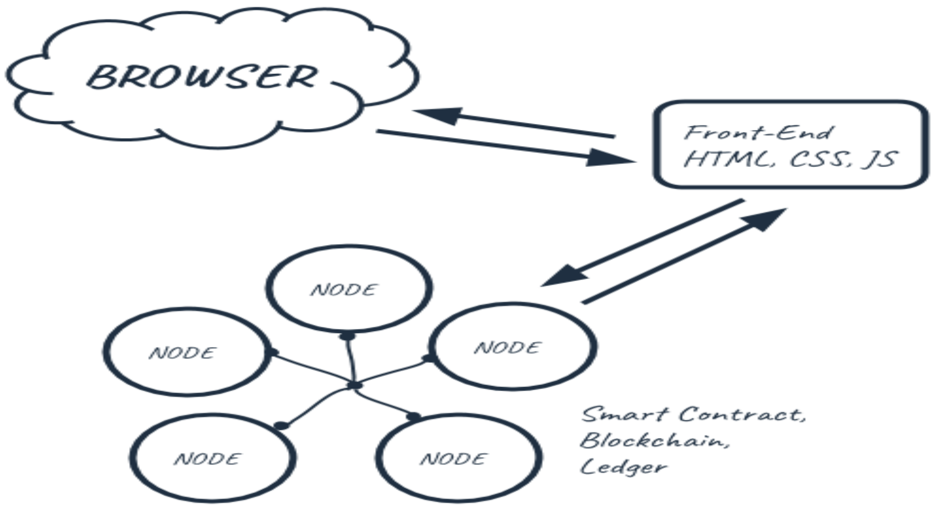


Fig 1: system diagram for a decentralized application using blockchain technology

This picture shows how a decentralized application would work. The front-end client would be talking to a network of nodes to store our data instead of a conventional database like Hadoop, SQL, etc. This network of nodes called the blockchain also has smart contracts, and the public ledger of accounts, the smart contracts hold the execution logic for the DAPP. Lastly you use the web browser to interact with the web application and the blockchain network as well through the web3 provider "Ganache".

Simple Smart contract tutorial:

Before we make a voting poll smart contract, let's start with a simple storage contract. Go to remix.ethereum.org.

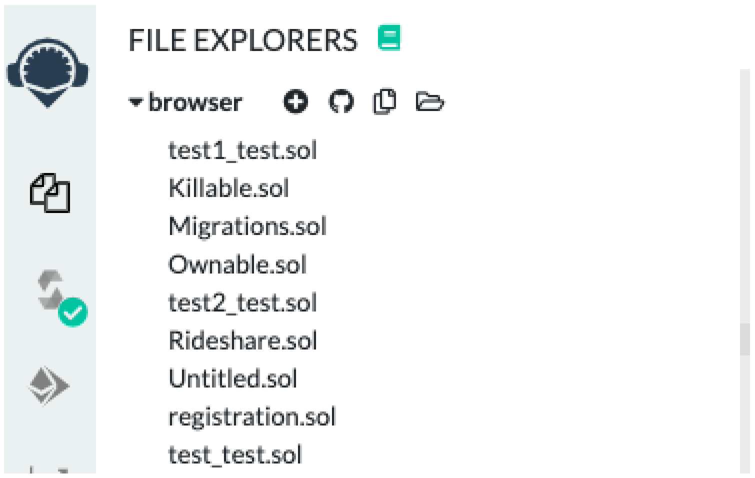
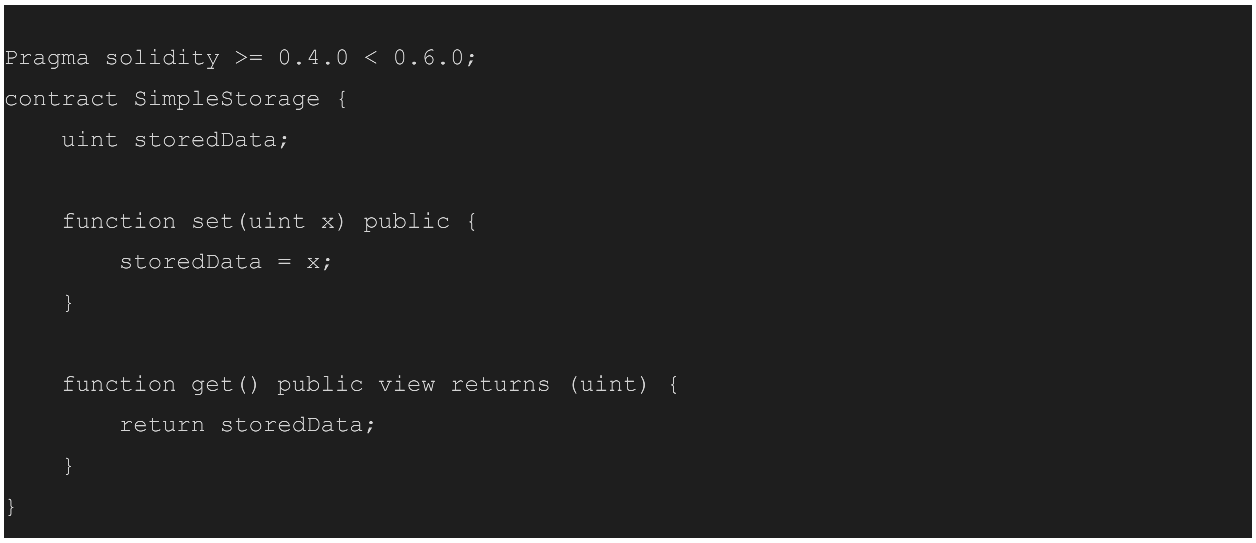


Fig 2: the remix IDE user interface for the different files

Looking at Figure 2, press the small + button that is to the right of the word "browser". Make a new contract called "SimpleStorage.sol". and copy and paste the following code (fig.3) into the contract. The code contains one variable called "storedData" and two functions that set the value of the stored data and then returns the value.



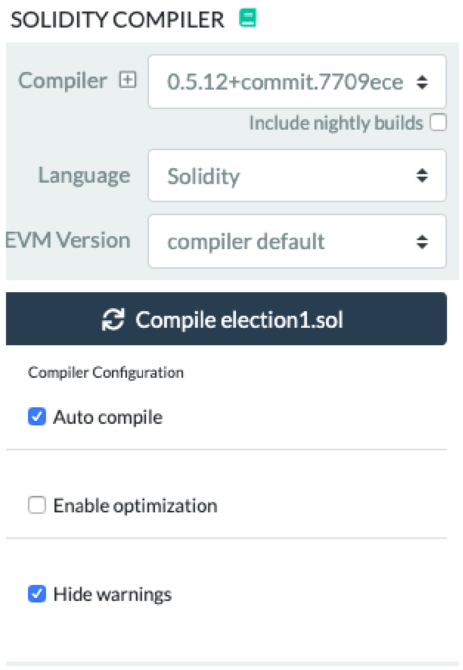
```
Pragma solidity >= 0.4.0 < 0.6.0;
contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Fig 3: the code for the "SimpleStorage" smart contract

After copying and pasting the code above, go to the compile tab where you can compile the code. Press on the "compile" button to compile the code, as shown in figure 4. The compiler button is right underneath the file directory button



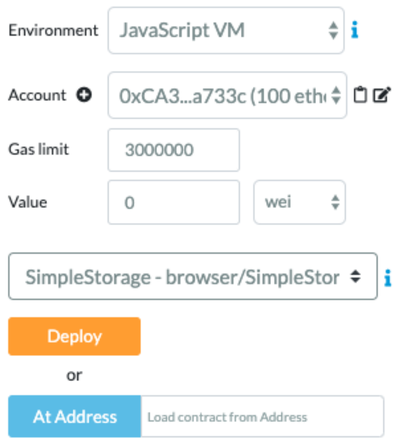
SOLIDITY COMPILER

Compiler: 0.5.12+commit.7709ece
Include nightly builds ☐
Language: Solidity
EVM Version: compiler default

Compiler Configuration
☑ Auto compile
☐ Enable optimization
☑ Hide warnings

Fig 4: the section of the website to compile your code

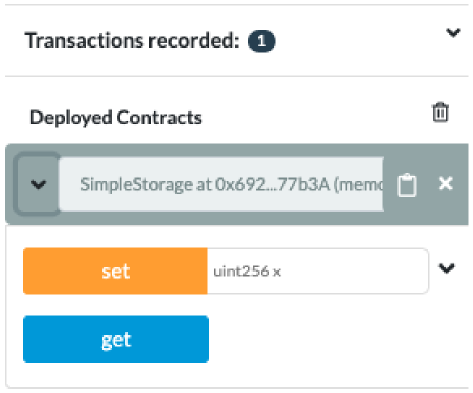
After you compile your code you can then run the code, to run the code press on the button underneath the compiler. Refer to Figure 5 to see the transaction screen.



Environment: JavaScript VM
Account: 0xCA3...a733c (100 eth
Gas limit: 3000000
Value: 0 wei
SimpleStorage - browser/SimpleStor
Deploy
or
At Address: Load contract from Address

Fig 5: the section of the website to run your code

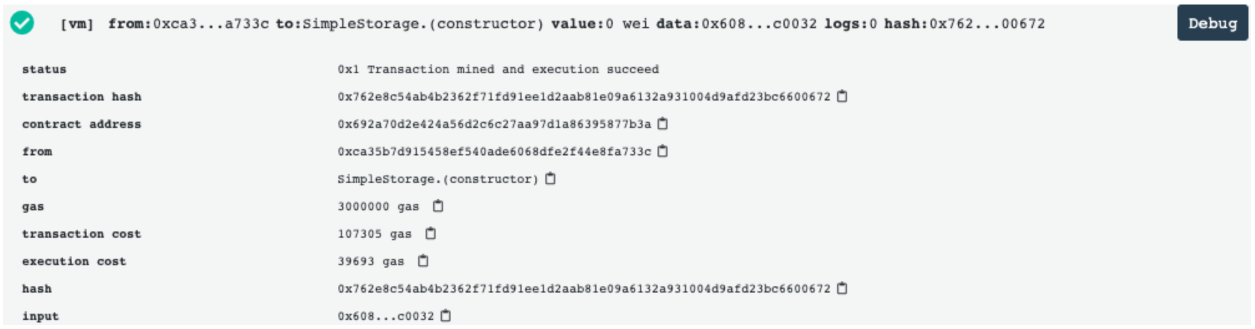
Once you arrive here, press on "deploy" to deploy your smart contract. The other settings can stay the same for now. After you press deploy look underneath to see the contract is created.



Transactions recorded: 1
Deployed Contracts
SimpleStorage at 0x692...77b3A (memo
set uint256 x
get

Fig 6: the section of the website to simulate the simple storage

Once you are here, type in any value into the box next to the "set" button and then press on the "get" button to return this value. This value is an integer value and behind the scenes is stored in a blockchain, permanently. Go to the right to see all the metrics about the transaction that you just appended to the blockchain.



```
[vm]  from:0xca3...a733c to:SimpleStorage.(constructor) value:0 wei data:0x608...c0032 logs:0 hash:0x762...00672   Debug

status                      0x1 Transaction mined and execution succeed
transaction hash            0x762e8c54ab4b2362f71fd91ee1d2aab81e09a6132a931004d9afd23bc6600672
contract address            0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
from                        0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to                          SimpleStorage.(constructor)
gas                         3000000 gas
transaction cost            107305 gas
execution cost              39693 gas
hash                        0x762e8c54ab4b2362f71fd91ee1d2aab81e09a6132a931004d9afd23bc6600672
input                       0x608...c0032
```

Fig 7: Transaction details

click on the arrow to the right of the "debug" button, this will show the complete details of the transaction that just happened. Don't worry about the "gas" cost, this will be explained later.

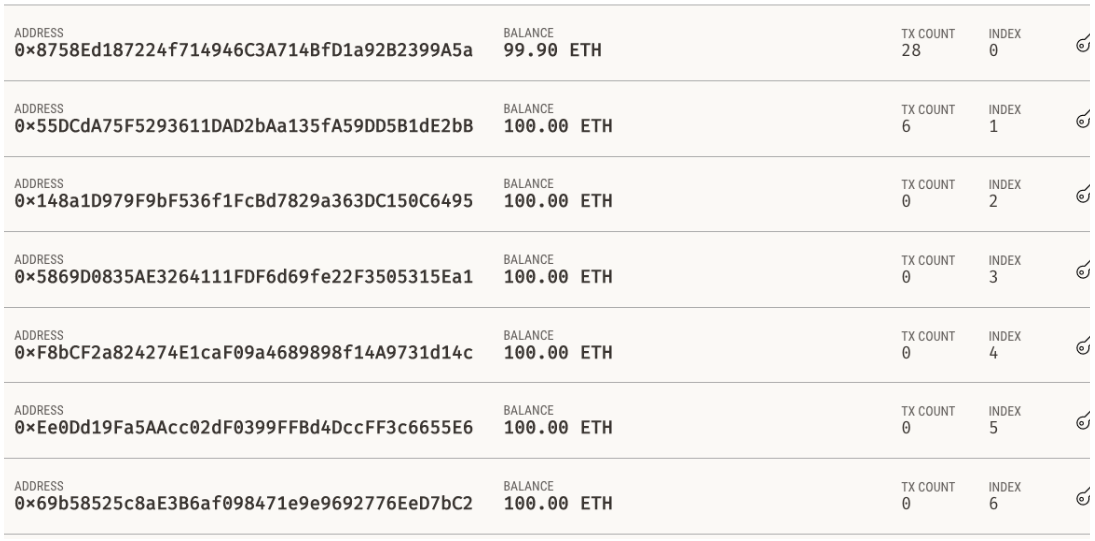
STEP 1: download the following software below

Set up the environment:

Ganache - Local in memory blockchain for development purposes. Installed at https://www.trufflesuite.com/ganache. This application gives you test accounts and private keys to use with Metamask. Each account holds 100 fake ether.

Remix IDE – all the development and testing will be done on an online IDE called remix IDE, the website is https://remix.ethereum.org/.

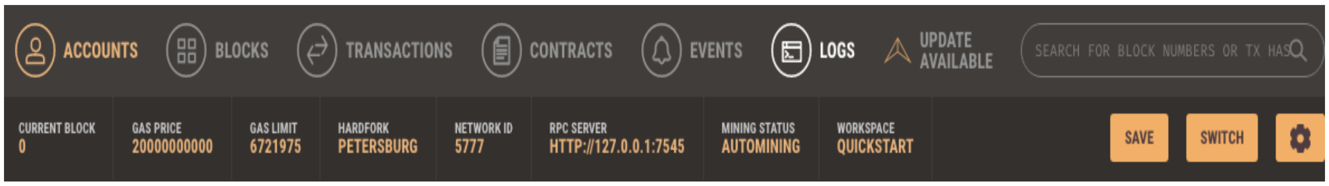
Step 2: run Ganache The first step is to start the Ganache application, then press "quick start" and it should look like this with your accounts:



ADDRESS	BALANCE	TX COUNT	INDEX
0×8758Ed187224f714946C3A714BfD1a92B2399A5a	99.90 ETH	28	0
0×55DCdA75F5293611DAD2bAa135fA59DD5B1dE2bB	100.00 ETH	6	1
0×148a1D979F9bF536f1FcBd7829a363DC150C6495	100.00 ETH	0	2
0×5869D0835AE3264111FDF6d69fe22F3505315Ea1	100.00 ETH	0	3
0×F8bCF2a824274E1caF09a4689898f14A9731d14c	100.00 ETH	0	4
0×Ee0Dd19Fa5AAcc02dF0399FFBd4DccFF3c6655E6	100.00 ETH	0	5
0×69b58525c8aE3B6af098471e9e9692776EeD7bC2	100.00 ETH	0	6

Fig 8: The virtual Ganache test network accounts, with fake ether in them.

make note of the RPC server address, which is the server port and address that you have to use for testing your local network.



CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	MINING STATUS	WORKSPACE
0	20000000000	6721975	PETERSBURG	5777	HTTP://127.0.0.1:7545	AUTOMINING	QUICKSTART

Fig 9: The top of the Ganache application showing the Server ip and port.

Step 3: open the Remix IDE and copy and paste the finished code.

1. Make sure you have the previous software downloaded and make note of your RPC server value, the one in the above example is 127.0.0.1:7545, the Mining status is automatically set to “automining” which means the blocks are automatically mined as a transaction.

Coding the smart contract:

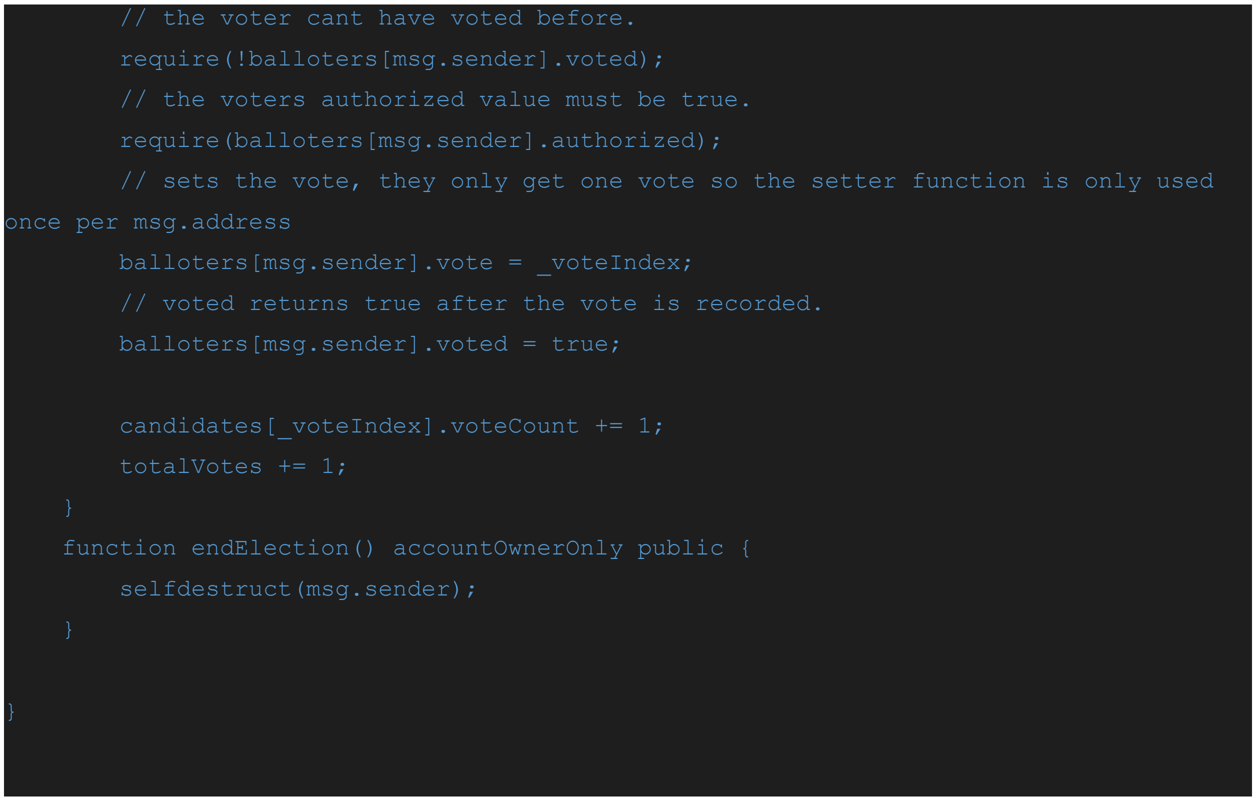
We continue coding the smart contract by writing code for the candidates that will run in the election. The candidate for the election is stored in a struct variable, in order to store multiple candidates, and the different attributes for each candidate.

Below is the complete code for the smart contract:

```
pragma solidity >= 0.4.0 <  0.6.0;

// smart contract which will simulate an election of whichever your choosing.

contract election {
    struct Voter {
        // authorized means that they are authorized to vote in the election, similar
to how a voter needs to be a US citizen.
        bool authorized;
        // to check if they already voted or not, returns false if havent voted and
true if they have
        bool voted;
        // keep track of which candidate was voted for.
        uint vote;
    }
    // struct "Candidate" is a container object and can be declared as a variable.
    struct Candidate{
        // keep track of the name,age and voteCount of the candidate.
        string name;
        uint voteCount;
        uint age;
    }
    // candidate array to hold the string and votecount and age
    Candidate[] public candidates;
    uint public totalVotes;
    // this is used later when setting the msg.address to the "AccountOwner" variable
    address public accountOwner;
    // whichever election you choose to be in, enter with " " as a string variable
    string public electionName;
    // gives a specific address to voters in an array similar to a dictionary.
    mapping(address=> Voter) public balloters;
    // constructor for when you make an election, parameter is the name of the
election
    constructor(string memory _name) public {
        accountOwner = msg.sender;
        electionName = _name;
    }
    // this function makes sure that the person who calls the function has the same
address as the account that called the function..
     modifier accountOwnerOnly() {
        require(msg.sender == accountOwner);
        // this line exhibits, continuing the rest of the code if the require passes.
        _;
    }
    // add a "candidate" to the candidates array
    function addCandidate(string memory _name, uint _age) accountOwnerOnly public {
        candidates.push(Candidate(_name,0,_age));
    }
    // returns number of candidates
    function getNumCandidate() public view returns (uint) {
        return candidates.length;
    }
    // authorize a voter so the person can vote
    function authorize( address _person )  public {
        balloters[_person].authorized = true;
    }
    function getAge(uint _candidateID) public view returns (uint) {
        return candidates[_candidateID].age;
    }
    // set the "Account" you specify at the top for your msg.address value, and
voteIndex as the paramater for the specific CandidateID you want
    function vote(uint _voteIndex) public {
```



```
        // the voter cant have voted before.
        require(!balloters[msg.sender].voted);
        // the voters authorized value must be true.
        require(balloters[msg.sender].authorized);
        // sets the vote, they only get one vote so the setter function is only used
once per msg.address
        balloters[msg.sender].vote = _voteIndex;
        // voted returns true after the vote is recorded.
        balloters[msg.sender].voted = true;

        candidates[_voteIndex].voteCount += 1;
        totalVotes += 1;
    }
    function endElection() accountOwnerOnly public {
        selfdestruct(msg.sender);
    }


}
```

Fig 10: complete code for the Election smart contract

Copy the code above and head over to the remix IDE website, which the link is provided earlier.

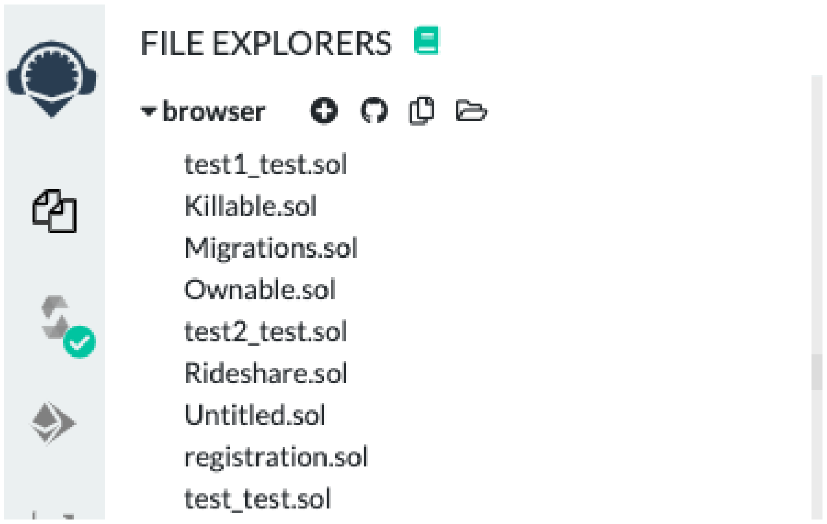


Fig 11: Remix IDE panel

On the panel, click on the little plus button and type "election.sol" for the name of the smart contract, then paste the code you copied above into the smart contract.

The complete code will be explained right now:

- After the structs, we require a variable to store the voters. This is done with a Mapping variable in Solidity. The mapping type variable is an hash which stores a key-value pair. In our example the key is an unsigned integer value, meanwhile the mapping value is a candidate. We can now look up a specific voter with their address.

```
  // Store accounts that have voted
   mapping(address => Voter) public balloters;
```

Fig 12: mapping variable declared in the election.sol smart contract

- The next declaration is a variable to hold the number of candidates, which is also an unsigned integer.
- If we try to look up a candidate number higher than the number of candidates we have, a default value will be returned for the candidate. We have to use the counter cache to figure out how many candidates there are.

```
// Store Candidates Count
    Candidate[] public candidates;
```

Fig 13: unsigned integer variable is declared to hold the count of candidates

The function "addCandidate" takes one string variable as well as an integer variable, which is representative of the candidates name and age. The candidates count cache is updated by one for each candidate. A new Candidate struct is added to the mapping variable. This function is made with the private accessor, because the function is only used within the contract.

```
function addCandidate(string memory _name, uint _age) ownerOnly public {
        candidates.push(Candidate(_name,0,_age));
    }
```

Fig 14: The "addCandidate" function for the smart contract.

The two candidates get added from the constructor, which are the two candidates from the Voting poll. These candidates are added into the constructor from the "addCandidate" function.

```
constructor(string memory _name) public {
        owner = msg.sender;
        electionName = _name;
```

```
    }
```

Fig 15: the “constructor” function is created when the contract is made

Next, a integer variable is created which holds the total count of votes that have been recorded.

```
// Store accounts that have voted
   uint public totalVotes;
```

Fig 16: integer variable is declared to keep track of how many of the balloters who have voted.

A Vote function is then added into the smart contract, The candidates vote count is increased in the below vote function. The vote function accepts the candidate’s ID as an argument. The visibility for the function is public, so that an external ID can call it. The function adds the account that performed the vote into the voters mapping. This function is called with the global variable “msg.sender” which keeps track of the specific address of an account.

The require statements of the vote function make sure the function is called properly. The voter using the contract cannot already be in the voters mapping, because they are allowed only one vote. The require statements check that the candidate’s ID is valid, also it checks that they have been authorized to vote.

```
function vote(uint _voteIndex) public {
        require(!voters[msg.sender].voted);
        require(voters[msg.sender].authorized);

        voters[msg.sender].vote = _voteIndex;
        voters[msg.sender].voted = true;

        candidates[_voteIndex].voteCount += 1;
        totalVotes += 1;
    }
   }
```

Fig 17: vote function which takes in a candidate ID

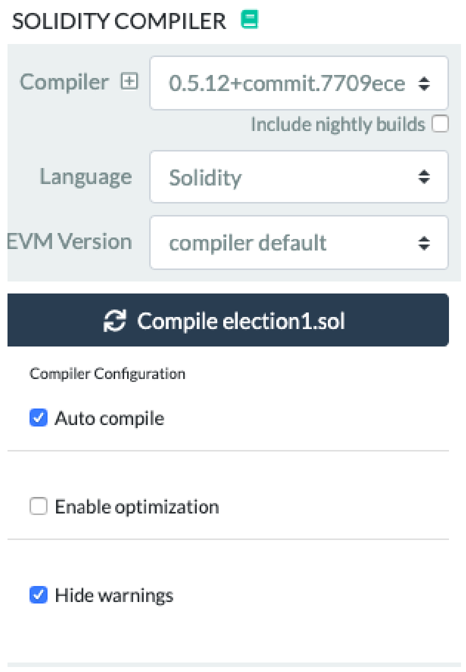
The last part of the contract is the triggering of an event whenever a vote is made in the application. The event below is triggered in our vote function itself.

```
function authorize( address _person )  public {
        voters[_person].authorized = true;
    }
```

Fig 18: authorize function which takes in a voters address

The authorize function is used to authorize the voters address to be able to vote, in a real election there are requirements to be able to vote in the first place. In order to simulate the election as realistic as we can, we add this function.

Step 3: Testing the code on Remix IDE



SOLIDITY COMPILER

Compiler: 0.5.12+commit.7709ece

Include nightly builds ☐

Language: Solidity

EVM Version: compiler default

Compiler Configuration

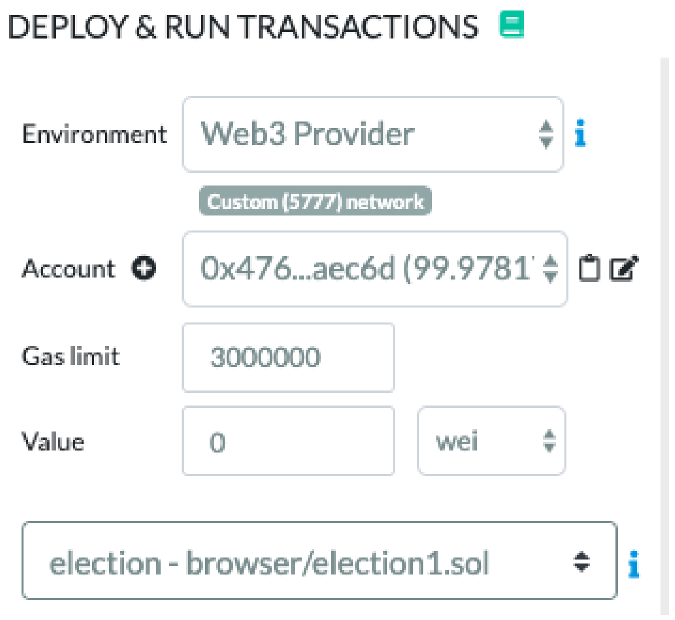
☑ Auto compile

☐ Enable optimization

☑ Hide warnings

Fig 19: The Solidity compiler on the Remix IDE website

Head to the compiler section and then compile your smart contract, by pressing on the button which is in Figure 19. After the smart contract is compiled, head over to “deploy and run transactions” section which is right underneath and will allow you to run your code.

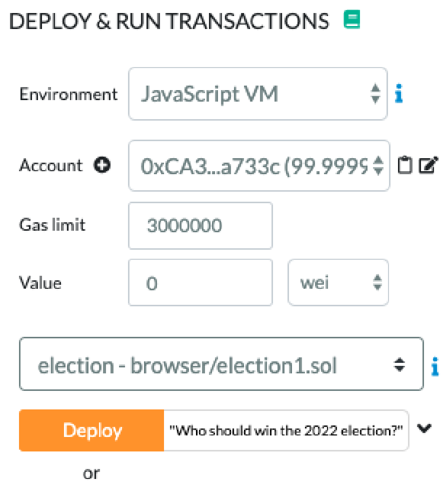


DEPLOY & RUN TRANSACTIONS

Environment: Web3 Provider
Custom (5777) network
Account: 0x476...aec6d (99.9781
Gas limit: 3000000
Value: 0 wei
election - browser/election1.sol

Fig 20: section where you can deploy your smart contract and record a transaction

Next to deploy the actual smart contract, first change the Environment to “Web3 Provider” this will connect your local blockchain “Ganache” to the remix IDE. You have to type in the port for the Ganache network. Next, type in the name for the election that you would like to host, for this project we are going to make an election where voters can vote for different candidate’s as their choice for the presidential election. After you type in the value make sure that the value Is surrounded by quotation marks because it is a string literal.

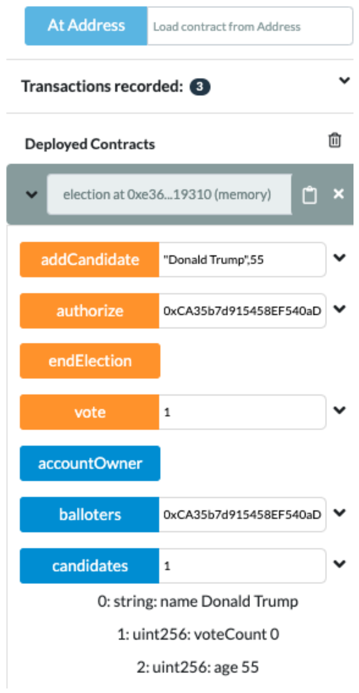


DEPLOY & RUN TRANSACTIONS

Environment: JavaScript VM
Account: 0xCA3...a733c (99.9999
Gas limit: 3000000
Value: 0 wei
election - browser/election1.sol
Deploy "Who should win the 2022 election?"
or

Fig 21: button to deploy your smart contract to the blockchain

As we can tell, the smart contract is deployed. Next head over to the section which is in figure 22.



"Donald Trump",55

0xCA35b7d915458EF540aD

1

0xCA35b7d915458EF540aD

1

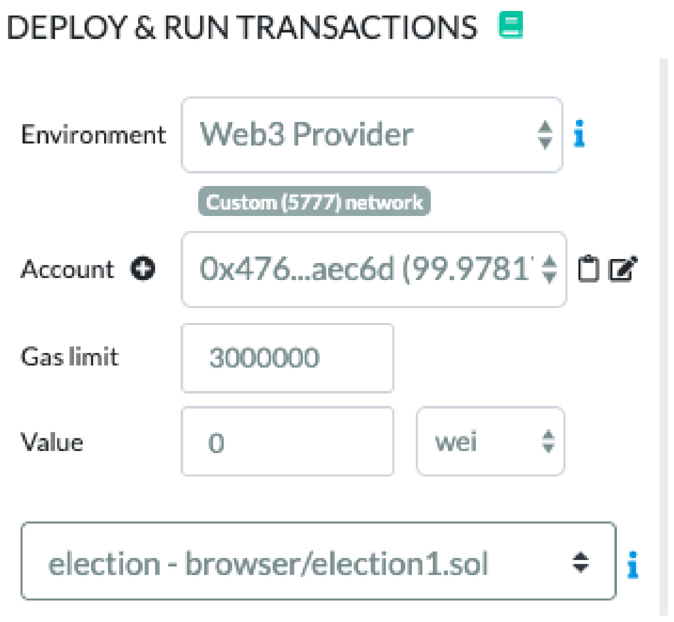
0: string: name Donald Trump

1: uint256: voteCount 0

2: uint256: age 55

Fig 22: Deployed smart contract interface

This is where we can actually test our smart contract, and simulate an election. Imagine you are making an election to vote for your choice of the presidential candidate. In the "addCandidate" section, type the name for an candidate that voters can vote for and then put a comma and enter a integer value for the age. Telling your voters the candidates age is a useful statistic, than can influence the vote. Next, press the "addCandidate" button. This will initialize the candidates to be used for your election simulation. Copy your account address by pressing on the clipboard button which is indicated in the picture below.


DEPLOY & RUN TRANSACTIONS

Environment: Web3 Provider

Custom (5777) network

Account: 0x476...aec6d (99.9781'

Gas limit: 3000000

Value: 0 wei

election - browser/election1.sol

Fig 23: deploy and run transactions page with the specific account address.

Copy and paste that address into the “authorize” box and then click on the “authorize” button to allow this address to vote in the election.

The orange buttons are functions that write data to the blockchain and modify data, while the blue buttons are smart contract functions that simply return data to the user. With the specific account address you can enter the ID for a candidate and then press the “vote” button to vote for. If you then enter the candidate ID into the “candidates” button and press on the button, it will return the vote count for the candidate. This shows that vote is permanently recorded on the blockchain network. Keep playing around with the smart contract and Remix IDE interface’s other functions and try and guess what they will do from the smart contract code provided.

Critical Analysis

Learning Outcomes:

The knowledge and skills that we used in the project that we acquired in earlier courses include:

Programming:
- Solidity

Other:
- Security (Hashing)
- Terminal commands
- Software Design concepts

The new knowledge and skills we acquired from this project include:
- Smart contracts, Blockchain, Solidity language syntax, Ethereum networks understanding,