Decentralized Voting Application

Ethereum & Solidity

This manual is an end-to-end guide to building a simple voting decentralized voting application using Smart Contracts.

The purpose of this demonstration is:

1. Setting up the development environment of Ethereum
2. The process of writing a smart contract using Solidity.
3. Implementing and interacting with the contract through the nodejs (terminal) console.
4. Deploying the contract as a localhost web application that displays vote counts and allows one to vote for the candidates.

The demonstration was performed on a Mac OS X. The same will work on Linux(Ubuntu) as well.

1. Setting up the development environment

The demonstration does not deploy the application on a live Blockchain. A Blockchain simulator has been used – ganache CLI.
Below (snippet) are the commands to install ganache, web3js and start the test Blockchain on the terminal (macos).
Web3js version used is 0.20.0 (please install the same version – other versions are unstable)

First check if you have the updated **npm** and **node** versions

```bash
Kunals-MacBook-Pro-2:~ kunalmehta$ node -v

v8.11.1

Kunals-MacBook-Pro-2:~ kunalmehta$ npm -v

5.6.0
```

If not, install the latest npm version using:

```bash
Kunals-MacBook-Pro-2:~ kunalmehta$ npm install npm@latest -g
```

After completing installation, create a directory (anywhere) for you application

```
Kunals-MacBook-Pro-2:~ kunalmehta$ mkdir Voting
```

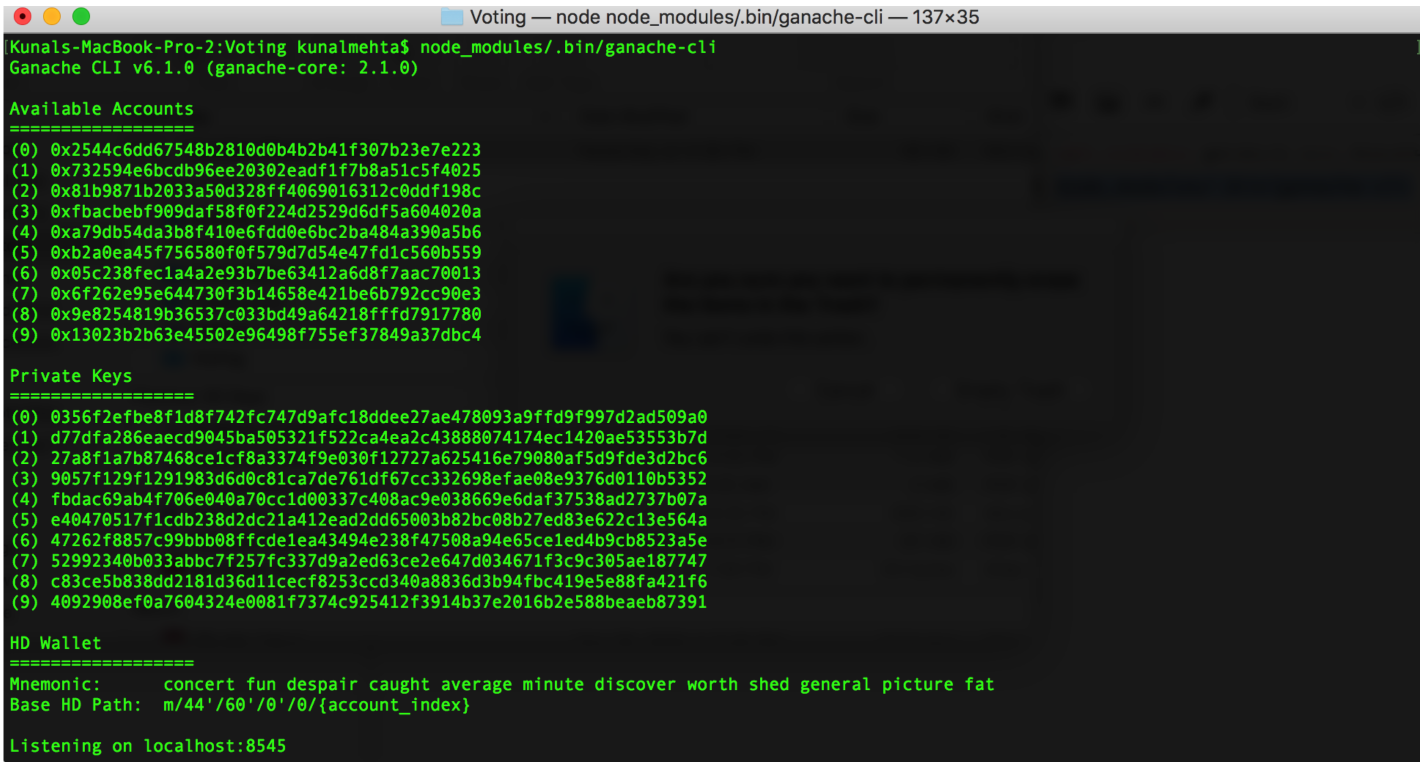
```bash
Kunals-MacBook-Pro-2:~ kunalmehta$ cd Voting
```

The inside your directory create a folder called **node_modules** to install your node.js modules and run the command to install web3 node modules. Then run ganache-cli to start the test environment.

```bash
Kunals-MacBook-Pro-2:Voting kunalmehta$ mkdir node_modules

Kunals-MacBook-Pro-2:Voting kunalmehta$ npm install ganache-cli web3@0.20.0

Kunals-MacBook-Pro-2:Voting kunalmehta$ node_modules/.bin/ganache-cli
```



```
Kunals-MacBook-Pro-2:Voting kunalmehta$ node_modules/.bin/ganache-cli
Ganache CLI v6.1.0 (ganache-core: 2.1.0)

Available Accounts
==================
(0) 0x2544c6dd67548b2810d0b4b2b41f307b23e7e223
(1) 0x732594e6bcdb96ee20302eadf1f7b8a51c5f4025
(2) 0x81b9871b2033a50d328ff4069016312c0ddf198c
(3) 0xfbacbebf909daf58f0f224d2529d6df5a604020a
(4) 0xa79db54da3b8f410e6fdd0e6bc2ba484a390a5b6
(5) 0xb2a0ea45f756580f0f579d7d54e47fd1c560b559
(6) 0x05c238fec1a4a2e93b7be63412a6d8f7aac70013
(7) 0x6f262e95e644730f3b14658e421be6b792cc90e3
(8) 0x9e8254819b36537c033bd49a64218fffd7917780
(9) 0x13023b2b63e45502e96498f755ef37849a37dbc4

Private Keys
==================
(0) 0356f2efbe8f1d8f742fc747d9afc18ddee27ae478093a9ffd9f997d2ad509a0
(1) d77dfa286eaecd9045ba505321f522ca4ea2c43888074174ec1420ae53553b7d
(2) 27a8f1a7b87468ce1cf8a3374f9e030f12727a625416e79080af5d9fde3d2bc6
(3) 9057f129f1291983d6d0c81ca7de761df67cc332698efae08e9376d0110b5352
(4) fbdac69ab4f706e040a70cc1d00337c408ac9e038669e6daf37538ad2737b07a
(5) e40470517f1cdb238d2dc21a412ead2dd65003b82bc08b27ed83e622c13e564a
(6) 47262f8857c99bbb08ffcde1ea43494e238f47508a94e65ce1ed4b9cb8523a5e
(7) 52992340b033abbc7f257fc337d9a2ed63ce2e647d034671f3c9c305ae187747
(8) c83ce5b838dd2181d36d11cecf8253ccd340a8836d3b94fbc419e5e88fa421f6
(9) 4092908ef0a7604324e0081f7374c925412f3914b37e2016b2e588beaeb87391

HD Wallet
==================
Mnemonic:      concert fun despair caught average minute discover worth shed general picture fat
Base HD Path:  m/44'/60'/0'/0/{account_index}

Listening on localhost:8545
```

Running the command will create a test environment or simulation of blockchain with 10 test accounts with 100 fake ethers in each for you to play around. The localhost: 8545 is running. Don't shut this down.

2. Writing the Smart Contract - Solidity

Now that we have set up the environment, we move on to writing the smart contract.
Note: Use Sublime Text or Visual Studio Code to edit the smart contract.

A Smart Contract is an autonomous software which functions on the set of rules defined in it. If the conditions satisfy, the smart contract is executed. It is similar to the “class” concept in object oriented programming.

Go to the **node_modules** directory and install solc (solidity compiler modules)s

```bash
Kunals-MacBook-Pro-2:node_modules kunalmehta$ npm install solc
```
BASH

The program is written in **Solidity**, a language built on Ethereum to code smart contracts. The contract is named "**Voting.sol**". There will be two main functions, precisely one to update the vote count and another to show the total votes of the candidate.

Note: The constructor is invoked once and only once when the contract is deployed. After that, if you make any changes then you have to deploy it again as in blockchain, once the contract is deployed it is immutable.

```go
pragma solidity ^0.4.18;
```
GO

We define which version of solidity to be used using **pragma** .

Then, we define the contract just as we define a class in java

```go
contract Voting {

//The code comes here

}
```
GO

Then we define **votesReceived** - the key of mapping of candidate name and store its value in an unsigned integer. We create a **candidateList** array of byte32 datatype as solidity doesn't let you pass an array of strings in the constructor.

```go
mapping (bytes32 => uint8) public votesReceived;

bytes32[] public candidateList;
```
GO

Now, we create a constructor with the same name as of the contract. Here, we will pass an array of candidates who will contest for the election.

```go
function Voting(bytes32[] candidateNames) public {

        candidateList = candidateNames;
```

```
    }
```
GO

This function return the total votes of the candidate but before that checks if the candidate is valid by calling the **validCandidate** function which will we see further.

```
function totalVotesFor(bytes32 candidate) view public returns (uint8) {
        require(validCandidate(candidate));
        return votesReceived[candidate];
    }
```
GO

The below function checks for which candidate has been voted for and increments that candidate's vote by 1.

```
 function voteForCandidate(bytes32 candidate) public {
        require(validCandidate(candidate));
        votesReceived[candidate] += 1;
    }
```
GO

The most important function is the **validCandidate** function which checks if the entered name is there in the candidate list. If not it throws an error.

```
 function validCandidate(bytes32 candidate) view public returns (bool) {
        for(uint i = 0; i < candidateList.length; i++) {
            if (candidateList[i] == candidate) {
                return true; //verified that candidate is there in list
            }
        }
        return false; //throws error if not in list
    }
```
GO

Hence, we have completed the code part of the Solidity and we have here (below) the complete code flow.

```
contract Voting {


mapping (bytes32 => uint8) public votesReceived;

bytes32[] public candidateList;


//This is the constructor which will be called once when you deploy the
contract to the blockchain, where you will pass the array list of candidates
contesting for election


    function Voting(bytes32[] candidateNames) public {

        candidateList = candidateNames;

    }


    function totalVotesFor(bytes32 candidate) view public returns (uint8) {

        require(validCandidate(candidate));

        return votesReceived[candidate];

    }



    function voteForCandidate(bytes32 candidate) public {

        require(validCandidate(candidate));

        votesReceived[candidate] += 1;

    }



    function validCandidate(bytes32 candidate) view public returns (bool) {

        for(uint i = 0; i < candidateList.length; i++) {

            if (candidateList[i] == candidate) {

                return true;

            }

        }

        return false;
```

```go
        }
    }
```
GO

3. Deploying the contract on blockchain through node.js console

Now, that the code is complete, we need to test it by deploying it on the test environment we created. web3js is a library which lets you interact with the blockchain through RPC.

```javascript
Kunals-MacBook-Pro-2:Voting kunalmehta$ node

>Web3 = require('web3')

>web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
```
JAVASCRIPT

To make sure the web3 object is initialzed and can communicate with the blochchain, lets check the ether accounts we have created in (1).

```javascript
> web3.eth.accounts

[ '0x2544c6dd67548b2810d0b4b2b41f307b23e7e223',

  '0x732594e6bcdb96ee20302eadf1f7b8a51c5f4025',

  '0x81b9871b2033a50d328ff4069016312c0ddf198c',

  '0xfbacbebf909daf58f0f224d2529d6df5a604020a',

  '0xa79db54da3b8f410e6fdd0e6bc2ba484a390a5b6',

  '0xb2a0ea45f756580f0f579d7d54e47fd1c560b559',

  '0x05c238fec1a4a2e93b7be63412a6d8f7aac70013',

  '0x6f262e95e644730f3b14658e421be6b792cc90e3',

  '0x9e8254819b36537c033bd49a64218fffd7917780',

  '0x13023b2b63e45502e96498f755ef37849a37dbc4' ]
```
JAVASCRIPT

This shows that accounts are active and can be used.

Now, to compile the contract, load the code from your 'Voting.sol' into a string and compile it.

```javascript
> code = fs.readFileSync('Voting.sol').toString()

> solc = require('solc')

> compiledCode = solc.compile(code)
```
JAVASCRIPT

When you compile the code successfully and print the 'contract' object (just type **compiledCode** in the node console to see the contents), there are two important fields you will see which are important:

- **compiledCode.contracts[':Voting'].bytecode**: This is the bytecode you get when the source code in Voting.sol is compiled. This is the code which will be deployed to the blockchain.
- **compiledCode.contracts[':Voting'].interface**: This is an interface of the contract (called **abi**) which tells the contract user what methods are available in the contract. Whenever you have to interact with the contract in the future, you will need this abi definition.

Now, to deploy the contract on the blockchain, we need to create a contract object and initiate the contract.

VotingContract is the object we created.

```javascript
> abiDefinition = JSON.parse(compiledCode.contracts[':Voting'].interface)

> VotingContract = web3.eth.contract(abiDefinition)
```
JAVASCRIPT

Now, we use the bytecode to deploy the contract onto the blockchain

```javascript
> byteCode = compiledCode.contracts[':Voting'].bytecode

> deployedContract = VotingContract.new(['Arya','John','Sansa'],{data:
byteCode, from: web3.eth.accounts[0], gas: 4700000})
```
JAVASCRIPT

Each time you make a transaction or deploy a contract on blockchain, there is a charge in value of gas (which is ether). You have passed the list of 3 candidates : Arya, John and Sansa through the contructor of the contract.
You can put names of your choice and more than 3 too.

data: This is the compiled bytecode which we deploy to the blockchain.
from: The blockchain has to keep track of who deployed the contract. In this case, we just pick the first account we get back from calling web3.eth.accounts to be the owner of this contract (who will deploy it to the blockchain).

Whereas, in the live blockchain, you can not just use any account. You have to own that account and unlock it before transacting. You are asked for a passphrase while creating an account and that is what you use to prove your ownership of that account. **Ganache** by default unlocks all the 10 accounts for convenience. (It is a test environment).

Now, we need to check the address of our deployed smart contract (for web app purpose) and also need to see if our contract is deployed successfully at that address.

```
> deployedContract.address

> contractInstance = VotingContract.at(deployedContract.address)
```
JAVASCRIPT

Finally, we have successfully built and deployed the contract on blockchain and we need to test if it works.

```
> contractInstance.totalVotesFor.call('Arya')

{ [String: '0'] s: 1, e: 0, c: [ 0 ] }
```
JAVASCRIPT

No one has voted yet, hence the total votes for Arya is 0   (c:[0])

Let's vote for Arya. you will recieve a long hashed value (transaction ID) when u vote. This value is the proof that this transaction of voting has occurred. Vote twice for anyone to see the next hash value. Now call totalvotesfor function

```
> contractInstance.voteForCandidate('Arya', {from: web3.eth.accounts[0]})

'0xdedc7ae544c3dde74ab5a0b07422c5a51b5240603d31074f5b75c0ebc786bf53'


> contractInstance.voteForCandidate('John', {from: web3.eth.accounts[0]})

'0x3da069a09577514f2baaa11bc3015a16edf26aad28dffbcd126bde2e71f2b76f'


> contractInstance.totalVotesFor.call('John').toLocaleString()

'1'
```
JAVASCRIPT

The hash value for second vote is different. It is the hash value derived from the previous hash and hence if someone tampers any vote in the blockchain it would be clearly evident as it will change the forthcoming hash values and therefore blockchain technology, being immutable and untamperable is emerging in various fields.

4. Webpage to connect to the blockchain and vote

Create the below HTML files and Javascript file and save it in your Voting folder.

HTML : index.html

```html
<!DOCTYPE html>
<html>
<head>
  <title>Voting Dapp</title>
  <link href='https://fonts.googleapis.com/css?family=Open+Sans:400,700'
rel='stylesheet' type='text/css'>
  <link
href='https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css'
rel='stylesheet' type='text/css'>
</head>
<body class="container">
  <h1>Think and Vote</h1>
  <div class="table-responsive">
    <table class="table table-bordered">
      <thead>
        <tr>
          <th>Candidate</th>
          <th>Votes</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Arya</td>
          <td id="candidate-1"></td>
        </tr>
        <tr>
          <td>John</td>
          <td id="candidate-2"></td>
        </tr>
        <tr>
```

```
            <td>Sansa</td>
            <td id="candidate-3"></td>
          </tr>
        </tbody>
      </table>
    </div>
    <input type="text" id="candidate" />
    <a href="#" onclick="voteForCandidate()" class="btn btn-primary">Vote</a>
  </body>
  <script src="https://cdn.rawgit.com/ethereum/web3.js/develop/dist/web3.js">
  </script>
  <script src="https://code.jquery.com/jquery-3.1.1.slim.min.js"></script>
  <script src="./index.js"></script>
</html>
```

MARKUP

To learn HTML , CSS and Javascript scripting and syntax click on the respective.

Javascript: index.js

```
web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
abi = JSON.parse('[{"constant":true,"inputs":
[{"name":"candidate","type":"bytes32"}],"name":"totalVotesFor","outputs":
[{"name":"","type":"uint8"}],"payable":false,"stateMutability":"view","type":
"function"},{"constant":true,"inputs":
[{"name":"candidate","type":"bytes32"}],"name":"validCandidate","outputs":
[{"name":"","type":"bool"}],"payable":false,"stateMutability":"view","type":"
function"},{"constant":true,"inputs":
[{"name":"","type":"bytes32"}],"name":"votesReceived","outputs":
[{"name":"","type":"uint8"}],"payable":false,"stateMutability":"view","type":
"function"},{"constant":true,"inputs":
[{"name":"","type":"uint256"}],"name":"candidateList","outputs":
[{"name":"","type":"bytes32"}],"payable":false,"stateMutability":"view","type
":"function"},{"constant":false,"inputs":
[{"name":"candidate","type":"bytes32"}],"name":"voteForCandidate","outputs":
```

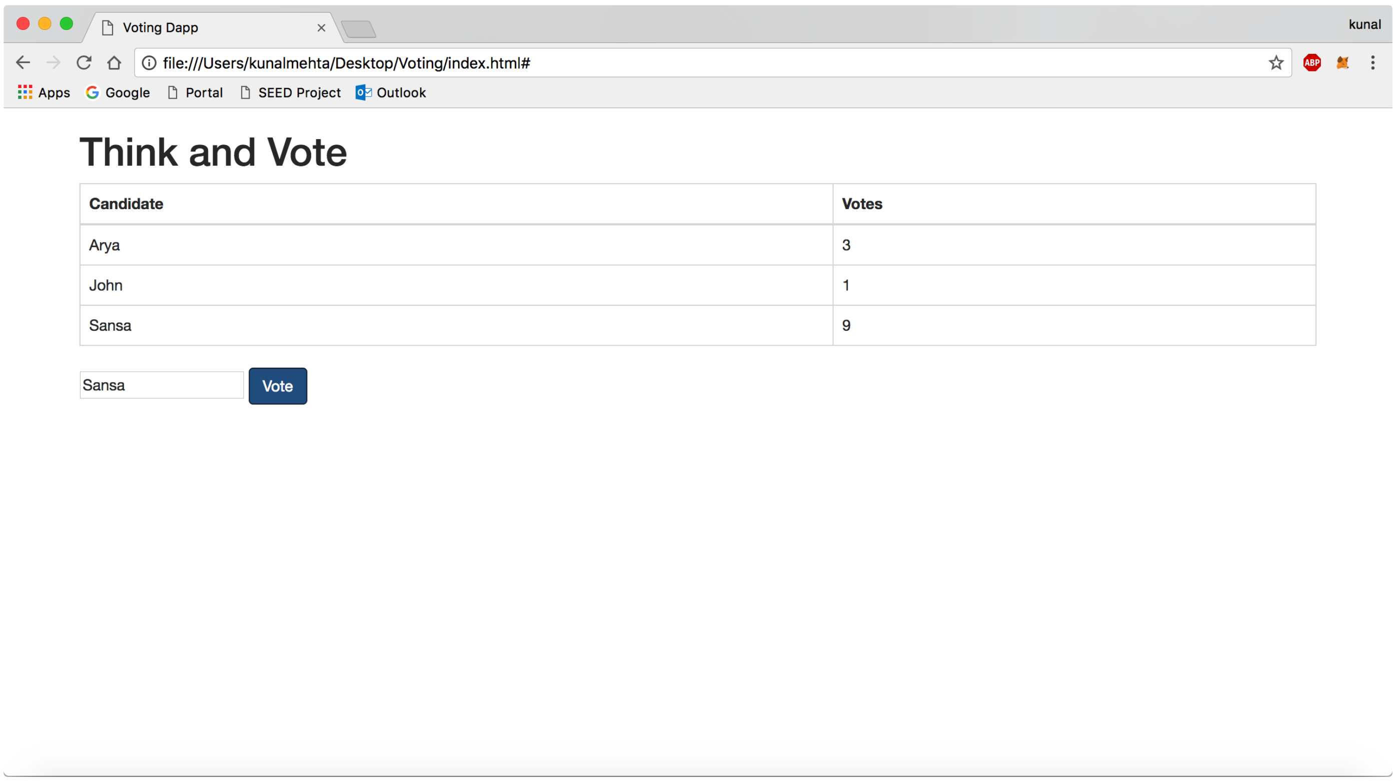
```
[],"payable":false,"stateMutability":"nonpayable","type":"function"},
{"inputs":
[{"name":"candidateNames","type":"bytes32[]"}],"payable":false,"stateMutabili
ty":"nonpayable","type":"constructor"}]')
VotingContract = web3.eth.contract(abi);
contractInstance =
VotingContract.at('0x1269b8f3296df785b5ee6aa0ad3d1d386da82ace');
candidates = {"Arya": "candidate-1", "John": "candidate-2", "Sansa":
"candidate-3"}

function voteForCandidate() {
  candidateName = $("#candidate").val();
  contractInstance.voteForCandidate(candidateName, {from:
web3.eth.accounts[0]}, function() {
    let div_id = candidates[candidateName];
    $("#" +
div_id).html(contractInstance.totalVotesFor.call(candidateName).toString());
  });
}

$(document).ready(function() {
  candidateNames = Object.keys(candidates);
  for (var i = 0; i < candidateNames.length; i++) {
    let name = candidateNames[i];
    let val = contractInstance.totalVotesFor.call(name).toString()
    $("#" + candidates[name]).html(val);
  }
});
```
JAVASCRIPT

Everytime you deploy the contract you have to update the new address and abi definition in the index.js. Make sure you pass the names correctly in the javascript file or else the function will not execute.

The webapp will run locally from your computer.


Voting Dapp

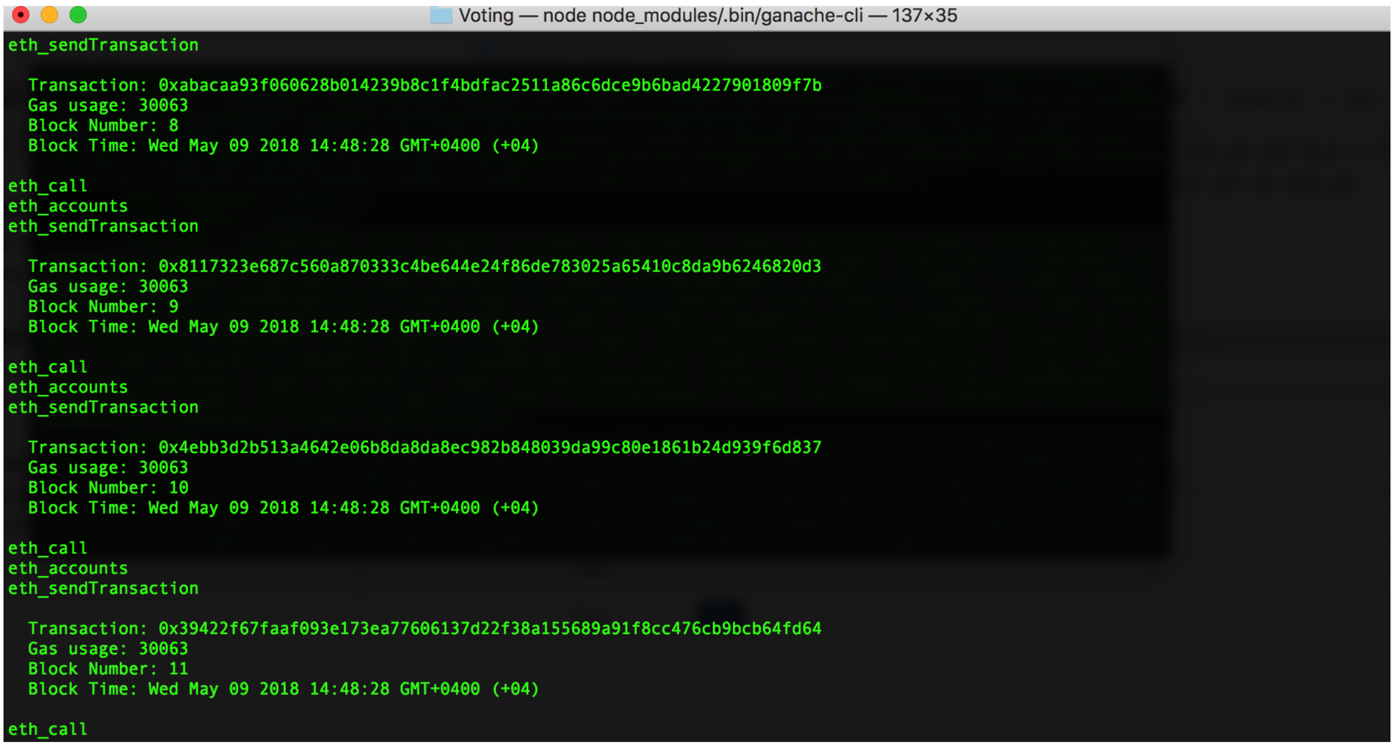
file:///Users/kunalmehta/Desktop/Voting/index.html#

Think and Vote

Candidate	Votes
Arya	3
John	1
Sansa	9

Sansa

Each time you vote, the transaction is saved as a block and generates the transaction ID (below)


Voting — node node_modules/.bin/ganache-cli — 137×35

```
eth_sendTransaction

  Transaction: 0xabacaa93f060628b014239b8c1f4bdfac2511a86c6dce9b6bad4227901809f7b
  Gas usage: 30063
  Block Number: 8
  Block Time: Wed May 09 2018 14:48:28 GMT+0400 (+04)

eth_call
eth_accounts
eth_sendTransaction

  Transaction: 0x8117323e687c560a870333c4be644e24f86de783025a65410c8da9b6246820d3
  Gas usage: 30063
  Block Number: 9
  Block Time: Wed May 09 2018 14:48:28 GMT+0400 (+04)

eth_call
eth_accounts
eth_sendTransaction

  Transaction: 0x4ebb3d2b513a4642e06b8da8da8ec982b848039da99c80e1861b24d939f6d837
  Gas usage: 30063
  Block Number: 10
  Block Time: Wed May 09 2018 14:48:28 GMT+0400 (+04)

eth_call
eth_accounts
eth_sendTransaction

  Transaction: 0x39422f67faaf093e173ea77606137d22f38a155689a91f8cc476cb9bcb64fd64
  Gas usage: 30063
  Block Number: 11
  Block Time: Wed May 09 2018 14:48:28 GMT+0400 (+04)

eth_call
```

Important note:

The contents of your file should be the following:

```
Kunals-MacBook-Pro-2:Voting kunalmehta$ ls
```

```
index.html      package-lock.json    index.js
node_modules    Voting.sol
```

You have successfully built a decentralized application for voting using a simulated ethereum blockchain environment. Enjoy !