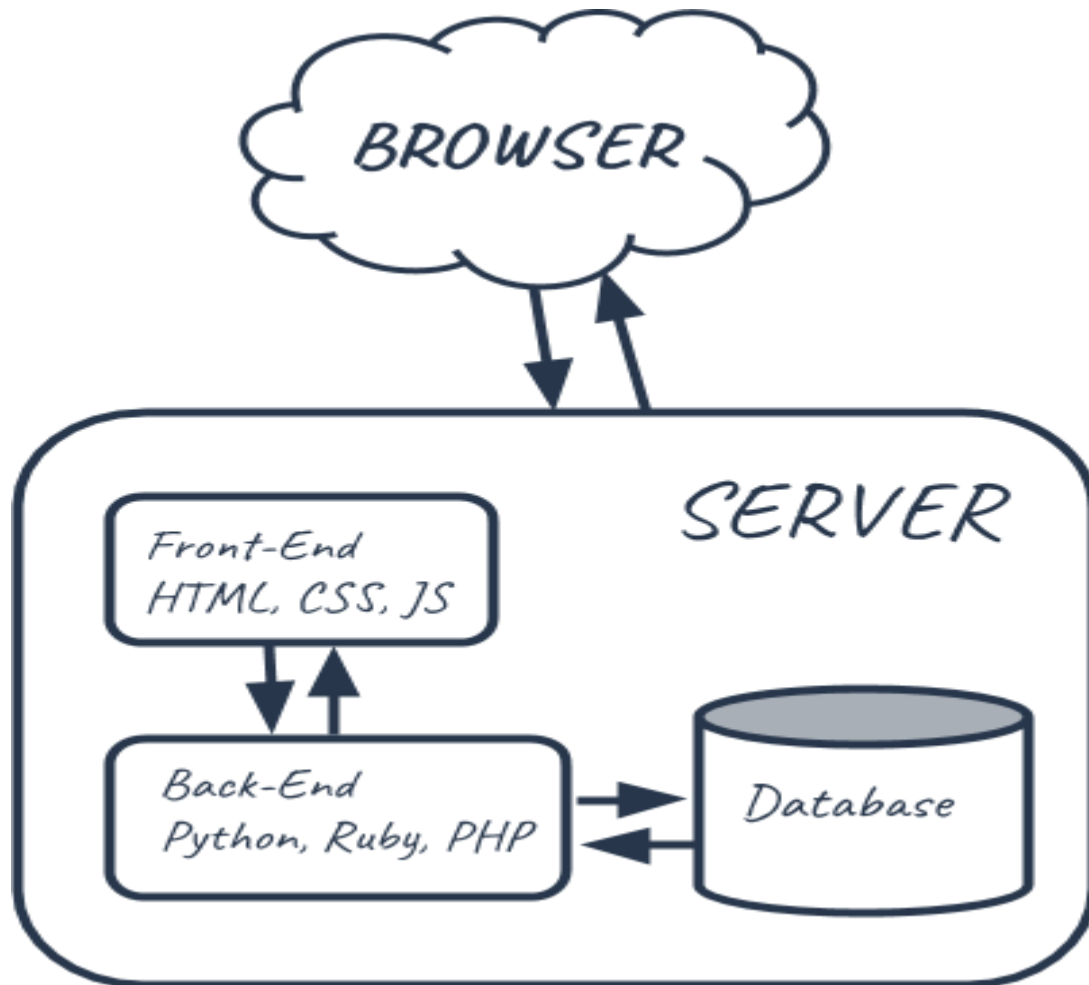


When you interact with the web application you use your browser and you talk the central server of a network all the code to your web application lives on the central server and all of the data lives in a centralised database and anytime you want to use your application you must talk to the central server ,this is how web application work



Problems with web application

1. Votes can change
2. election rules can change

This is very bad to built voting application this way ,

1. all data in your database could changed at any time of voting and
2. all the voting count could be changed or entirely deleted
3. All the code in your application could change at any time ,this means that the rules and the election could change

Requirements

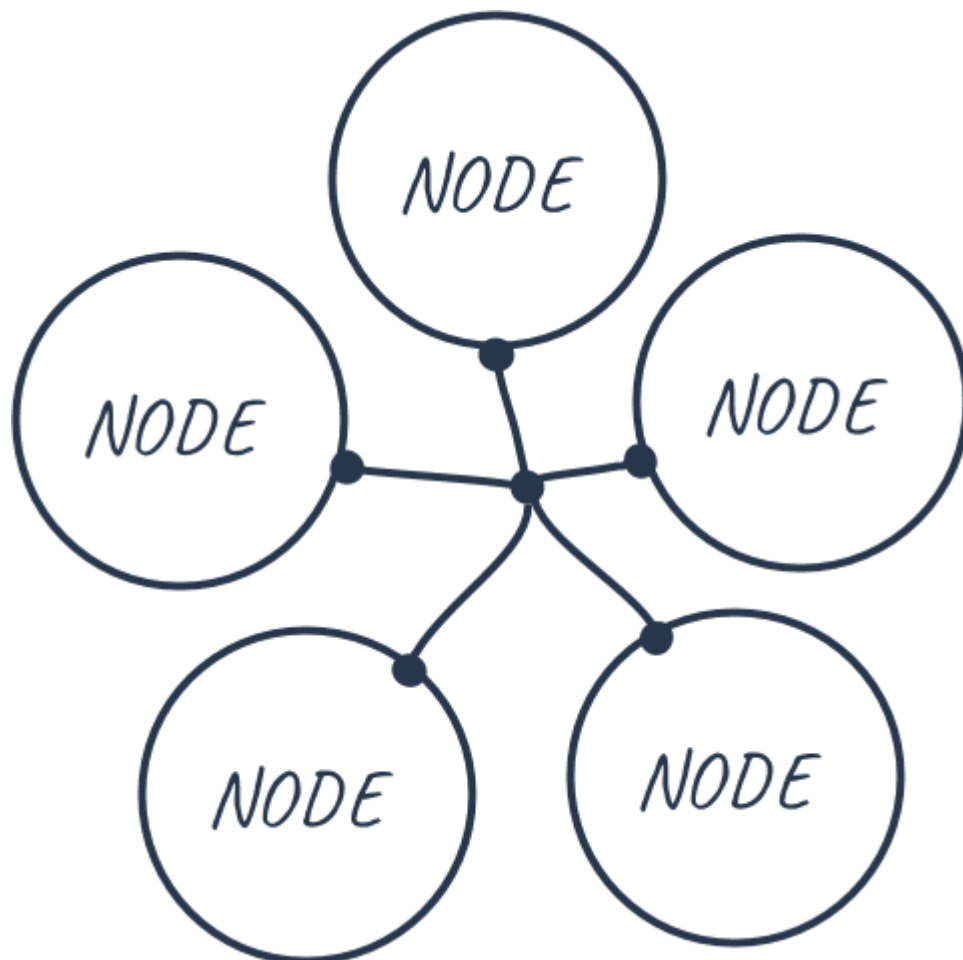
1. count votes once
2. dont change votes
3. most votes win

a central server, and a database, *the blockchain is a network and a database all in one.*

A blockchain is a peer-to-peer network of computers, called nodes, that share all the data and the code in the network. So, if you're a device connected to the blockchain, you are a node in the network, and you talk to all the computer nodes in the network. You now have a copy of all the data and the code on the blockchain.

There are no more central servers.

Just a bunch of computers that talk to one another on the same network.



Instead of a centralized database, all the transaction data that is shared across the nodes in the blockchain is contained in *bundles of records called blocks, which are chained together* to create the public ledger.

This public ledger represents all the data in the blockchain.

All the data in the public ledger is secured by cryptographic hashing, and validated by a consensus algorithm.

Nodes on the network participate to ensure that all copies of the data distributed across the network are the same.

That's one very important reason why we're building our voting application on the blockchain, because we want to ensure that our vote was counted, and that it did not change.

What would it look like for a user of our application to vote on the blockchain?

Well, for starters, the user needs an account with a wallet address with some Ether, Ethereum's cryptocurrency. Once they connect to the network, they cast their vote and pay a small transaction fee to write this transaction to the blockchain. This transaction fee is called "gas". Whenever the vote is cast, some of the nodes on the network, called miners, compete to complete this transaction. The miner who completes this transaction is awarded the Ether that we paid to vote.

As a recap, when I vote, I pay a gas price to vote, and when my vote gets recorded, one of the computers on the network gets paid the my Ether fee. I in turn am confident my vote was recorded accurately forever.

So it's also important to note that voting on the blockchain costs Ether, but just seeing a list of candidates does not. That's because reading data from the blockchain is free, but writing to it is not.

What about the code?

- also shared
- also unchangeable

What is DAPP

- Decentralised application
- peer to peer network connection
- data is a shared
- code is shared

What is a Smart Contract?

- Code on blockchain
- like a microservice
- Written in solidity language

That's how the voting process works, but how do we actually code our app? Well, the Ethereum blockchain allows us to execute code with the Ethereum Virtual Machine (EVM) on the blockchain with something called a smart contract.

Smart contracts are where all the business logic of our application lives.

This is where we'll actually code the decentralized portion of our app. Smart contracts are in charge of reading and writing data to the blockchain, as well as executing business logic. Smart contracts are written in a programming language called [Solidity](#), which looks a lot like Javascript.

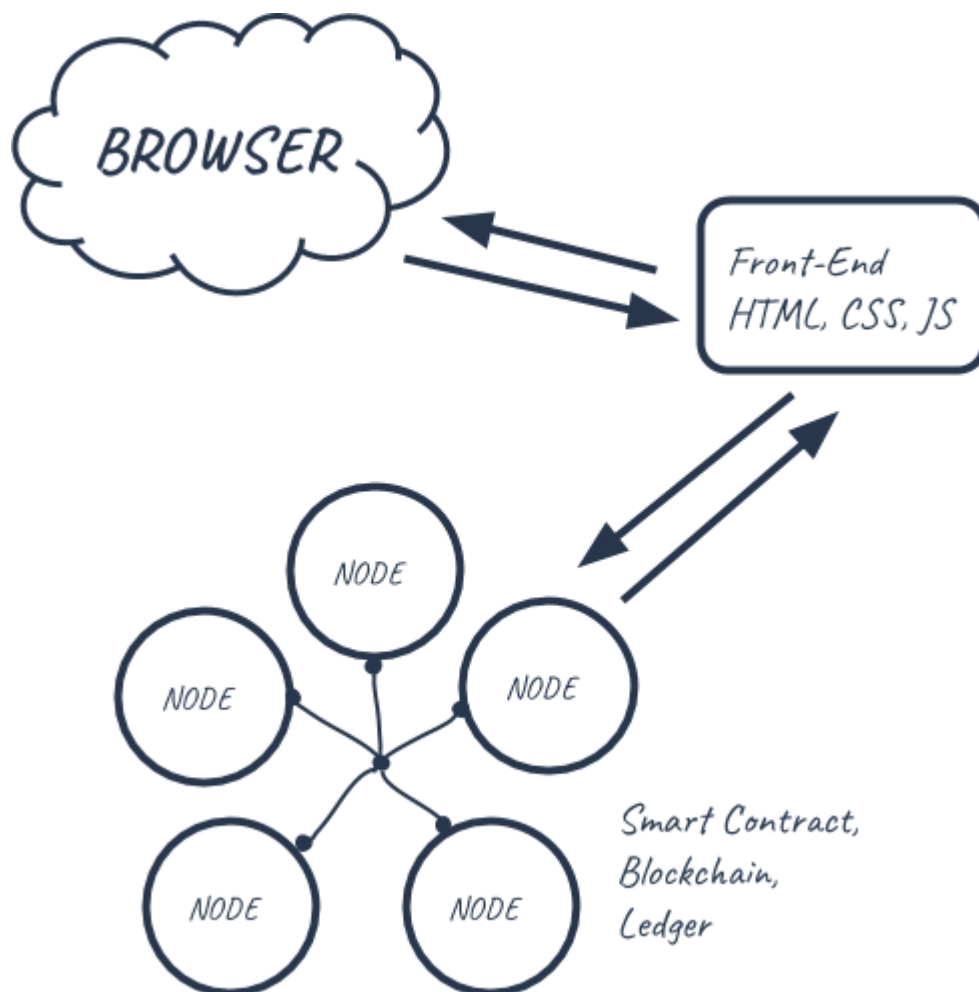
It is a full blown programming language that will allow us to do many of the same types of things Javascript is capable of, but it behaves a bit differently because of its use case, as we'll see in this tutorial.

The function of smart contracts on the blockchain is very similar to a microservice on the web.

If the public ledger represents the database layer of the blockchain, then smart contracts are where all the business logic that transacts with that data lives.

Also, they're called smart contracts because they represent a covenant or agreement. In the case of our voting dApp, it is an agreement that my vote will count, that other votes are only counted once, and that the candidate with the most votes will actually win the election.

Now let's take a quick look at the structure of the dApp we're building.



We'll have a traditional front-end client that is written in HTML, CSS, and Javascript. Instead of talking to a back-end server, this client will connect to a local Ethereum blockchain that we'll install. We'll code all the business logic about our dApp in an Election smart contract with the Solidity programming language. We'll deploy this smart contract to our local Ethereum blockchain, and allow accounts to start voting.

Now we've seen what a blockchain is and how it works. We've seen why we want to build our voting dApp on the blockchain instead of the current web. And we've seen that we want to code our dApp by writing a smart contract that will be deployed to the Ethereum blockchain. Now let's jump in and start programming!

What We'll Be Building

Here is a demonstration of the voting dApp that we'll be building.

Election Results

#	Name	Votes
1	Candidate 1	0
2	Candidate 2	0

Select Candidate

Candidate 1

Vote

Your Account: 0x2191ef87e392377ec08e7c08eb105ef5448eced5

We'll build a client-side application that will talk to our smart contract on the blockchain. This client-side application will have a table of candidates that lists each candidate's id, name, and vote count. It will have a form where we can cast a vote for our desired candidate. It also shows the account we're connected to the blockchain with under "your account".

1. Smoke test

Truffle Framework gives us boxes which are just packages Of boilerplate code that can be unboxed within our project To get us started faster

Using truffle petshop box

- This is a box which is published by truffle for all for their ethereum pet shop

We can unbox the truffle Pet Shop by using

truffle unbox pet-shop (in cmd prompt)

We can jump/directly use a sublime text using

Subl (in cmd prompt)

In contracts, migrations.sol

In migrations directory

this is where all of our migration files will live whenever we deploy our smart contracts to the blockchain and if you come from another web development environment.

you might be familiar with the migrations directory when you have to create a migration to change your database

This is the same thing in truffle so we need a migration to deploy our smart contract to blockchain Because when we do this we actually performing a transaction on the blockchain

We are changing its state just like we would change the state of a database

In node_modules

Here all node directories or modules will live in

In src

here we developed all our the client side app

In test

Keep all the test files

Package.json

Here we will specify all our dependency

Truffle.js

It is main configuration file for the truffle project

Code contracts:

1. Version of solidity

```
pragma solidity ^0.5.0;
```

2. Contract election made which contain store,read candidate . Define constructor using a keyword constructor

```
pragma solidity ^0.5.0;

contract Election{
    // Store candidate
    // Read Candidate
    string public candidate;

    // Constructor
    constructor () public {
        candidate = "Candidate 1";
    }
}
```

Constructor always going to run when we deploy a blockchain

3. here candidate represent as state variable And it can be access inside of a contract . It represent data that belong to our entire contract.

```
candidate = "Candidate 1";
```

4. here be we made it a state variable of string type whose accessibility is public

```
// Read Candidate  
string public candidate;
```

Create a migration to deploy a smart contract to a local blockchain So that we can directly interact with them in the console. for this we go to **migration directory**, we'll use this **initial migration** file as a reference point for creating our own migration

We create new file on same dir and we prepend it with 2 . we number all of our file in the migration directories So that the truffle knows the order to them

In deploy contracts

```
var Election = artifacts.require("./Election.sol");  
  
module.exports = function(deployer) {  
  deployer.deploy(Election);  
};
```

Artifacts = Represent the contract abstraction that is specific to truffle and it gives us election artifacts that represent our Smart contract.

Truffle will expose this so that we can interact with it in any case that we want to so in the console when we write test or When we use it in our front end app

```
module.exports = function(deployer) {  
  deployer.deploy(Election);  
};
```

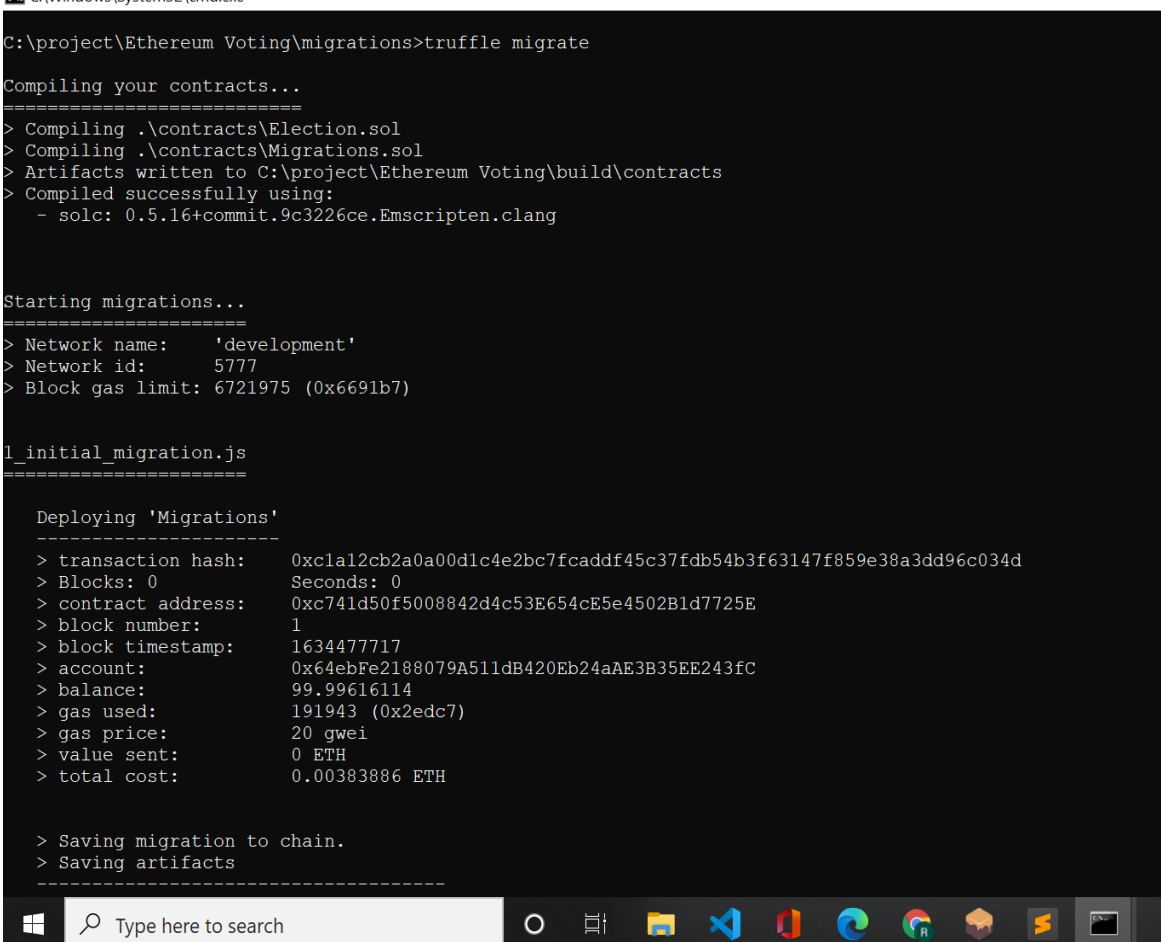
Here we added a directive to deploy a contract with this function

```
var Election = artifacts.require("./Election.sol");
```

```
module.exports = function(deployer) {  
  deployer.deploy(Election);  
};
```

Here are migration file is completed

To migrate the contract to the blockchain we use
truffle migrate



```
C:\Windows\System32\cmd.exe  
C:\project\Ethereum Voting\migrations>truffle migrate  
  
Compiling your contracts...  
=====
```

>	Compiling	.\contracts\Election.sol
>	Compiling	.\contracts\Migrations.sol
>	Artifacts written to C:\project\Ethereum Voting\build\contracts	
>	Compiled successfully using:	
	-	solc: 0.5.16+commit.9c3226ce.Emscripten.clang

```
  
Starting migrations...  
=====
```

>	Network name:	'development'
>	Network id:	5777
>	Block gas limit:	6721975 (0x6691b7)

```
  
1_initial_migration.js  
=====
```

```
  Deploying 'Migrations'  
  -----  
  > transaction hash: 0xc1a12cb2a0a00d1c4e2bc7fcaddf45c37fdb54b3f63147f859e38a3dd96c034d  
  > Blocks: 0  
  > contract address: 0xc741d50f5008842d4c53E654cE5e4502B1d7725E  
  > block number: 1  
  > block timestamp: 1634477717  
  > account: 0x64ebFe2188079A511dB420Eb24aAE3B35EE243fC  
  > balance: 99.99616114  
  > gas used: 191943 (0x2edc7)  
  > gas price: 20 gwei  
  > value sent: 0 ETH  
  > total cost: 0.00383886 ETH  
  
  > Saving migration to chain.  
  > Saving artifacts  
  -----
```

```
C:\Windows\System32\cmd.exe
> Saving migration to chain.
> Saving artifacts
-----
> Total cost:          0.00383886 ETH

2_deploy_contracts.js
=====

Deploying 'Election'
-----
> transaction hash:    0xc0ae53950dbb281da837c70078d7621429d35fa54bbff237adc78c5e09974f4f
> Blocks: 0           Seconds: 0
> contract address:   0xAF97Cd212E191557B67E6923A8E7f92988624FDB
> block number:       3
> block timestamp:    1634477718
> account:            0x64ebFe2188079A511dB420Eb24aAE3B35EE243fC
> balance:            99.99206904
> gas used:           162267 (0x279db)
> gas price:          20 gwei
> value sent:         0 ETH
> total cost:         0.00324534 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost:          0.00324534 ETH

Summary
=====
> Total deployments:   2
> Final cost:          0.0070842 ETH

C:\project\Ethereum Voting\migrations>
```

Run truffle on console write **truffle console**

Election.deployed().then(function(instance){app = instance})

Election here is the variable name that we define inside our migration file

.deployed() And we can get the deploy abstraction with this deploy function

Because of asynchronous nature of the smart contracts with the heavily rely on Javascript promises this are basically eventual result of an asynchronous operation

var app = Election.deployed()

I can't just expect that the result to be assigned to a **app** variable /It doesn't work in this way it will return promise then we have to call the **then() then** function on promise that will get executed once the promises finishes

function(instance){app = instance}

That promise will take a callback function where we can inject instance of an application and assign the instance to an app variable

We can get our address using **app.address**

We can also retrieve the value that we store when defined the state variable candidate and set it when we initialize our contract.

This is the value that we set in the constructor and check if our constructor was run whenever we deploy our contract

CMd : app.candidate()

Op 'Candidate 1'

Read on the blockchain is free but **write** on the block change cost gas

2. Listing Candidates

```
// Model candidate
struct Candidate
{
    uint id; //unsigned int
    string name;
    uint voteCount;
}
```

```
// Fetch candidate
```

mapping(uint => Candidate) public candidates;

Map in solidity like mapping in associative array or hash Where we associate key value pair . Declare this mapping with mapping **keyword (key => value)**

mapping(uint => Candidate) candidate structure type

Declare this mapping public

Solidity will create candidates function that allow to fetch our candidates from this mapping

Lastly ,we detect how many candidates that we have to created in our election

uint public candidatesCount;

This state variable **candidatesCount** used to checkout how many candidates are registered for election

- State variable just like instance variable
- Local variable have the scope within the function .
represent by **_name (underscore because variable name)**

To add candidate we use add candidate function

```
function addCandidate(String _name) private
{
    candidatesCount++; // increase candidate count
    candidates[candidatesCount] =
Candidate(candidatesCount,_name,0); //mapping (key count =
Candidate(struct)
}
```

Default size of unsigned integer is 0 **candidatesCount = 0 (default)**

**** Truffle migrate --reset**

To make any changes in the code ,we have to send the copy of change smart contract to a local blockchain and to get our code changes in the development purposes. For this we use this statement **--reset**

- When we do this we lose all the state of our smart
- we also be assigning a new address for it
- For this reason migration should be run only once but we are developing for this reason we can do that
- This is just like dropping all the table from the database and started from the beginning

For checking

1. Truffle console
2. Want deployed instance of our contract and assign it to variable

Election.deployed().then(function(i){app = i;})

3. app.candidates()

By default provide candidates func for mapping

For candidate 1

```
}
truffle(development)> app.candidates(1)
Result {
  '0': BN {
    negative: 0,
    words: [ 1, <1 empty item> ],
    length: 1,
    red: null
  },
  '1': 'Congress',
  '2': BN {
    negative: 0,
    words: [ 0, <1 empty item> ],
    length: 1,
    red: null
  },
  id: BN {
    negative: 0,
    words: [ 1, <1 empty item> ],
    length: 1,
    red: null
  },
  name: 'Congress',
  voteCount: BN {
    negative: 0,
    words: [ 0, <1 empty item> ],
    length: 1,
    red: null
  }
}
truffle(development)> _
```


For candidate 2

```
}
truffle(development)> app.candidates(2)
Result {
  '0': BN {
    negative: 0,
    words: [ 2, <1 empty item> ],
    length: 1,
    red: null
  },
  '1': 'BJP',
  '2': BN {
    negative: 0,
    words: [ 0, <1 empty item> ],
    length: 1,
    red: null
  },
  id: BN {
    negative: 0,
    words: [ 2, <1 empty item> ],
    length: 1,
    red: null
  },
  name: 'BJP',
  voteCount: BN {
    negative: 0,
    words: [ 0, <1 empty item> ],
    length: 1,
    red: null
  }
}
```

For invalid candidate (default value for candidate structure)

```
truffle(development)> app.candidates(3)
Result {
  '0': BN {
    negative: 0,
    words: [ 0, <1 empty item> ],
    length: 1,
    red: null
  },
  '1': '',
  '2': BN {
    negative: 0,
    words: [ 0, <1 empty item> ],
    length: 1,
    red: null
  },
  id: BN {
    negative: 0,
    words: [ 0, <1 empty item> ],
    length: 1,
    red: null
  },
  name: '',
  voteCount: BN {
    negative: 0,
    words: [ 0, <1 empty item> ],
    length: 1,
    red: null
  }
}
truffle(development)>
```

app.candidatesCount()

```
at REPLServer.EventEmitter.emit (domain.js:467:12)
truffle(development)> app.candidatesCount()
BN { negative: 0, words: [ 2, <1 empty item> ], length: 1, red: null }
truffle(development)> _
```

app.candidates(1).then(function(c){candidate1 = c;})

- Store value of candidate in candidate1 variable
- Assign candidate1 value with promise inside the callback function
- Return all the detail of the candidate1

Candidate1.name "congress"

Candidate1.name "1"

we can also use index value for fetching the values in candidate1

```
truffle(development)> candidate
Result {
  '0': BN {
    negative: 0,
    words: [ 1, <1 empty item> ],
    length: 1,
    red: null
  },
  '1': 'Congress',
  '2': BN {
    negative: 0,
    words: [ 0, <1 empty item> ],
    length: 1,
    red: null
  },
  id: BN {
    negative: 0,
    words: [ 1, <1 empty item> ],
    length: 1,
    red: null
  },
  name: 'Congress',
  voteCount: BN {
    negative: 0,
    words: [ 0, <1 empty item> ],
    length: 1,
    red: null
  }
}
```

```
truffle(development)> candidate[0]
BN { negative: 0, words: [ 1, <1 empty item> ], length: 1, red: null }
truffle(development)> candidate[1]
'Congress'
truffle(development)> candidate[2]
BN { negative: 0, words: [ 0, <1 empty item> ], length: 1, red: null }
truffle(development)> _
```

Return integer value only for that index

- candidate1[0].toNumber() OP “1”
- candidate1[2].toNumber() OP “0”

Now coming to the voters ganache provide 10 account , and some of the account is connected in the console and it is done with the help of web3 library

Web3 has ethereum object which is eth

- **web3.eth**

```
C:\Windows\System32\cmd.exe - truffle console
truffle(development)> candidate[2].toNumber()
0
truffle(development)> web3.eth
<ref *2> Eth {
  currentProvider: [Getter/Setter],
  _requestManager: RequestManager {
    provider: HttpProvider {
      withCredentials: false,
      timeout: 0,
      headers: undefined,
      agent: undefined,
      connected: true,
      host: 'http://127.0.0.1:7545',
      httpAgent: [Agent],
      send: [Function (anonymous)],
      _alreadyWrapped: true
    },
    providers: {
      WebsocketProvider: [Function: WebsocketProvider],
      HttpProvider: [Function: HttpProvider],
      IpcProvider: [Function: IpcProvider]
    },
    subscriptions: Map(0) {}
  },
  givenProvider: null,
  providers: {
    WebsocketProvider: [Function: WebsocketProvider],
    HttpProvider: [Function: HttpProvider],
    IpcProvider: [Function: IpcProvider]
  },
  _provider: HttpProvider {
    withCredentials: false,
    timeout: 0,
    headers: undefined,
    agent: undefined,
    connected: true,
    host: 'http://127.0.0.1:7545',
    httpAgent: Agent {
      _events: [Object: null prototype],
      _eventsCount: 2,
```

And so on

- **Web3.eth.accounts**

It will give all hash of accounts in ganache

- **Web3.eth.getAccounts()**

Return All the accounts hash address in the ganache

```
truffle(development)> web3.eth.getAccounts()  
[  
  '0x64ebFe2188079A511dB420Eb24aAE3B35EE243fC', 3582cCA',  
  '0x1E27A0EA64808c0538cd18F609dd82bd5C1c2eA6', 7d935c1c6b416b9ab754933d3e56df0ac2',  
  '0x5B553cA9B92cB608C2Be0f950787C42dE4947bf9',  
  '0xEACb8cC444034F26d99394A1F74bf871bFf0B3C8',  
  '0x316aeaFEECA568a73652cEA024cb02d8716aa9d2',  
  '0xFc00AD3F7Ec93c0aBc5C78A6800315F349a22E0C',  
  '0xf7c81A0e38Cf79309083D8C2004a6b98E9Cd2fEB',  
  '0x799962690181AE527cA3270541707f0ce919e72E',  
  '0x1216b133e4d1d8668895E810A7bb645a3cB2b3ac',  
  '0xC164cd35F7687F0Dad48739beDdc166710833F5E'  
]
```

Now in the **test directory** ; we make **election.js** where we run our test

- We write test in JavaScript to simulate client-side interaction With our contract just like in the **truffle console**
- Truffle comes bundle in Mocha (Testing framework) and Chai (assertion library)
- Assertion libraries **are tools to verify that things are correct.**
This makes it a lot easier to test your code, so you don't have to do thousands of if statements
- A testing framework is **a set of guidelines or rules used for creating and designing test cases.** A framework is comprised of a combination of practices and tools that are designed to help QA professionals test more efficiently.
- Both of these things will give us everything to write test for our smart contract.

```
var Election = artifacts.require(".Election.sol");
```

- here we made artifacts that truffle will use to create an abstraction to interact with the contract
- Next thing is to declare the contract

```
contract("Election", function(accounts)
{
});
```

- **function(accounts)** - This will going to inject all the accounts exists in the development environment and can use this account for the testing purposes

```
contract("Election", function(accounts)
{
    it("initializes with three candidates",function(){
        return Election.deployed().then(function(instance)
        {
            return instance.candidatesCount();
        }).then(function(count){

            assert.equal(count,3);
        });//count

    });//it
});
```

- We get all this from mocha testing framework

```
it("initializes with three candidates",function(){
    return Election.deployed().then(function(instance){
        return instance.candidatesCount();
    }).then(function(count){
        assert.equal(count,3);
    });//count
});//it
```

- An **assert.equal(count,3)** something to get from **chai** (Assertion library)
- first of all we add a description initialises the three candidate
- After that fetch an instance of our deploy contract And get access to the instance in this callback function
- once we have that we got going to fetch our candidateCount from it.

- candidatecount call is asynchronous for this we need to execute the promise chain and inject value of count here **(function(count)**
- After that assert a certain wheel compare the value if count is equal to 3
- Now save and run the test using
Truffle test

For assert.equal(count,3) : op passed

```
C:\project\Ethereum Voting>truffle test
Using network 'development'.

Compiling your contracts...
=====
> Compiling .\contracts\Election.sol
> Compiling .\contracts\Migrations.sol
> Artifacts written to C:\Users\sarry\AppData\Local\Temp\test--64936-ZqI4wcgm2AyL
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang

Contract: Election
  ✓ initializes with three candidates (137ms)

1 passing (243ms)
```

For assert.equal(count,2) : op failed

```
C:\project\Ethereum Voting>truffle test
Using network 'development'.

Compiling your contracts...
=====
> Compiling .\contracts\Election.sol
> Compiling .\contracts\Migrations.sol
> Artifacts written to C:\Users\sarry\AppData\Local\Temp\test--59628-cb9uwOC6dWyY
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang

Contract: Election
  1) initializes with three candidates
     > No events were emitted

0 passing (240ms)
1 failing

1) Contract: Election
   initializes with three candidates:
     AssertionError: expected <BN: 3> to equal 2
       at C:\project\Ethereum Voting\test\election.js:11:11
       at processTicksAndRejections (internal/process/task_queues.js:93:5)
```

- Next thing to test is our candidate is initialised with the initial value
- **it** - Initialises the candidate with the correct value

```
it("it initializes the candidates with correct values", function(){
    return Election.deployed().then(function(instance){
        electionInstance = instance;
        return electionInstance.candidates(1);

    }).then(function(candidate){

        assert.equal(candidate[0],1,"contains the correct id");
        assert.equal(candidate[1],"Congress","contains the correct name");
        assert.equal(candidate[2],0,"contains the correct vote count");
        return electionInstance.candidates(2);

    }).then(function(candidate){

        assert.equal(candidate[0],2,"contains the correct id");
        assert.equal(candidate[1],"BJP","contains the correct name");
        assert.equal(candidate[2],0,"contains the correct vote count");
        return electionInstance.candidates(3);


    }).then(function(candidate){

        assert.equal(candidate[0],3,"contains the correct id");
        assert.equal(candidate[1],"AAP","contains the correct name");
        assert.equal(candidate[2],0,"contains the correct vote count");
    });
}); //it
```

- We get the copy of our deployed contract by this
Election.deployed().then(function(instance){
 electionInstance = instance;
 return electionInstance.candidates(1);

- `var electionInstance;` - Be declared `electionInstance` variable because we need to be have the scope of it entirely test
- `return electionInstance.candidates(2);`
`}).then(function(candidate){`
This Calling is asynchronous for this we muse use promise chain with callback function and pass candidate in it

Test it using **truffle test**

 C:\Windows\System32\cmd.exe

```
C:\project\Ethereum Voting>truffle test
Using network 'development'.

Compiling your contracts...
=====
> Compiling .\contracts\Election.sol
> Compiling .\contracts\Migrations.sol
> Artifacts written to C:\Users\sarry\AppData\Local\Temp\test--59400-Ltk307Mlodi9
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang

Contract: Election
  ✓ initializes with three candidates (190ms)
  ✓ it initializes the candidates with correct values (397ms)

2 passing (829ms)

C:\project\Ethereum Voting>
```

For client side , index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Election Results</title>

    <!-- Bootstrap -->
    <link href="css/bootstrap.min.css" rel="stylesheet">
  </head>
  <body>
    <div class="container" style="width: 650px;">
      <div class="row">
        <div class="col-lg-12">
          <h1 class="text-center">Election Results</h1>
          <hr/>
          <br/>
          <div id="loader">
            <p class="text-center">Loading...</p>
          </div>
          <div id="content" style="display: none;">
            <table class="table">
              <thead>
                <tr>
                  <th scope="col">#</th>
                  <th scope="col">Name</th>
                  <th scope="col">Votes</th>
                </tr>
              </thead>
              <tbody id="candidatesResults">
                </tbody>
            </table>
            <hr/>
            <p id="accountAddress" class="text-center"></p>
          </div>
        </div>
      </div>
    </div>

    <!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>
    <!-- Include all compiled plugins (below), or include individual files as
needed -->
    <script src="js/bootstrap.min.js"></script>
    <script src="js/web3.min.js"></script>
    <script src="js/truffle-contract.js"></script>
    <script src="js/app.js"></script>
  </body></html>
```

In app.js

```
App = {
  web3Provider: null,
  contracts: {},
  account: '0x0',

  //initialize our app initialize web3
  init: function() {
    return App.initWeb3();
  },

  //it is basically client side application to our local blockchain
  initWeb3: function() {
    if (typeof web3 !== 'undefined') {
      // If a web3 instance is already provided by Meta Mask.
      App.web3Provider = web3.currentProvider;
      web3 = new Web3(web3.currentProvider);
    } else {
      // Specify default instance if no web3 instance provided
      App.web3Provider = new
Web3.providers.HttpProvider('http://localhost:7545');
      web3 = new Web3(App.web3Provider);
    }
    return App.initContract();
  },

  //initialize contract
  //getJSON("Election.json", works because we have browser sync
package bs-config.json file that came with our truffle box
  initContract: function() {
    $.getJSON("Election.json", function(election) {
      // Instantiate a new truffle contract from the artifact
      App.contracts.Election = TruffleContract(election);
      // Connect provider to interact with contract
      App.contracts.Election.setProvider(App.web3Provider);

      return App.render();
    });
  },

  //render out content of the application on the page
  render: function() {
    var electionInstance;
    var loader = $("#loader");
    var content = $("#content");
```

```

        loader.show();
        content.hide();

        // Load account data
        web3.eth.getCoinbase(function(err, account) {
            if (err === null) {
                App.account = account;
                $("#accountAddress").html("Your Account: " + account);
            }
        });

        // Load contract data
        App.contracts.Election.deployed().then(function(instance) {
            electionInstance = instance;
            return electionInstance.candidatesCount();
        }).then(function(candidatesCount) {
            var candidatesResults = $("#candidatesResults");
            candidatesResults.empty();

            for (var i = 1; i <= candidatesCount; i++) {
                electionInstance.candidates(i).then(function(candidate) {
                    var id = candidate[0];
                    var name = candidate[1];
                    var voteCount = candidate[2];

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

            loader.hide();
            content.show();
        }).catch(function(error) {
            console.warn(error);
        });
    }
};

$(function() {
    $(window).load(function() {
        App.init();
    });
});

```

Let's take note of a few things that this code does :

1. Set up web3: [web3.js](#) is a javascript library that allows our client-side application to talk to the blockchain. We configure web3 inside the "initWeb3" function.
2. Initialize contracts: We fetch the deployed instance of the smart contract inside this function and assign some values that will allow us to interact with it.
3. Render function: The render function lays out all the content on the page with data from the smart contract. For now, we list the candidates we created inside the smart contract. We do this by looping through each candidate in the mapping, and rendering it to the table. We also fetch the current account that is connected to the blockchain inside this function and display it on the page.

- **Truffle Pet Shop** give us a lite server see in **package.json**
- **We will use this to built all our client-side assets**
- **Truffle box** also give us browser sync **bs-config.json** to giving access to get JSON function for our artifacts abstraction or our contract abstraction. Its also going to reload content on the page

RUN

- Truffle migrate --reset
- To run Lite server we use command

npm run dev

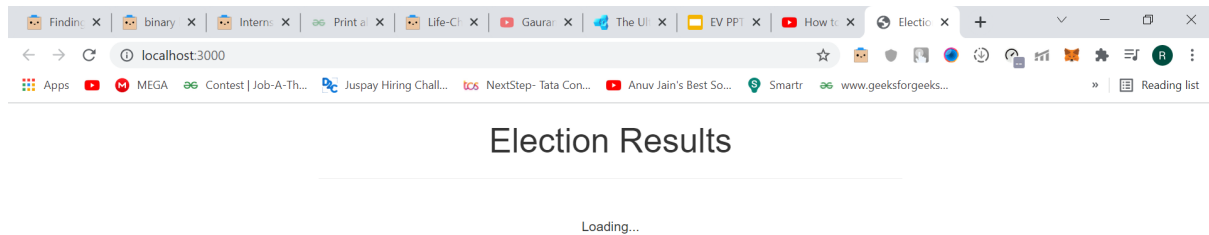
```
lite-server
Microsoft Windows [Version 10.0.19042.1288]
(c) Microsoft Corporation. All rights reserved.

C:\project\Ethereum Voting>npm run dev

> pet-shop@1.0.0 dev C:\project\Ethereum Voting
> lite-server

** browser-sync config **
{
  injectChanges: false,
  files: [ './**/*.html', './**/*.css', './**/*.js' ],
  watchOptions: { ignored: 'node_modules' },
  server: {
    baseDir: [ './src', './build/contracts' ],
    middleware: [ [Function (anonymous)], [Function (anonymous)] ]
  }
}
[Browsersync] Access URLs:
-----
    Local: http://localhost:3000
  External: http://192.168.137.1:3000
-----
    UI: http://localhost:3001
  UI External: http://localhost:3001
-----
[Browsersync] Serving files from: ./src
[Browsersync] Serving files from: ./build/contracts
[Browsersync] Watching files...
21.11.17 13:30:04 200 GET /index.html
21.11.17 13:30:04 200 GET /js/bootstrap.min.js
21.11.17 13:30:04 200 GET /css/bootstrap.min.css
21.11.17 13:30:04 200 GET /js/app.js
21.11.17 13:30:04 200 GET /js/web3.min.js
21.11.17 13:30:04 200 GET /js/truffle-contract.js
21.11.17 13:30:04 200 GET /Election.json
21.11.17 13:30:04 404 GET /favicon.ico
[Browsersync] Reloading Browsers... (buffered 4 events)
21.11.17 13:30:04 304 GET /index.html
21.11.17 13:30:04 304 GET /css/bootstrap.min.css
21.11.17 13:30:04 304 GET /js/bootstrap.min.js
```

It will directly redirect to the client application it is currently running but it is not connected to the local blockchain .For that reason it is showing loading



- In order to connect we use the metamask extension dependency

3. Cast Votes

Msg.sender (in vote func)

MSG stand for message which is send as a metadata and the sender is the address that cast vote

```
// store accounts that have voted  
mapping(address=>bool) public voters;
```

`web3.eth.getAccounts()` - get all account address

```
web3.eth.getAccounts().then(e => { firstAccount = e[0]; console.log("A: " +  
firstAccount); })
```

```
web3.eth.getAccounts().then(e => let firstAcc=e[0]; console.log(firstAcc))
```

Get access individual address outside getAccounts() func

- `var a`
- `web3.eth.getAccounts().then(e=> {a = e[0];})`
- `a`
`op-'0x64ebFe2188079A511dB420Eb24aAE3B35EE243fC'`

Or do this

- `let accounts = await web3.eth.getAccounts()`
- `Accounts[0]` same output

```
app.vote(1, {from : a} )
```

- Candidate number , metadata

[illegible]

- **Allows of voter to cast a vote in the test directory for the testing**

```
it("allows a voter to cast a vote", function(){
    return Election.deployed().then(function(instance){
```



```

        electionInstance = instance;
        candidateId = 1;
        return electionInstance.vote(candidateId,{from : accounts[0]});

    }).then(function(receipt){
        return electionInstance.voters(accounts[0]);

    }).then(function(voted){

        assert(voted,"the voter was marked as voted");
        return electionInstance.candidates(candidateId);

    }).then(function(candidate){

        var voteCount = candidate[2];
        assert.equal(voteCount,1,"increment the candidate's vote
count");
    });
}); //it

```

- **electionInstance.vote(candidateId,{from : accounts[0]});**
pass in the candidate ID that we vote for and we pass in the function metadata like we saw on the console and we specify the account who is voting this is going to be the voter
- **[contract("Election", function(accounts)]** if you Remember earlier the accounts come in whenever we create this contract test here so once we cast the vote you know will trigger a promise chain and the return value of the vote function like we see on console upper picture
- **return electionInstance.voters(accounts[0]);**
Read our voters mapping that we created . we read account out of that mapping and it return boolean value **then(function(voted){ [voted = boolean value]**
- If true then voted = true and display marked as voted **assert(voted,"the voter was marked as voted");**
- **return electionInstance.candidates(candidateId);**
}).then(function(candidate){

Fetch out voter(candidateId) from mapping voters and return it in next function candidate

- **var voteCount = candidate[2];**
assert.equal(voteCount,1,"increment the candidate's vote count");
FETCH CANDIDATE VOTE COUNT AND CHECK IF VOTE COUNT FOR VOTER TO ANY CANDIDATE IS INCREMENTED BY ONE OR NOT
- Solidity if statement - **require()**:
If true : then run entire func
Else stop from here
- **require(!voters[msg.sender]):**
If the voter hasn't voted yet then show false and opp (!) of that is true
check this address not in voters mapping
 // require that they haven't voted before
 require(!voters[msg.sender]):
- // require a valid candidate

```
require( _candidateId > 0 && _candidateId <= candidatesCount );
```

Check if the candidateId is greater than 0 and less than or equal to candidatesCount

now test

- `Election.deployed().then(function(i){app = i}`
- `app.vote(1,{from : web3.eth.accounts[0]})`
- `var a`
- `web3.eth.getAccounts().then(e=>{a=e[0];})`
- `a`
- `app.vote(1,from:{a})`

[illegible]

If vote again using this gives exception

- `app.vote(1, from:{a})`

- `app.vote(1,{from : a})` //it will work as 1 is valid candidate Id

Test Voting Function

We want to test two things here:

1. Test that the function increments the vote count for the candidate.
2. Test that the voter is added to the mapping whenever they vote.

Next we can write a few test for our function's requirements. Let's write a test to ensure that our vote function throws an exception for double voting:

THIS TEST IS FOR WRONG CANDIDATE ID 99 INSTEAD OF CANDIDATE ID 1 to vote

```
it("throws an exception for invalid candidates",
function() {
    return Election.deployed().then(function(instance) {
        electionInstance = instance;
        return electionInstance.vote(99, { from: accounts[1]
    })
    }).then(assert.fail).catch(function(error) {
        assert(error.message.indexOf('revert') >= 0, "error
message must contain revert");
        return electionInstance.candidates(1);
    }).then(function(candidate1) {
        var voteCount = candidate1[2];
        assert.equal(voteCount, 1, "candidate 1 did not
receive any votes");
        return electionInstance.candidates(2);
    }).then(function(candidate2) {
        var voteCount = candidate2[2];
        assert.equal(voteCount, 0, "candidate 2 did not
receive any votes");
```

```
});  
});
```

- `Assert.fail` - we can check for error for assert fail
- `catch(function(error) = catch this error and return a callback function`
- `assert(error.message.indexOf('revert') >= 0, "error message must contain revert");` - inspect our error object. We can check its msg and check if this msg have substring "revert" inside it
- If this `require(!voters[msg.sender]): gets False then not do voteCount`
- `then(function(candidate1) {`
- `var voteCount = candidate1[2];`
- `assert.equal(voteCount, 1, "candidate 1 did not receive any votes");`
- Also `voteCount == 1` as we already give vote one time to this candidateID

We can assert that the transaction failed and that an error message is returned. We can dig into this error message to ensure that the error message contains the "revert" substring. Then we can ensure that our contract's state was unaltered by ensuring that the candidates did not receive any votes.

Now let's write a test to ensure that we prevent double voting:

```
it("throws an exception for double voting", function() {  
    return Election.deployed().then(function(instance) {  
        electionInstance = instance;  
        candidateId = 2;  
        electionInstance.vote(candidateId, { from:  
accounts[1] });  
        return electionInstance.candidates(candidateId);  
    }).then(function(candidate) {  
        var voteCount = candidate[2];  
        assert.equal(voteCount, 1, "accepts first vote");  
        // Try to vote again
```

```

    return electionInstance.vote(candidateId, { from:
accounts[1] });
    }).then(assert.fail).catch(function(error) {
    assert(error.message, "error message must contain
revert");
    return electionInstance.candidates(1);
    }).then(function(candidate1) {
    var voteCount = candidate1[2];
    assert.equal(voteCount, 1, "candidate 1 did not
receive any votes");
    return electionInstance.candidates(2);
    }).then(function(candidate2) {
    var voteCount = candidate2[2];
    assert.equal(voteCount, 1, "candidate 2 did not
receive any votes");
    });
    });

```

- First, we'll set up a test scenario with a fresh account that hasn't voted yet i.e 2
- Then we'll cast a vote on their behalf.
- Then we'll try to vote again. We'll assert that an error has occurred here.
- We can inspect the error message, and ensure that no candidates received votes, just like the previous test.
- Fetch all account in array format :
let accounts = await web3.eth.getAccounts()
- ```
assert(error.message, "error message must contain
revert");
```
- By using this it will work

```
C:\Windows\System32\cmd.exe

C:\project\Ethereum Voting>truffle test
Using network 'development'.

Compiling your contracts...
=====
> Compiling .\contracts\Election.sol
> Compiling .\contracts\Migrations.sol
> Artifacts written to C:\Users\sarry\AppData\Local\Temp\test--5624-QnrSOIzZ74pa
> Compiled successfully using:
 - solc: 0.5.16+commit.9c3226ce.Emscripten.clang

Contract: Election
 ✓ initializes with three candidates (253ms)
 ✓ it initializes the candidates with correct values (583ms)
 ✓ allows a voter to cast a vote (1669ms)
 ✓ throws an exception for invalid candidates (1787ms)
 ✓ throws an exception for double voting (1723ms)

5 passing (6s)

C:\project\Ethereum Voting>
```

Now we change some code of client side application

Let's add a form that allows accounts to vote below the table in our "index.html" file:

```
<form onSubmit="App.castVote(); return false;">
 <div class="form-group">
 <label for="candidatesSelect">Select Candidate</label>
 <select class="form-control" id="candidatesSelect">
 </select>
 </div>
 <button type="submit" class="btn btn-primary">Vote</button>
 <hr />
</form>
```

Let's examine a few things about this form:



- We create the form with an empty select element. We will populate the select options with the candidates provided by our smart contract in our "app.js" file.
  - The form has an "onSubmit" handler that will call the "castVote" function. We will define this in our "app.js" file.
- 
- Form that allow our account to vote it have a select input
  - Append item in select input correspond to each candidate in smart contracts
  - It will have **onSubmit** handler that call a castVote **App.castVote() in app.js file**

Now let's update our app.js file to handle both of those things.

- First we list all the candidates from the smart contract inside the form's select element.
- Then we'll hide the form on the page once the account has voted.
- We'll update the render function to look like this:

From line 60 to 62

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

From line 71 to 73

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

From line 78 to 89

```
return electionInstance.voters(App.account);
```

```
 }).then(function(hasVoted) {
 // do not allow a user to vote if voted
 if(hasVoted)
 {
 $('form').hide();
 }
 loader.hide();
 content.show();
 }).catch(function(error) {
 console.warn(error);
 });
```

- App.account = read current account to see if they exist in mapping .if they exist and already voted then it will return a boolean value of true
- If its true i.e hasVoted is true then it will hide voting option i.e form

Next, we want to write a function that's called whenever the form is submitted:

```

castVote: function() {
 var candidateId = $('#candidatesSelect').val();

 App.contracts.Election.deployed().then(function(instance)
 {
 return instance.vote(candidateId, { from:
App.account });
 }).then(function(result) {
 // Wait for votes to update
 $("#content").hide();
 $("#loader").show();
 }).catch(function(err) {
 console.error(err);
 });
}

```

First, we query for the candidateId in the form.

When we call the vote function from our smart contract, we pass in this id, and we provide the current account with the function's "from" metadata.

This will be an asynchronous call. When it is finished, we'll show the loader and hide the page content. Whenever the vote is recorded, we'll do the opposite, showing the content to the user again.

- app.castVote in the form
- Get user selected CandidateId from form and store in var
- Call the vote func with candidateId and pass func metadata with from : the current account for our app
- This will be a asynchronous call so we'll execute a promise chain
- For now hide the content and show the loader