*DEVELOPMENT OF DECENTRALISED TRADING BALLOT*

**Research on how to approach project i.e. what to use, how things work.**

<https://www.freecodecamp.org/news/developing-an-ethereum-decentralized-voting-application-a99de24992d9/>

Freecodecamp enabled me to develop on my knowledge on the rules and logic of the Ethereum blockchain, as well as what would be required to create this application.

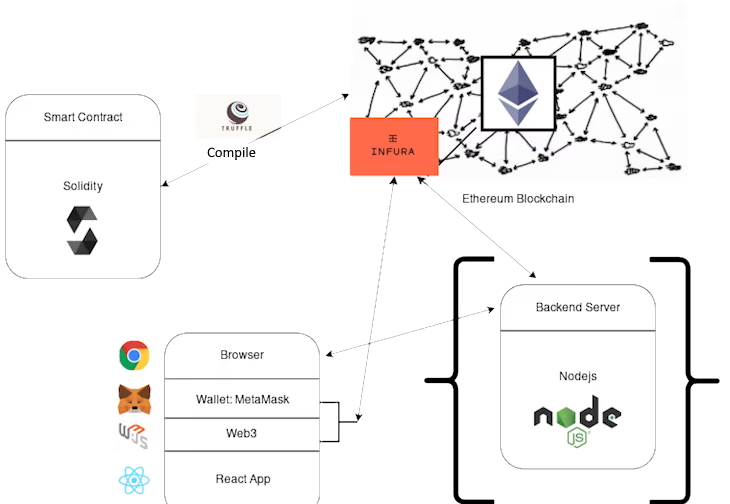
My knowledge of blockchain technology enabled me to discover the possibilities of truffle pet-shop, a template start code for creating decentralised applications.

<https://boostlog.io/_junp1234/how-to-make-a-dapp-of-a-pet-shop-using-solidity-and-truffle-5ab200c50814730093a2ec91/>

This site states 7 steps for concrete flow of implementation of a Dapp, which allowed me to gather knowledge of the step by step process of creating this application.

What really helped me map out what I needed to do was a figure obtained from:

<https://blog.logrocket.com/interacting-smart-contracts-via-nodejs-api/>



This figure helps convey the general purpose for each incorporation of application to enable this project to function.

To exercise my knowledge on DApp development I began training with the tutorial as shown on the website below:

<https://www.moesif.com/blog/blockchain/ethereum/Tutorial-for-building-Ethereum-Dapp-with-Integrated-Error-Monitoring/>

I am aiming to create a decentralised voting system. The decentralised nature of blockchain technology ensures absolute anonymity and impossibility of vote manipulation.

To be able to create this, we must ensure we can operate on a network. Using the most popular Ethereum truffle development framework, we will connect to the Ethereum testing local blockchain ganache to enable for testing of the application.

Truffle will enable us to use solidity to test and create smart contracts.

Ganache is a testing facility platform.

To utilise the Ethereum blockchain, we will use metamask to enable the browser to connect to it.

To determine what would make a decentralised voting system unique, I viewed various opinions online and came to this on StackOverflow which specified a number of requirements to make the program worthwhile and work. These were the ability to prevent the selling of votes, prevent the stealing of votes, promote safer voting, and how an absentee may vote.

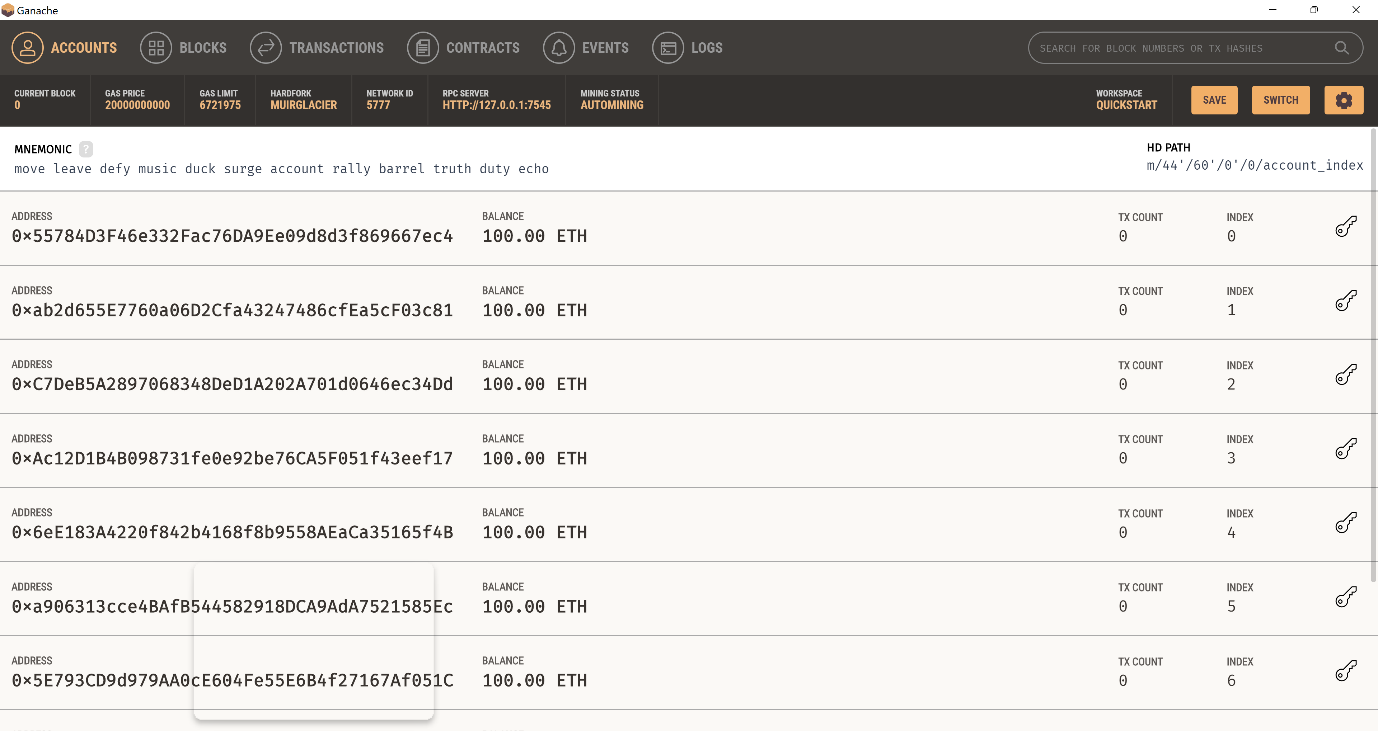
<https://politics.stackexchange.com/questions/25618/decentralized-electoral-system>

I based much of my idea around this decentralised ballot around those 4 principles and made it a priority to solve each one using decentralised technology.

**Beginning of the development and progress (explaining decision making is important).**

**(How you came to decisions is important.)**

We will import the Ethereum package from package control, into sublime text.

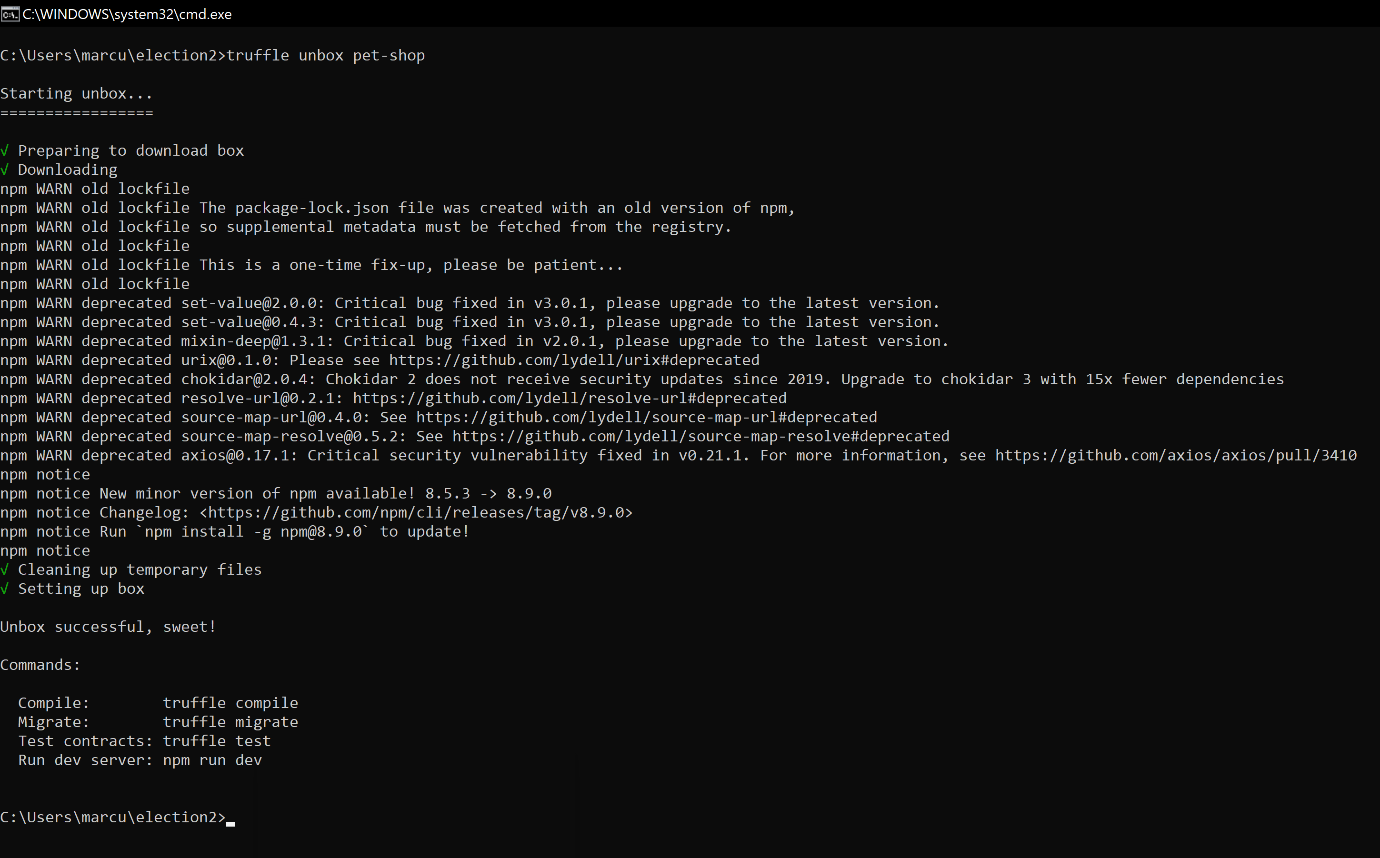


Each account provided by ganache contains 100 Ethereum.

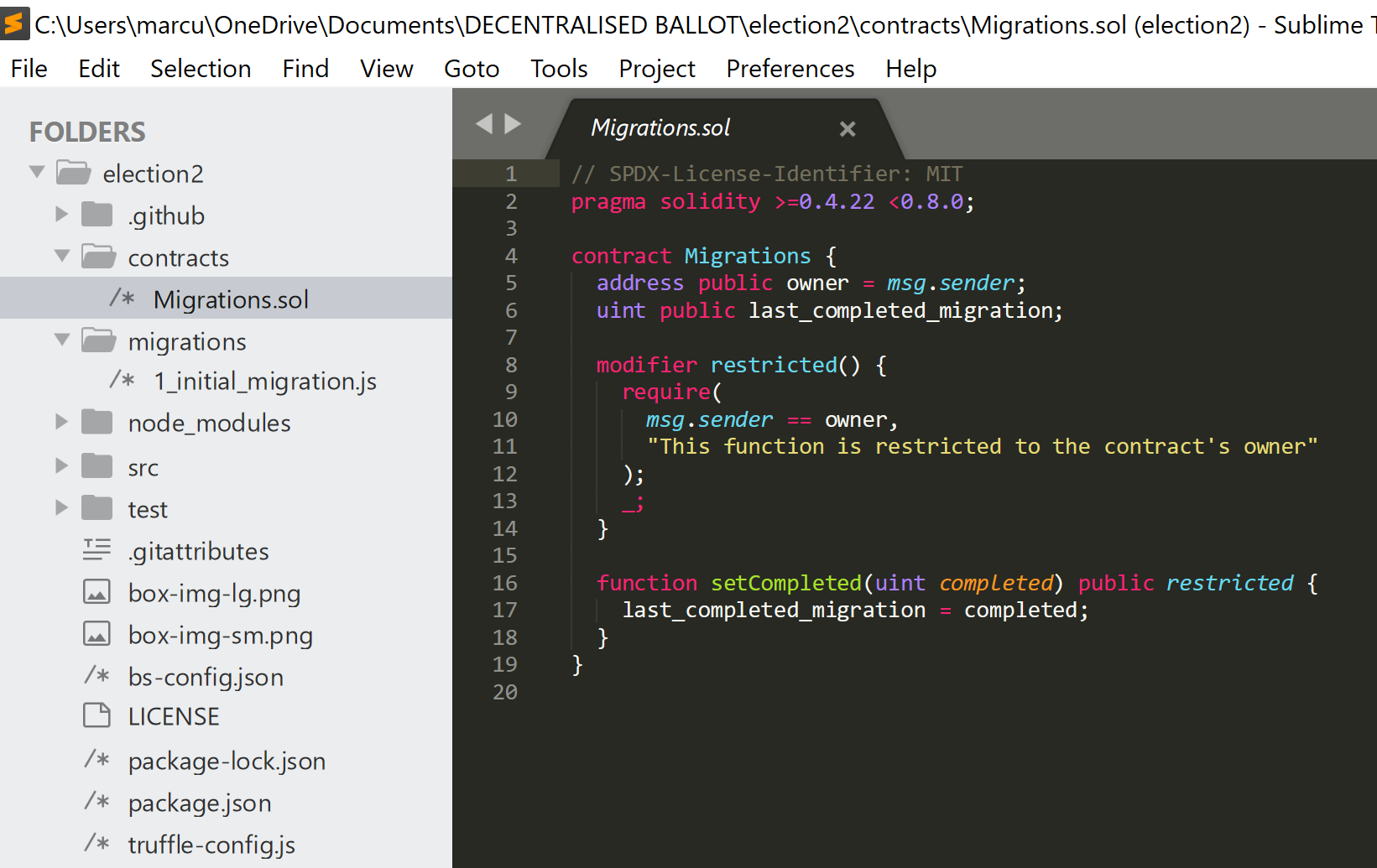
The addresses represent a singular voter in an election.

To enhance progress and help finish off the project in the specific timeframe, we will use a truffle ‘box’ which is full of pre made code to get projects started faster. This specific box is called truffle ‘petshop’.

We need to unbox this within our directory containing this project.



The pet-shop box comes with a pre made ‘migrations.sol’ file. This file is important as it enables us to push the smart contracts/run the code on the Ethereum blockchain.



Additionally, you can see the pet-shop box includes an ‘src’ folder. This folder contains pre-created client-side configurations, such as a HTML and CSS file to get started.

The configuration structure has now been configured.

**Next key stage of development: Smart contracts**

Smart contract applications are programmed using solidity. Within solidity it is essential that the version is always specified at the beginning, to enable compatibilities and the use of certain functions and packages. To begin we must make a smart contract which is a predetermined set of rules and code that executes when certain conditions are met.

To begin we create ‘Election.sol’ to write the smart contracts that will enable the application to function. We need to establish variables, which we do through the function ‘contract’ followed by the variable name. Then the variables need to be validated and initialised when the code is executed, which is why constructers are essential.

<https://cryptomarketpool.com/constructor-in-solidity-smart-contracts/>

This site proves useful in explaining the importance and use for a constructor.

To be able to run the constructor on the blockchain, one must make it public rather than private.

If the constructor is not functioning then it is likely because it has not been specified whether it is permissioned to initialise the code, which is the purpose of regarding it as ‘public’ or ‘internal’.

<https://ethereum.stackexchange.com/questions/30223/should-the-constructor-function-be-public>



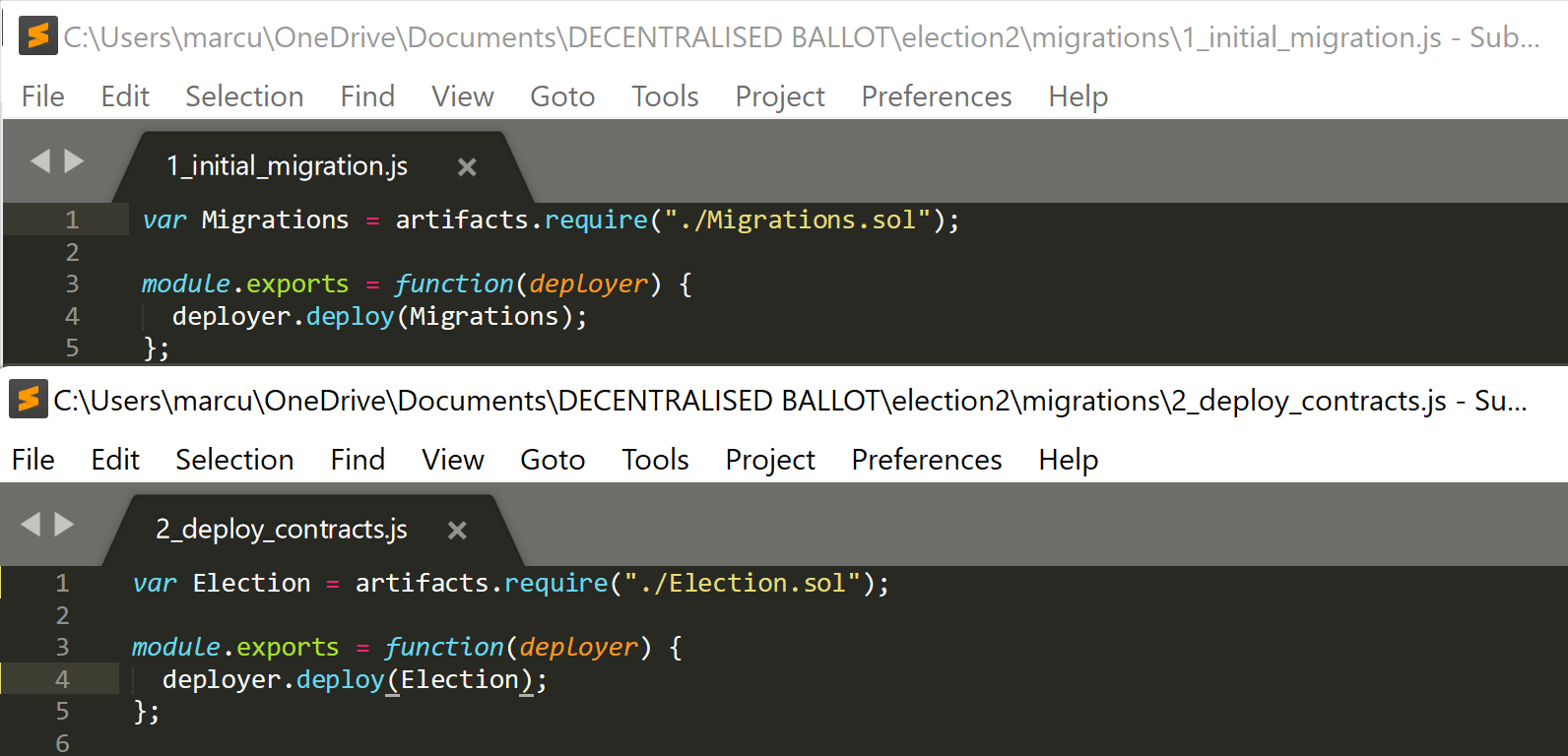
The following code demonstrates three things: line 4 shows the declaration of the state variable’s data type, which is required within the solidity programming language. Line 5 declares the constructor as public to enable it to initialise code to function on the blockchain. Line 6 is intended to show the candidate data when the code is executed, the ‘candidate’ variable value will identify to the “Candidate 1” its value/how many votes it has received.

The ability to interact with the contracts through the client side and testing side is through node.js. Using the pet-shop pre-installed node.js compatible file called *‘1\_initial\_migration.js’,* we can enable interaction. The purpose of the initial migrations is to push/deploy a contracts’ data to the blockchain through the back-end server., it helps compile and pack together the smart contracts to be safely initialised on the Ethereum blockchain it is being deployed to. The following link explains migrations and their importance in more detail. It is essentially the delivery mechanism.

<https://www.sitepoint.com/truffle-migrations-explained/>

Truffle has a simple method of migration file configuration, in which it runs files in numerical order, therefore it is important to establish a numerical at the beginning of file names.

Since the pet-shop package already configures the method of migration for us, it is now up to me to do the same for any other contracts I wish to deploy. This is made easy as all it takes is to copy and paste the *‘1\_initial\_migration.js’* code into another file that will act as the delivery mechanism for deploying the Election.sol smart contract. Therefore, the name ‘migration’ must be changed to Election.



As I am determined to become a blockchain developer, I intend to analyse and evaluate the purpose of every part of code, dissecting it completely.

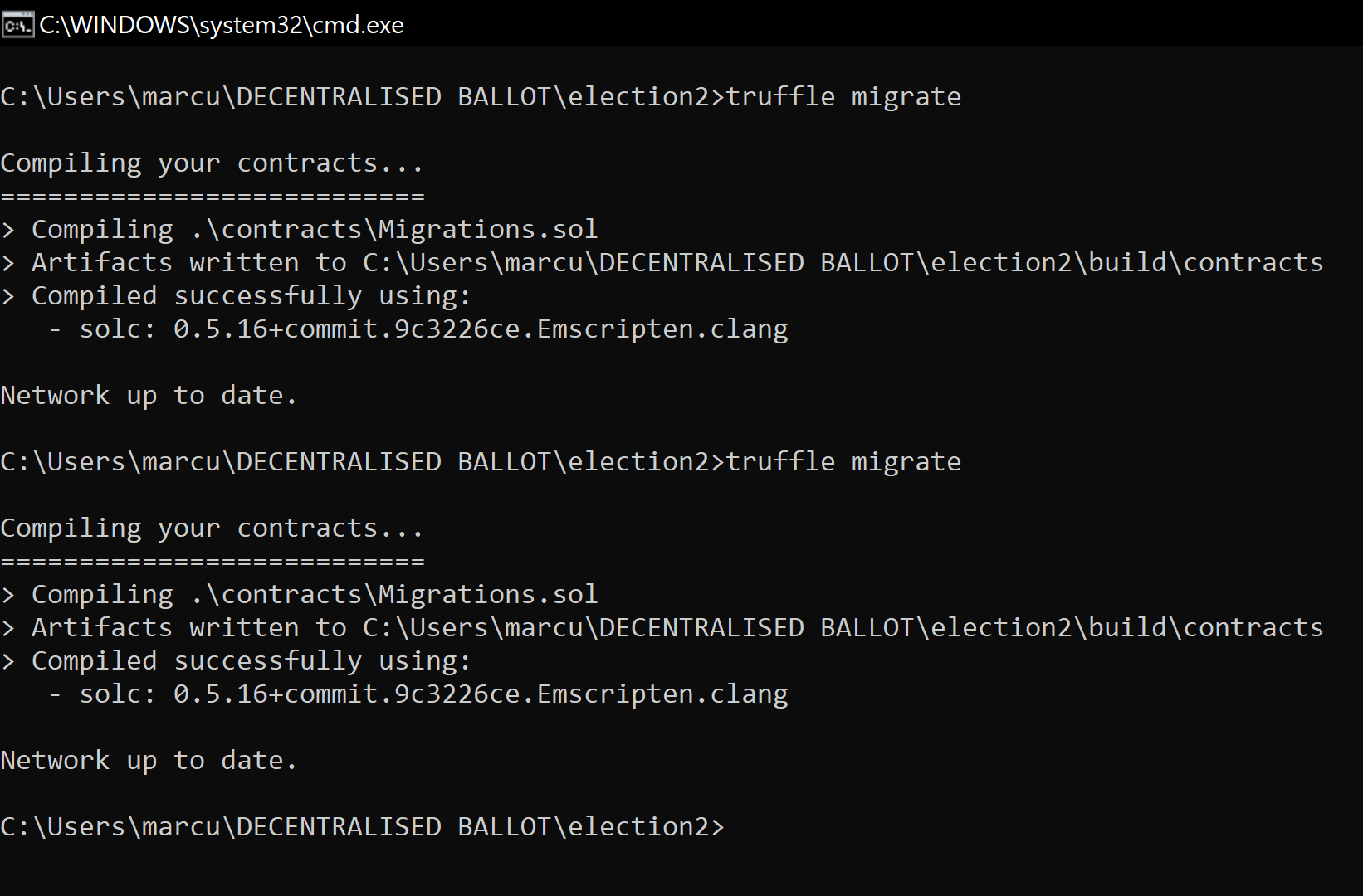
The purpose of the code is as follows:

The designation of *Election* as a variable establishes the variable that is being configured, which is located in the Election.sol file, which is specified at the end of the line.

*Artifacts.require* is a function special to truffle. Truffle allocates artifacts which enables a contract to be interactive. It wraps up all the code from the specified contract and then deploys it using the migration. Essentially, a smart contract contains the instructions, the migration file is the method of delivery to the blockchain, and the artifact is unpacked with all the data that’s been wrapped up.

<https://www.ionos.co.uk/digitalguide/websites/web-development/what-is-a-wrapper/>

<https://trufflesuite.com/docs/truffle/getting-started/interacting-with-your-contracts/>

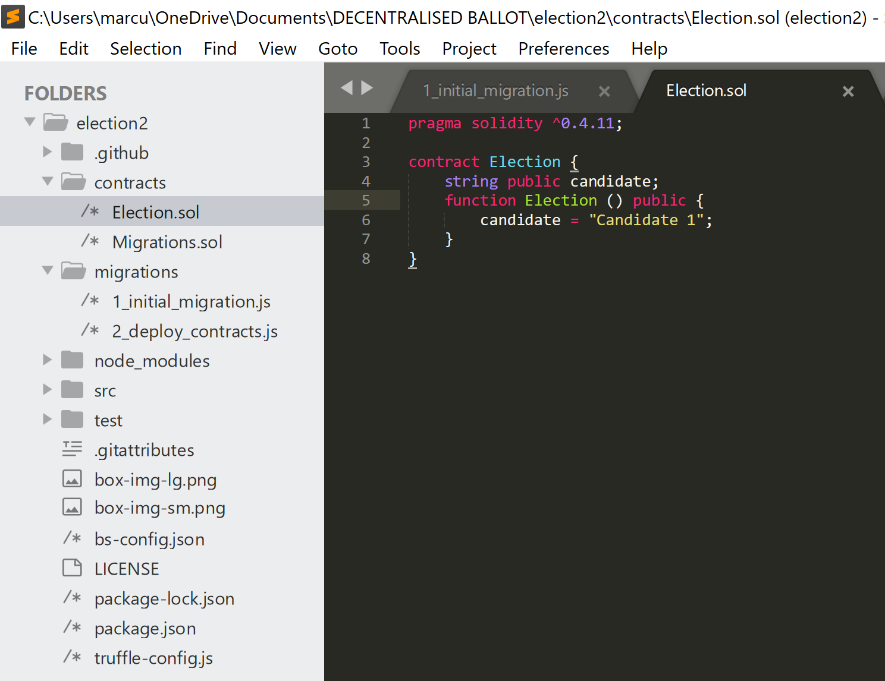


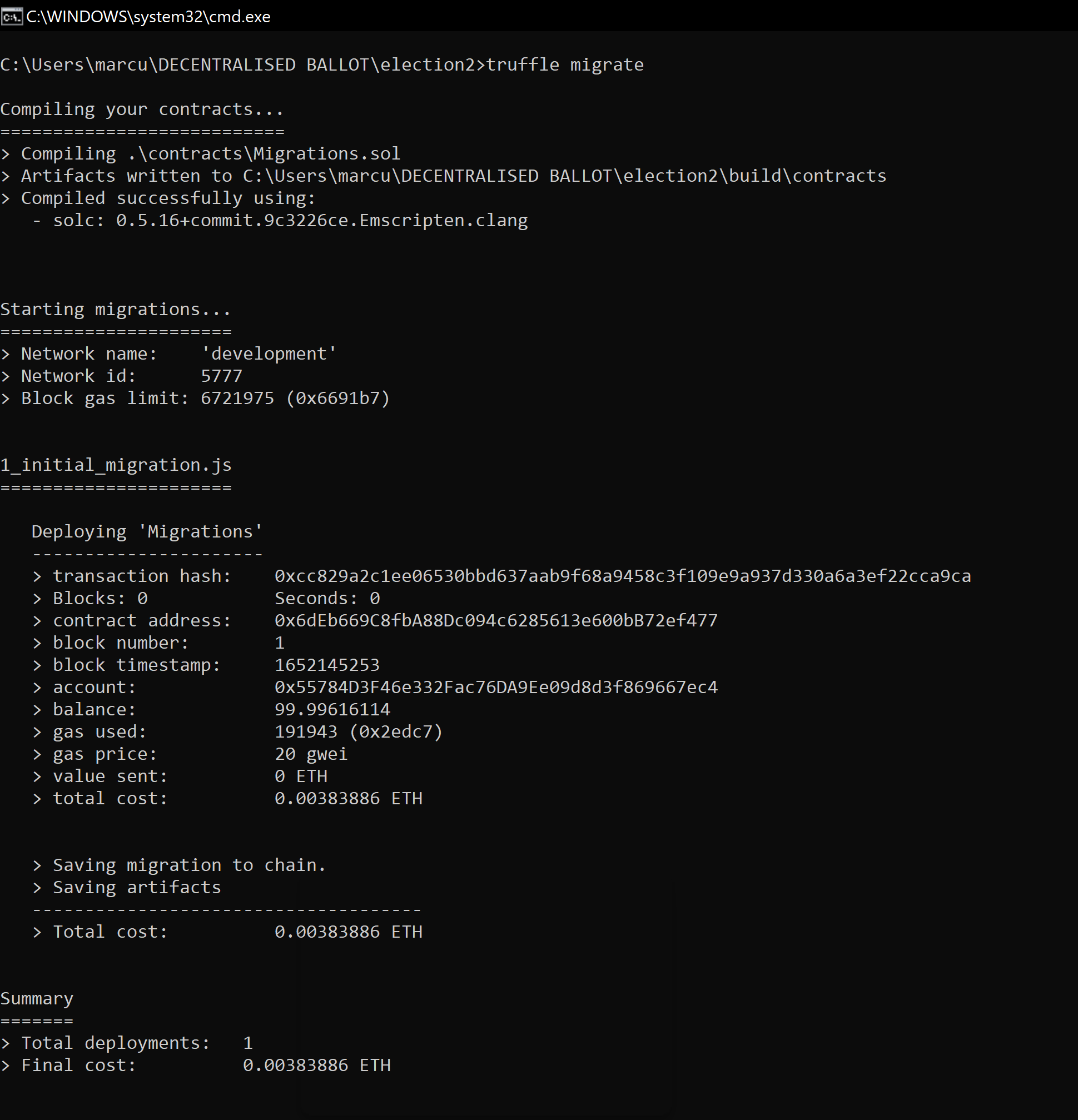
The test as shown is somewhat a success, However I found I should be able to view balances and the gas fees data should be shown. I researched extensively as to why there is an error so early on, through consistent analysis of the code, I concluded the data of the state varibales was not being shown, which led to the google search “why can’t I see my variable data in solidity?”.

This search leads to a top answer: <https://ethereum.stackexchange.com/questions/44893/how-do-i-see-the-value-of-a-string-stored-in-a-private-variable>

They discuss the need of specifying the visibility of a variable, in which I needed to set it as public to be able to view the data when the program is run.

<https://medium.com/coinmonks/visibility-in-solidity-e758a4739c95>

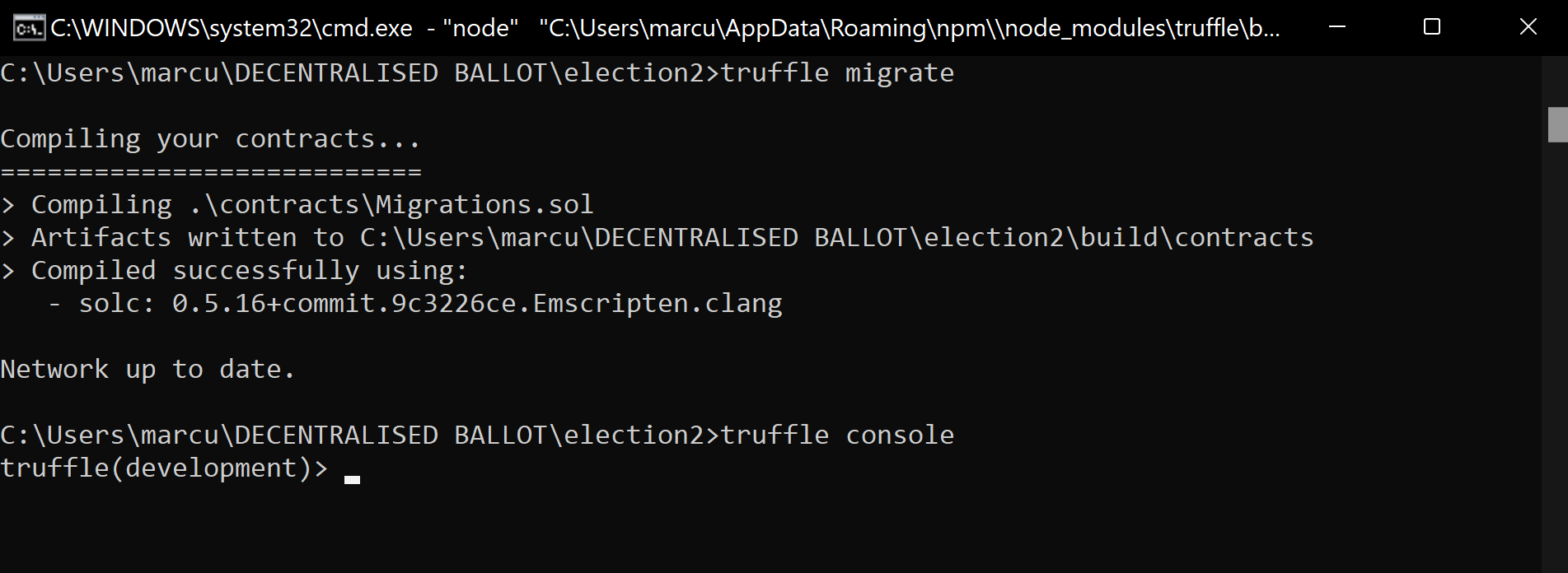




Part of the reason I was able to recognise this error was through tutorials such as that from ‘rsk’:

<https://developers.rsk.co/webinars/202007-005/>

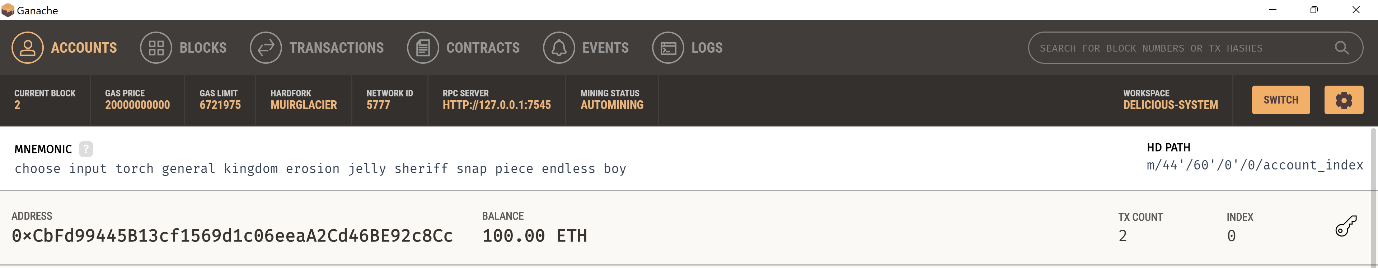
Time to enter the truffle testing suite, via the truffle console.

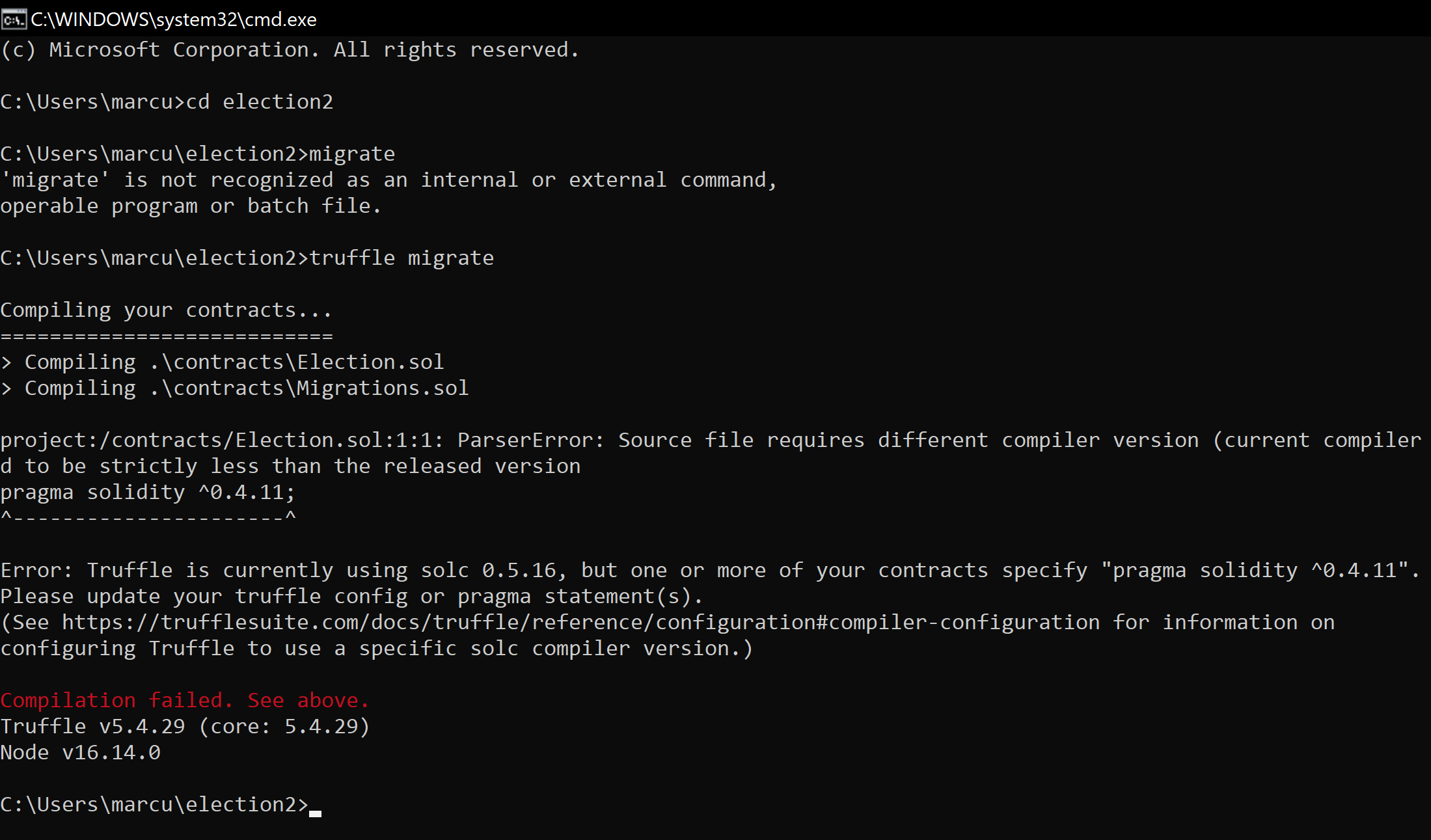


It was decided that testing every aspect of the code individually wasn’t feasible, therefore I decided to wrap the whole source code into a variable called ‘app’. This means instead of compiling and running each smart contract individually through the console, running the app will initialise it all at the same time.

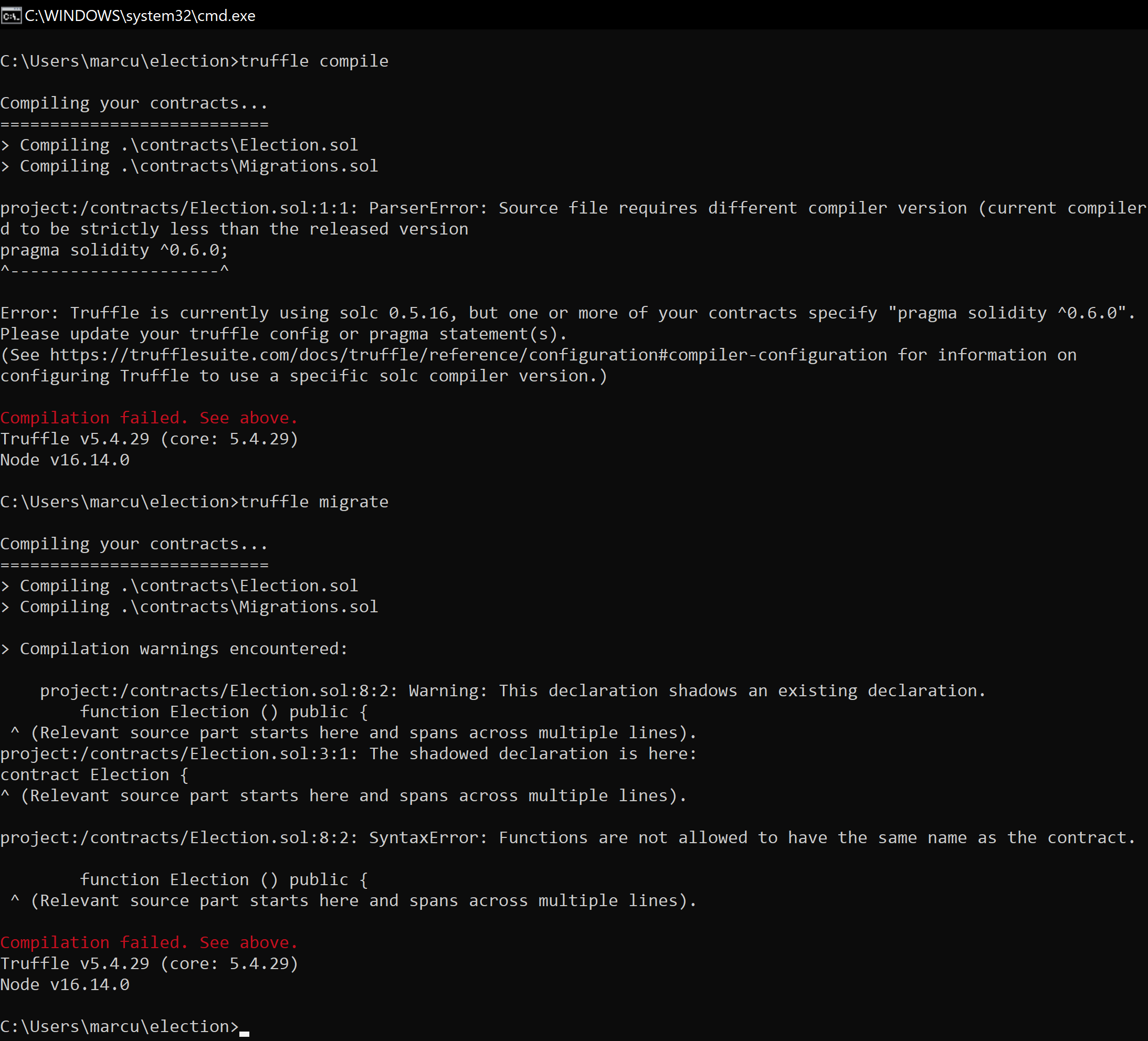
(SHOW WRAP APP GIF)

I realised the transactions were going through ganache, so the blockchain environment was functioning as intended.



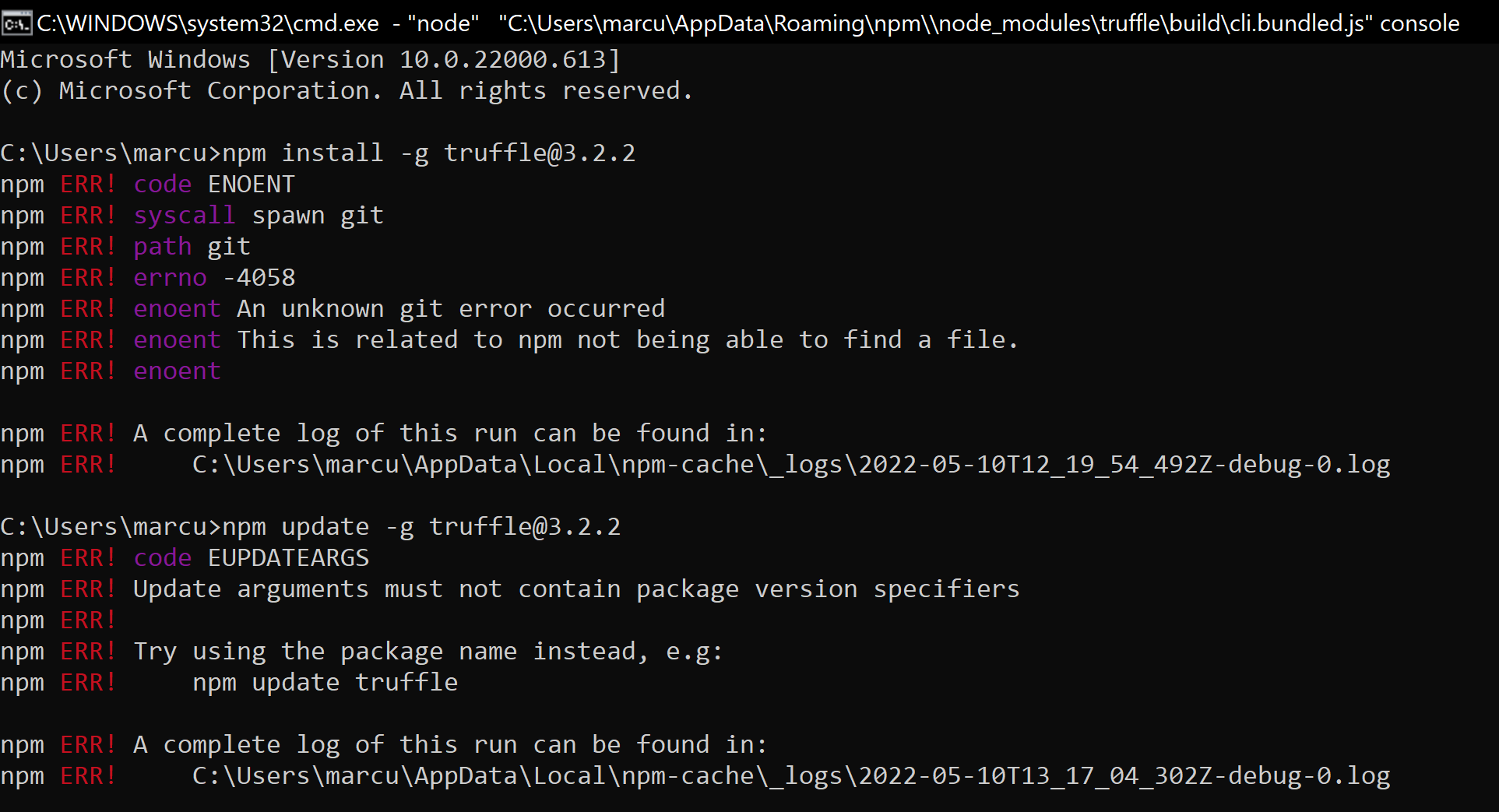


To solve this error, I ensured the solidity version matched what was specified in the error message but then was met with this:



<https://www.npmjs.com/package/truffle?activeTab=versions>

**Error concerning version compatibility**

****

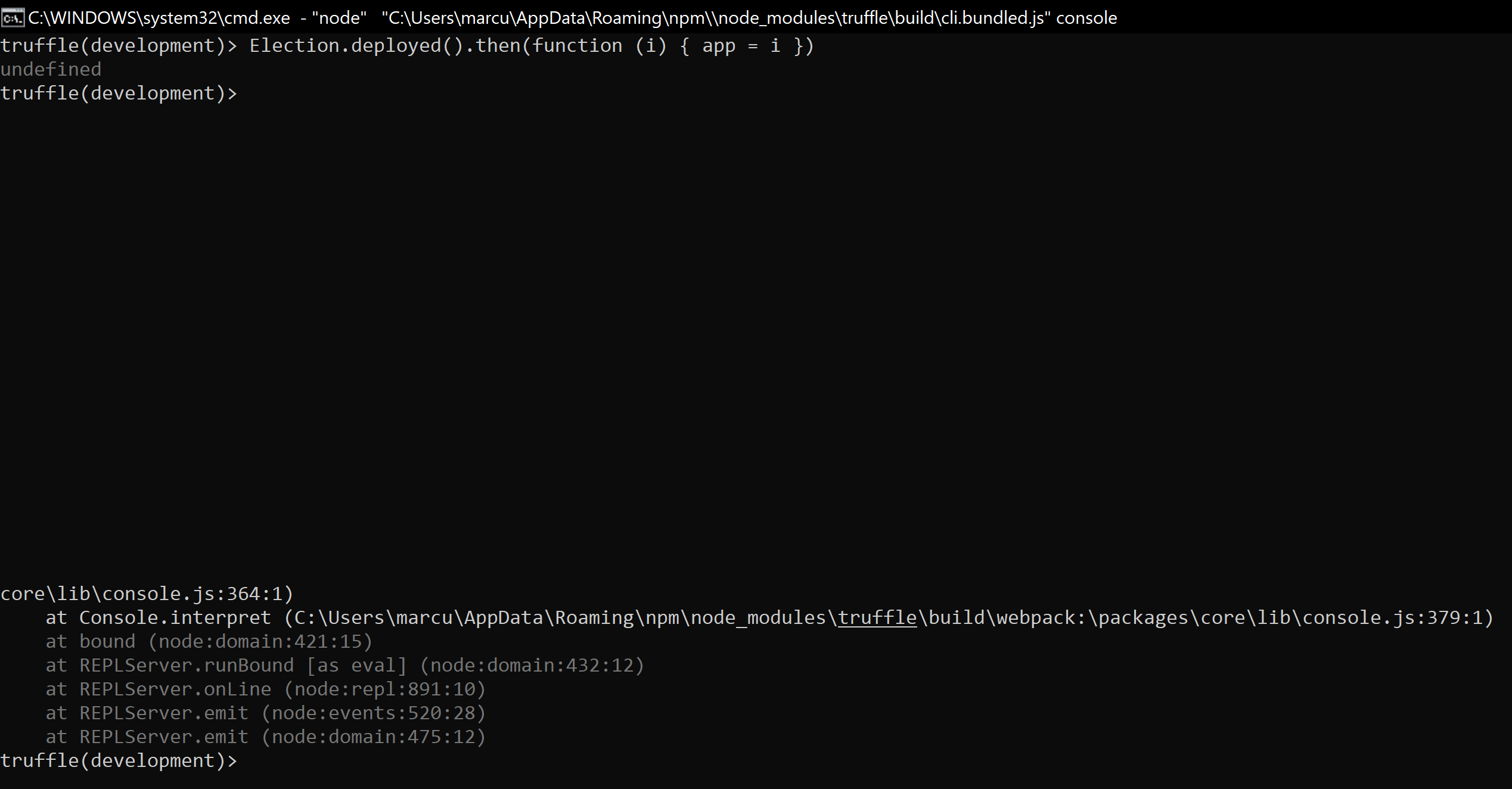
This error shows me trying to change the truffle version, in order to make it compatible with the latest version of solidity. I changed both the solidity and truffle framework versions multiple times through trial and error to eventually come to a compatibility consensus.



I needed to configure the specific compiler version for solidity within the built in truffle-config.js file. This is to ensure the program recognises and accepts the solidity version.

<https://ethereum.stackexchange.com/questions/67870/app-candidates-is-not-a-function-getting-these-error-while-caling-the-functio>

To wrap the entirety of the code as an app, the file must be compressed into an app variable. Using the link above, I was able to perform this task.



It is time to determine whether we can interact with the app within the console, then we can begin extending the app possibilities.

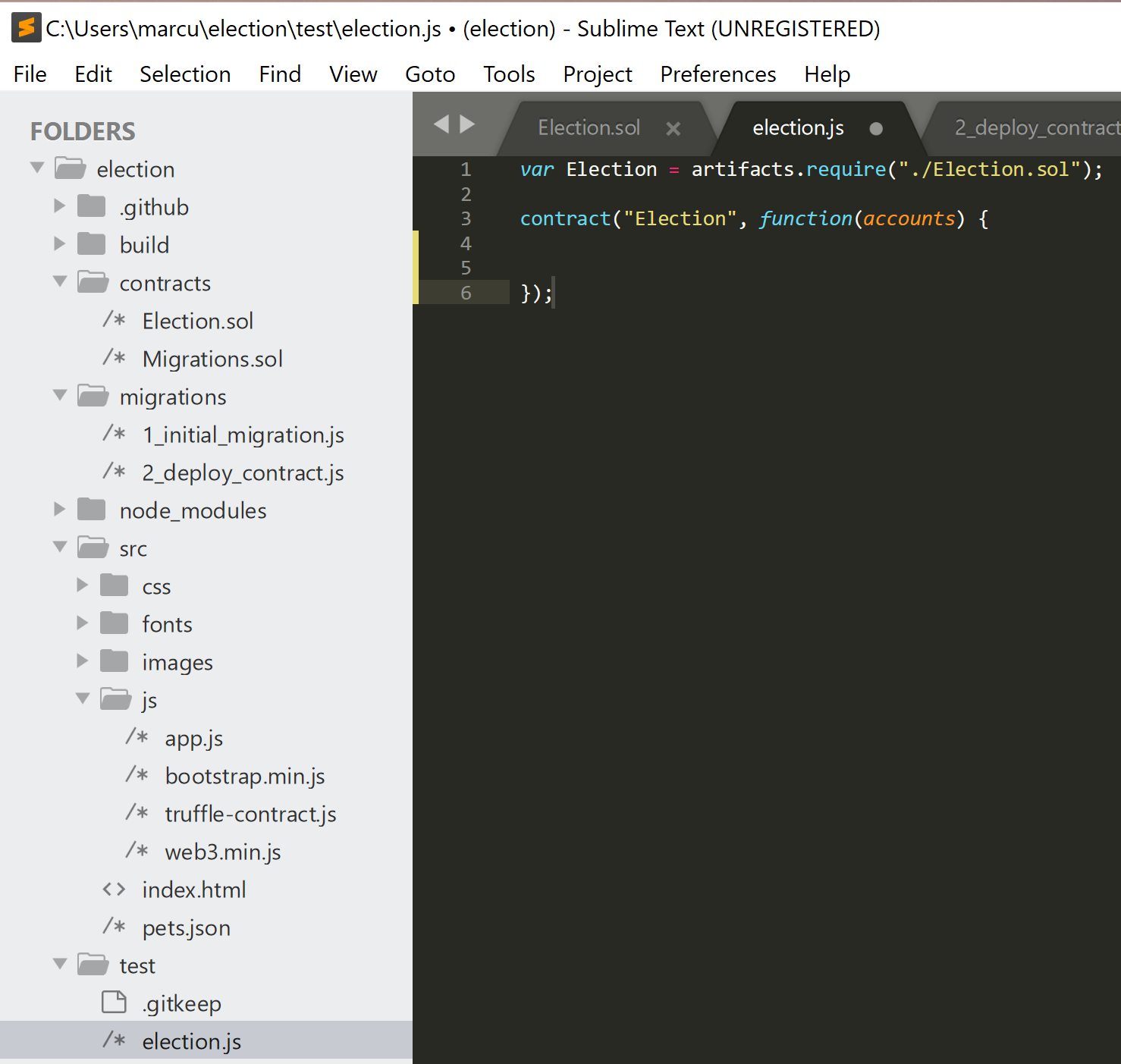
Now we move onto using mocha testing framework and the chai assertion library. Mocha is a testing library to node and required to conduct the further testing of the application. It is necessary for test executions, however we require a verification system in place, to verify data is correct. Therefore, we combine the use of Mocha with Chai, an assertion library.

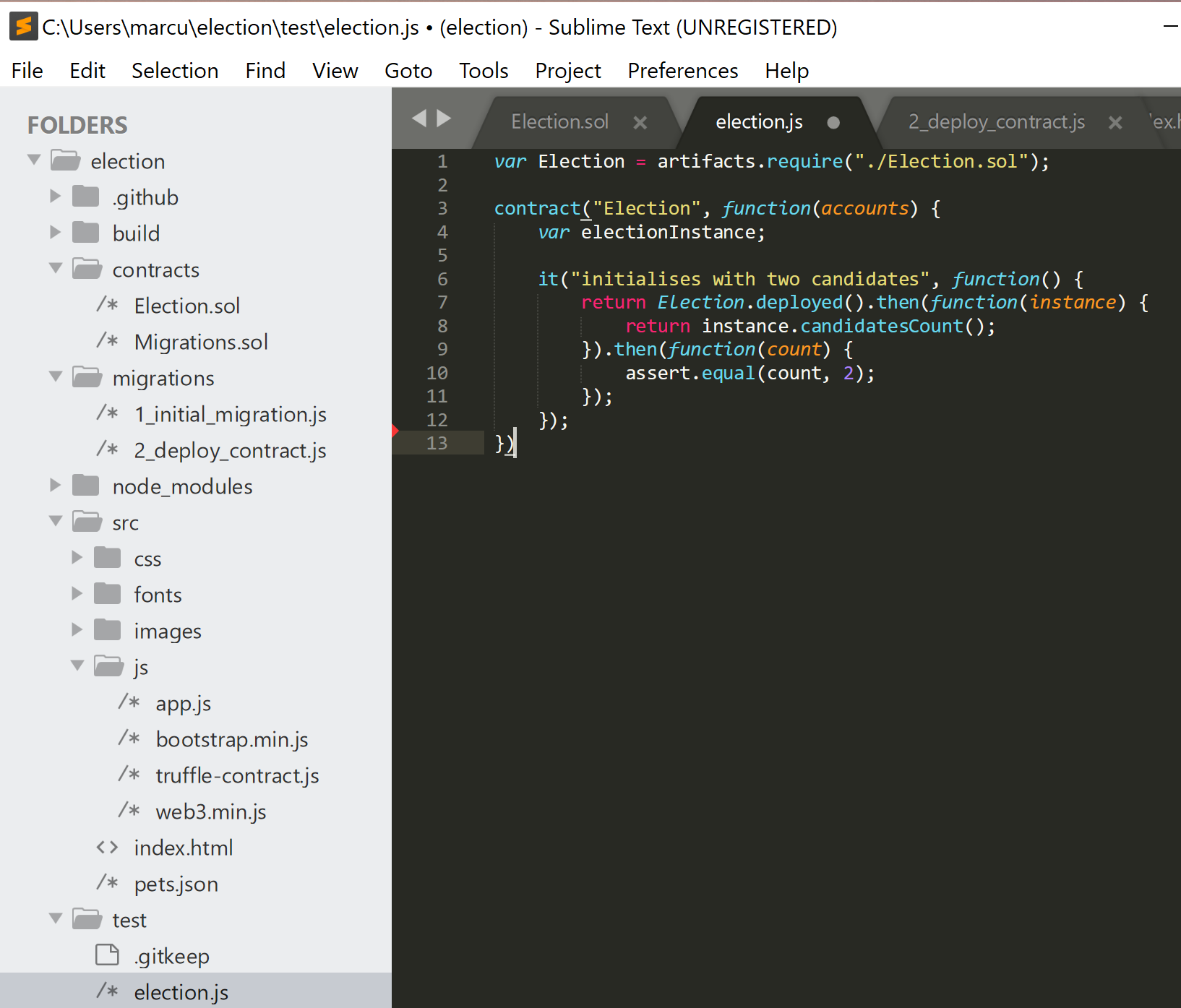
We will be using the JavaScript testing framework to enable us to test the app from a user’s perspective, rather than from the console.



We designate ‘Election’ as an artifact as it makes it an abstraction, to prevent unnecessary information on the client side of the application.

The next step is to declare our contract and fetch the data from our development environment (ganache) to be able to run the code with the proper information.

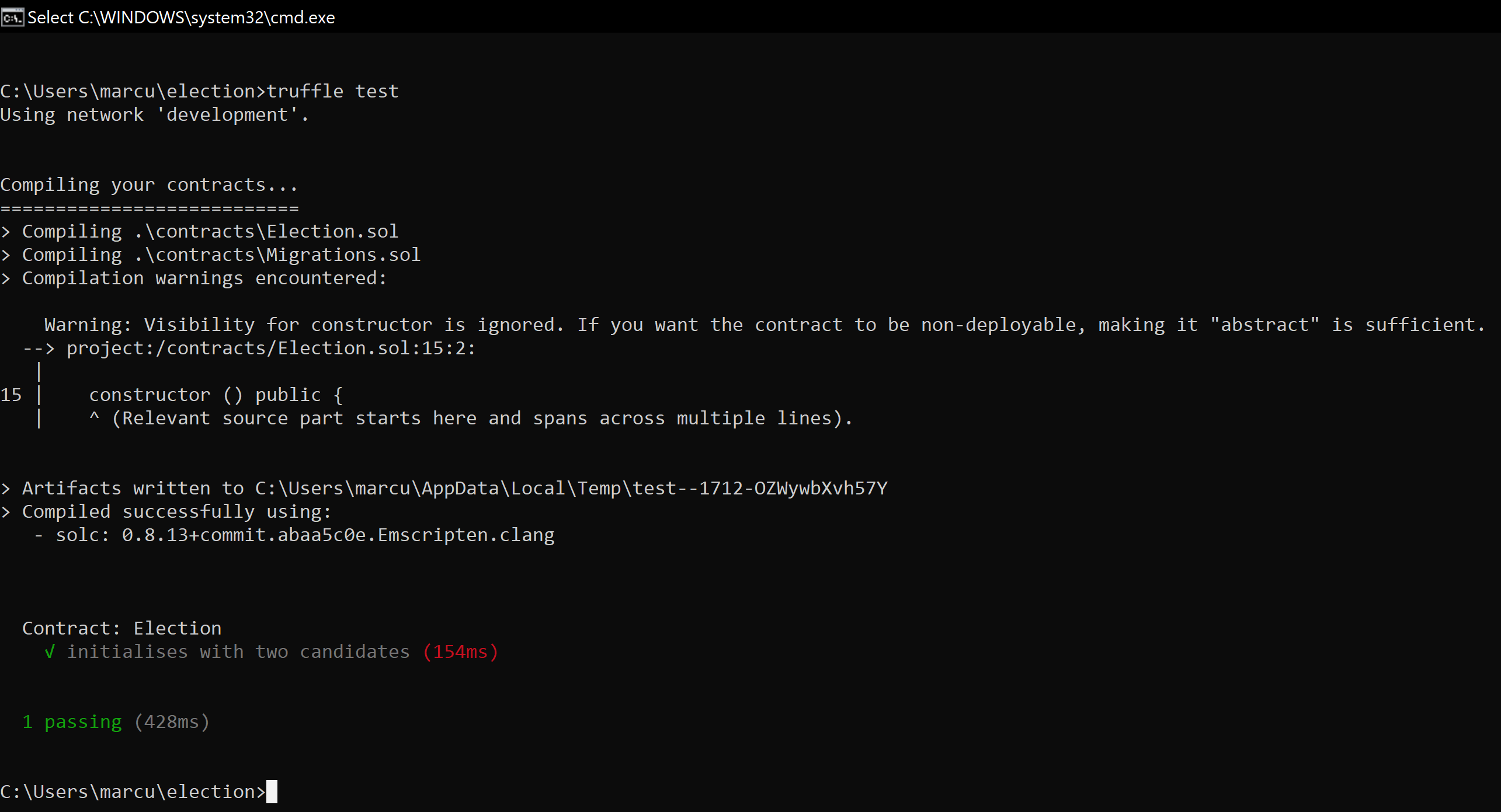


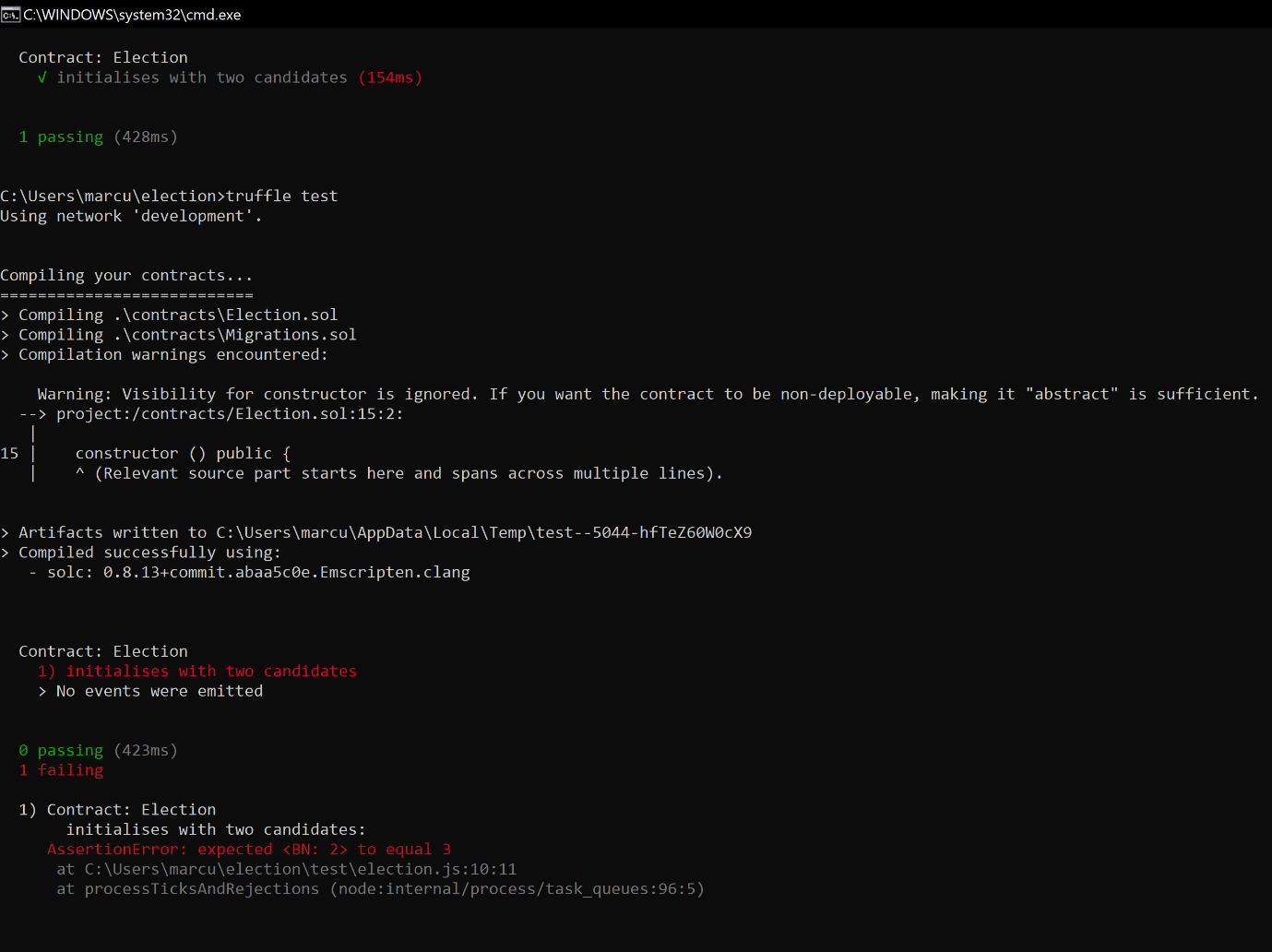


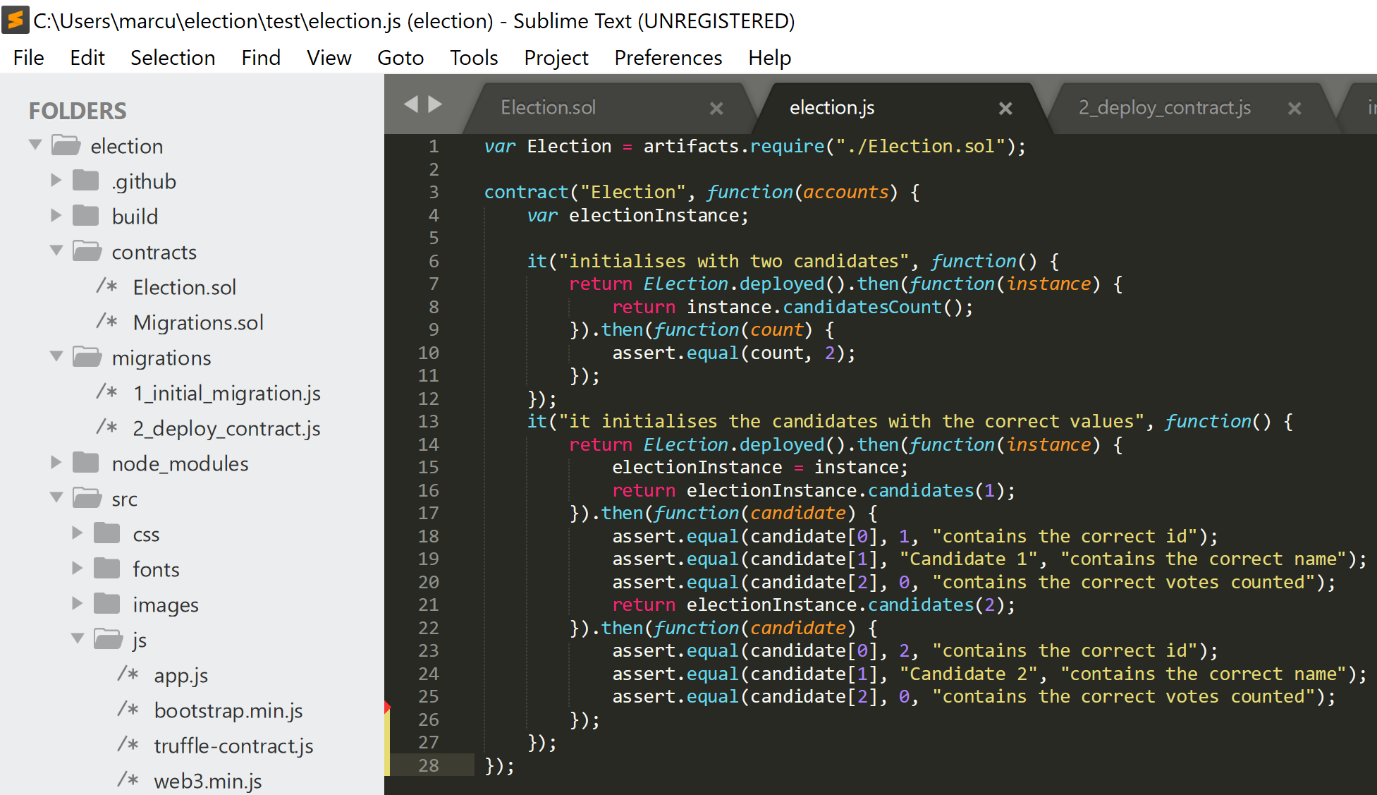
The code above demonstrates the implementation of mocha functions such as ‘it’. The use of ‘*assert.equal*’ is an example of the chai.

This code tests the program to look for inaccuracies. Where it says assert.equal(count, 2);’ this is an example of using the chai assertion library to verify that there is only 2 candidates. See example below:

TESTING WITH CHAI







Here is a demonstration of code that enables mocha and chai to test the following:

* The correct ID of each candidate.
* The correct name.
* The correct number of votes for each participant.

In order for mocha and chai to access the data that enables them to conduct these tests, they must be linked to the Election smart contract as well as the voter accounts operating in ganache. This is done through

*contract("Election", function(accounts) {*

*var electionInstance;*

The script is linked via line 1 to the Election.sol code, therefore containing the data from that file. The function(accounts) fetches the account data from ganache and the relevant data from the Election.sol file. Then it is all compiled into a variable ‘electionInstance’ to be used within the testing and initialisation code below it, thus enabling verification of candidate information.

it("it initialises the candidates with the 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], "Candidate 1", "contains the correct name");

assert.equal(candidate[2], 0, "contains the correct votes counted");

return electionInstance.candidates(2);

}).then(function(candidate) {

assert.equal(candidate[0], 2, "contains the correct id");

assert.equal(candidate[1], "Candidate 2", "contains the correct name");

assert.equal(candidate[2], 0, "contains the correct votes counted");

});

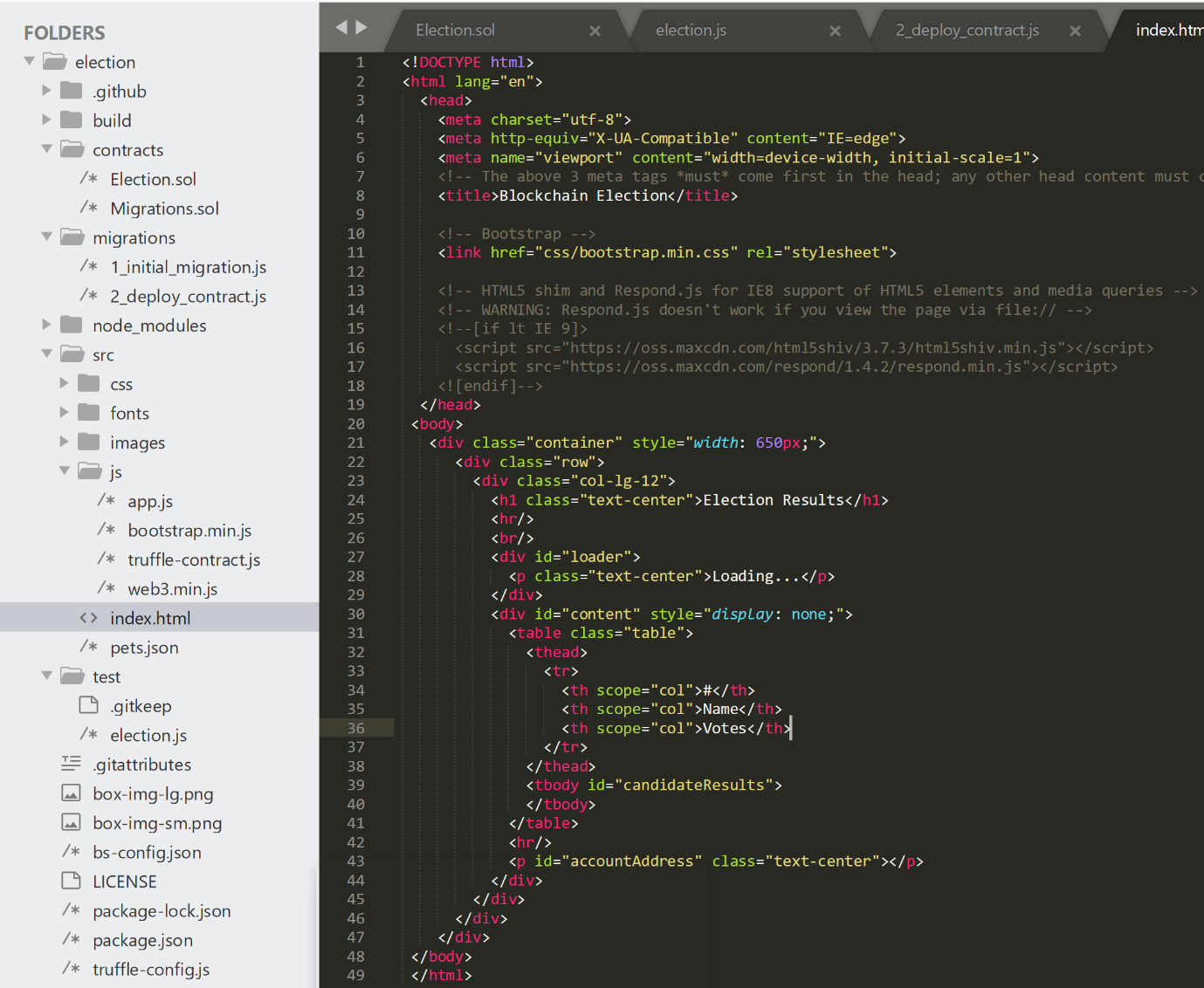
This code demonstrates the use of JavaScript promise chaining, necessary as the testing consists of several asynchronous operations. The candidates have their information validated one after the other in an *asynchronous* operation. In order for continuation of code operation after the first candidate’s information has been validated, the code performs a callback function through ‘electionInstance.candidates(1);’ and so on.

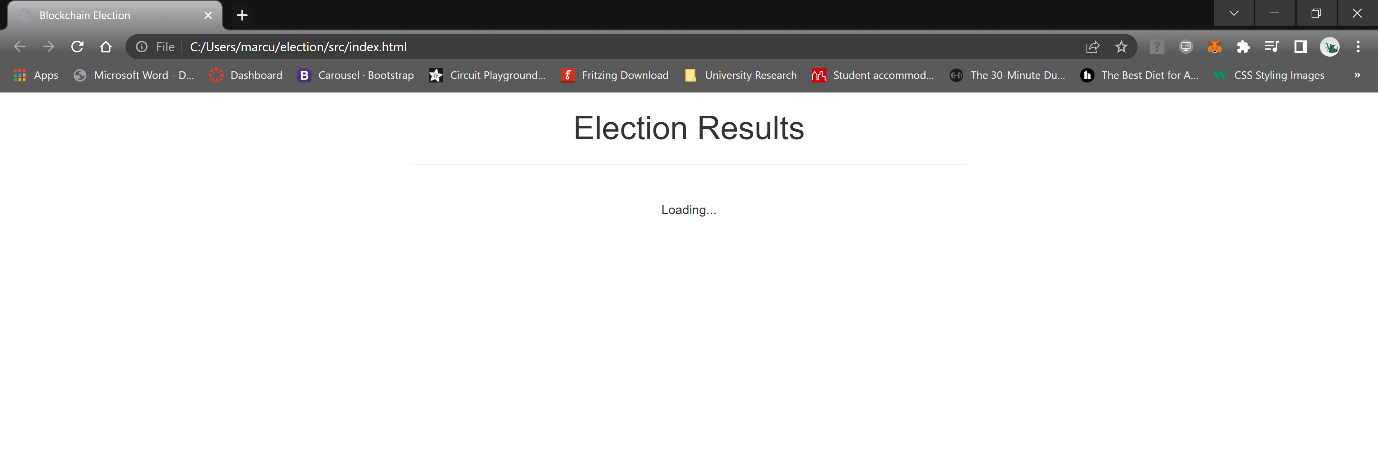
Here is a demonstration of the code working:

(SHOW GIF OF MOCHA AND CHAI TESTING)

\*Describe backend configuration sufficiently completed to be able to test some user side code\*

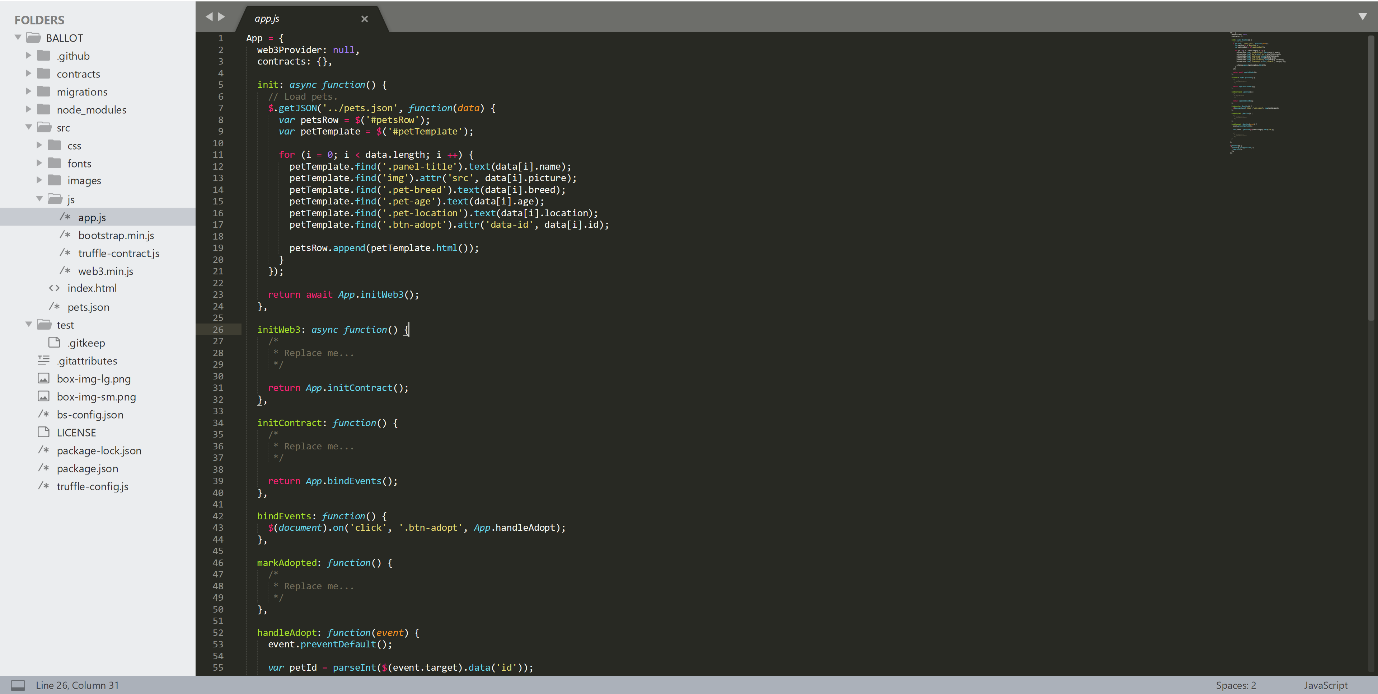
Now we move onto html and client side.





This is the site after the html code, a basic skeleton. The application requires a loading feature as it is an asynchronous application, functions operate one after the other rather than all at the same time, therefore it may take longer to configure than synchronous alternatives. The reason we use asynchronous code is to enable external operations outside of the main thread of code. There are many moving parts, from ganache, node.js, truffle, to metamask, therefore it is a necessity.

JavaScript was required to communicate the data from the smart contracts onto the front-end user interface. It is important to initialise the app whenever the window loads, to prevent errors in code configuration, and to ensure speed of operation for user satisfaction. To begin we initialise our application by creating the App value and attributes:

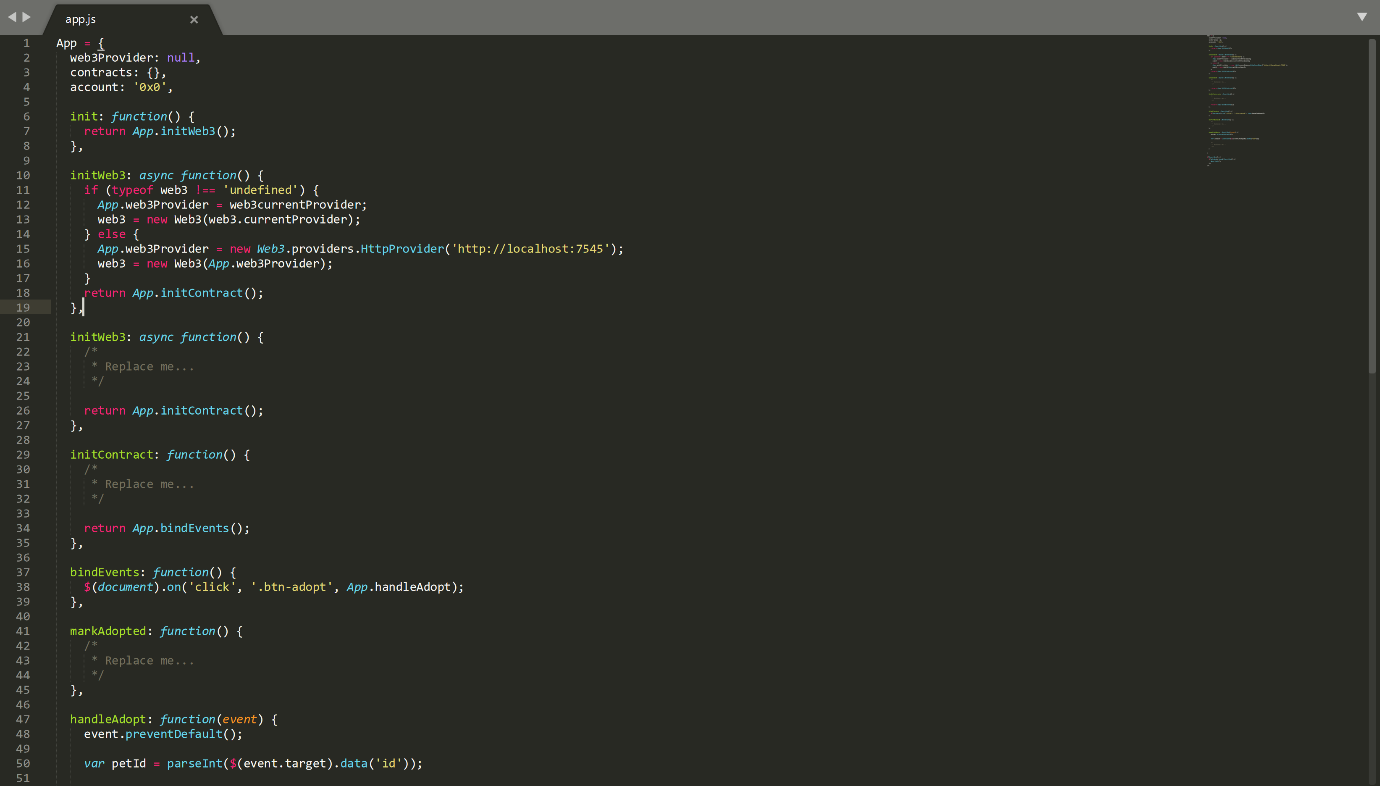


Truffle pet-shop comes with template code with App configurations and functions already made, the job now is to input my own code to make it correspond with the smart contracts and the local blockchain.

Therefore, we must initialise the connection to our web3 application, ganache, which acts as our local blockchain, the network that acts as the foundation for the application. To do this we use the Ethereum javascript API, code shown on the web3 github page:

<https://github.com/ChainSafe/web3.js>

The process of configuring the javascript code is very much a step-by-step process. First we initialise the local blockchain, then we initialise the smart contracts, then we render out the content on the webpage. This demonstrates the necessity of the asynchronous nature of the application, because each step requires the previous to be completed to function correctly.



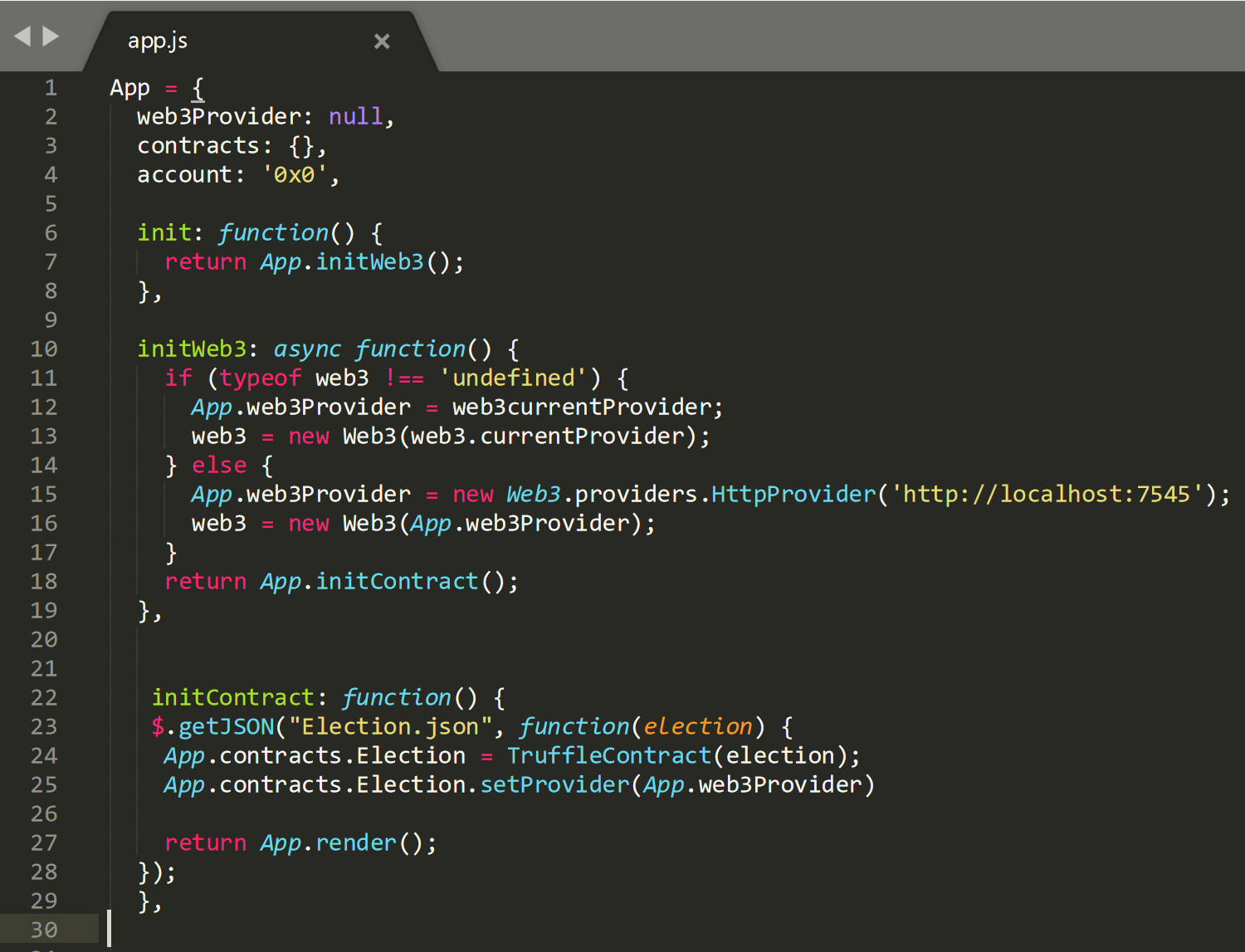
This code shows the initialisation of web3 for our application. We use metamask as our current web3 provider, to convert our browser into a blockchain supported browser, enabling connections to our local blockchain.

We match the metamask web3 provider to our own code web3 provider through ‘App.web3Provider = web3.currentProvider’.

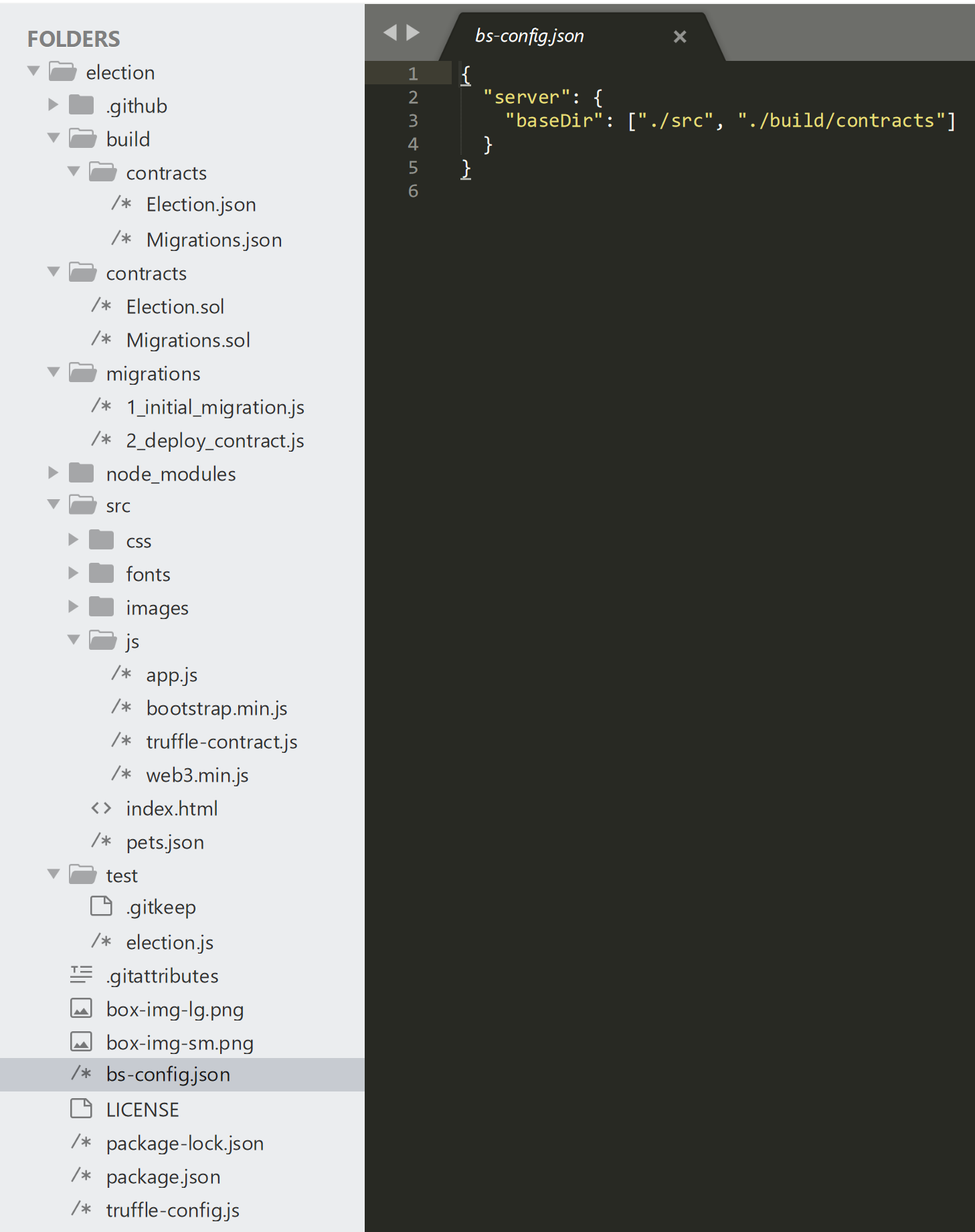
If metamask cannot provide the Ethereum blockchain, we will then use our own blockchain through ganache. This acts as a failsafe. Currently we are testing the app on our own independent node blockchain, using my own computer as both the overall blockchain network and the holder of all the accounts. Metamask will split the weight and act as the blockchain network, and then ganache would just have the responsibility of upholding accounts rather than both the network and the accounts. If metamask cannot provide the web3 provider, the localhost node will uphold the blockchain network and the accounts through ganache, as demonstrated through the else function.

The next step as stated previously is to initialise our smart contracts.

We begin by loading up our smart contract into our front end so that the user can interact with it. Because we are operating our contract on the web, it is necessary to initialise an Election.json file to transmit the data to the front-end web application.



The Election.json file is generated when the smart contracts are compiled and migrated within the command prompt. This is done through the browser-sync package that comes with the truffle pet-shop template called ‘bs-config.json’. To push our data to the client side we configure a new truffle contract which acts as a container for the code, compiling the created ‘artifact’. Each time the code is run it generates a new truffle contract.



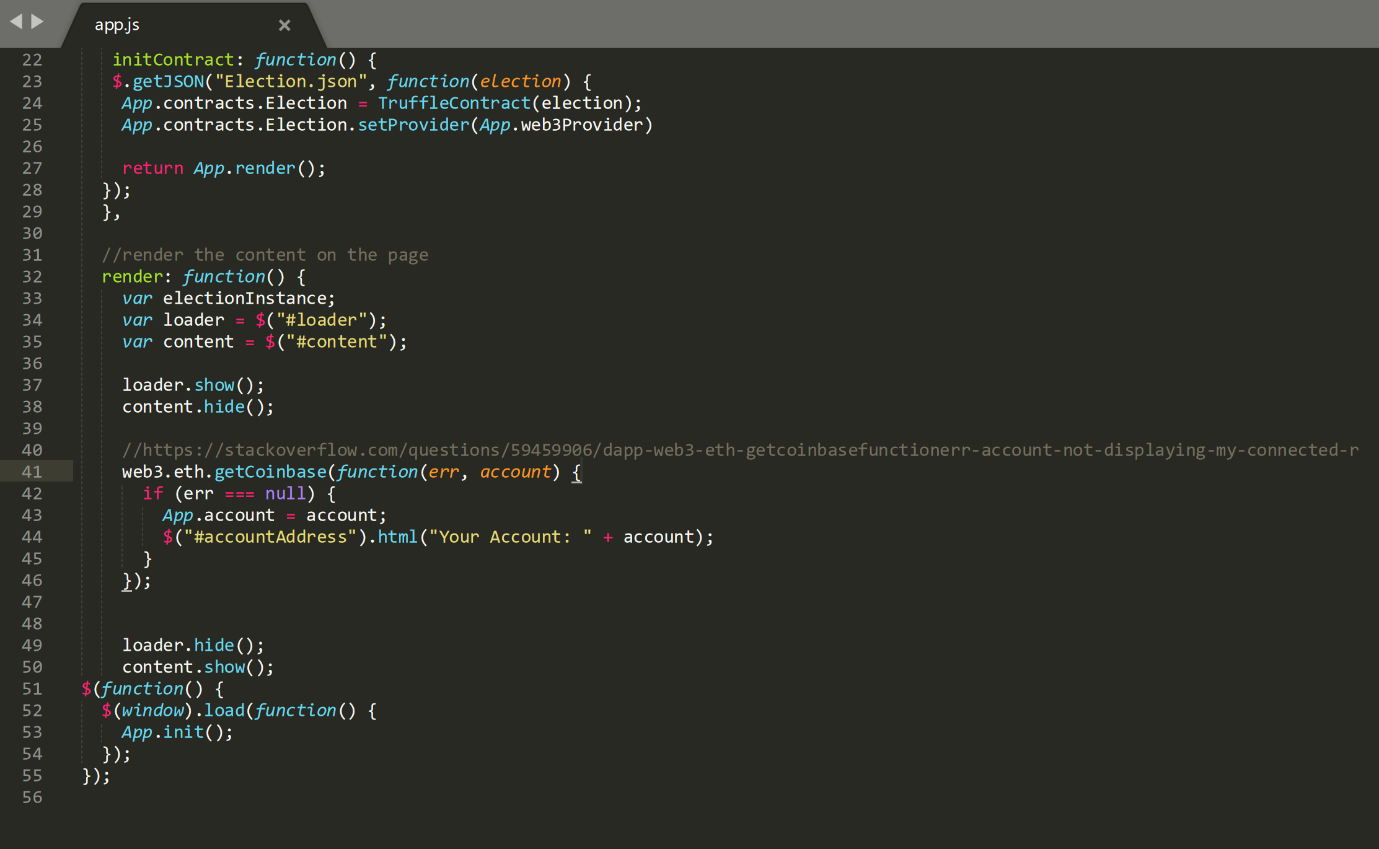
The bs-config.json file generates Election.json for you.

(INSERT JSON GIF OR MP4)

The next step is to render the information on the webpage. We do this by using ‘render’ function in JavaScript to show the output on the browser. There are two main elements of content that displays: loading screen and the content. We load these two elements asynchronously, therefore we load the accounts first, then using the loaded accounts data we can then load the contract data. Whilst we load the data in the background, we need to ensure the user knows the content is loading, therefore we input a loader and content ‘show’ and ‘hide’ function, available with JavaScript. We hide the content section whilst the data is loading with the loading screen. When the content data has been successfully initialised, we input the code at the end to switch around the show and hide functions, to hide the loading screen and show the content.

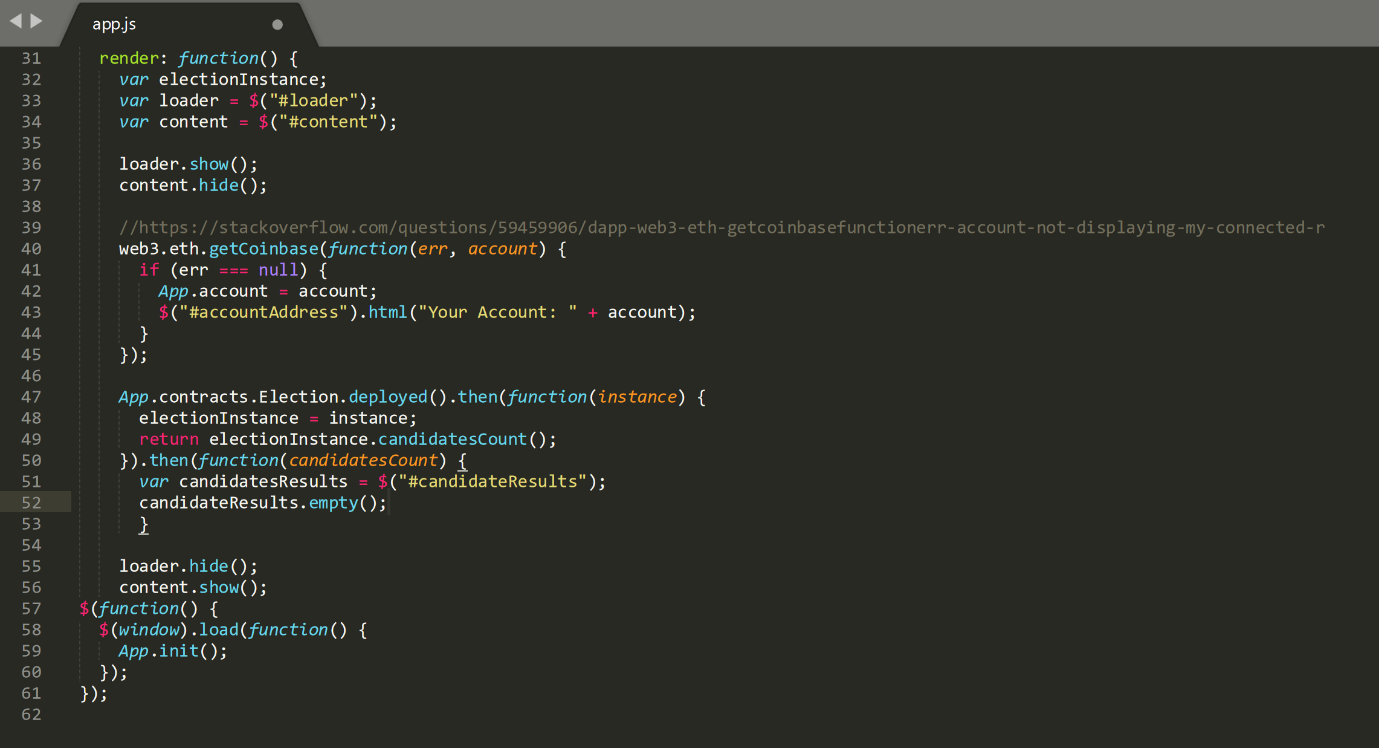
To help me show the account data I used:

<https://stackoverflow.com/questions/59459906/dapp-web3-eth-getcoinbasefunctionerr-account-not-displaying-my-connected-r>



Now that the accounts are rendered, we render the contract data. The contract section lists the candidates on the page. We use the ‘electionInstance’ variable universally among all the JavaScript files to enable them to communicate and configure data.

Firstly, we retrieve a copy of the deployed contract through accessing the generated artifact, designated as ‘App’, then into the contracts folder to retrieve the data from the Election.json file, compiled together using the bs-config.json file. Election.deployed() to display the deployed contract. We ‘then’ tell the code to initiate the instance variable programmed using the code below it



One of the most important variables is candidates count, as it is crucial it displays the exact number of candidates. Within the Election.sol file we used the ‘mapping’ function to link state variables that store the data. The nature of the mapping function means it cannot be iterated by itself, therefore the data once shown on screen will not change unless you restart the application. To overcome this we implement a ‘for loop’ to keep the application reiterating changes to the candidatesCount for example.



The for loop begins at 1, up to the number of candidates, shown through candidatesCount. For our application there is only two candidates, however if there were more the code would create a candidate for each value in that range, meaning if there are 100 candidates it would create 100.

electionInstance.candidates(i).then(function(candidate)

We call in the candidates function from the contract. We pass in the i value into the contract which will correspond to the specific candidate ID. This function is used to read the data from the mapping function, then that specific candidate is injected into the callback function ‘candidate’ and the code below scans the candidate data from the callback ‘candidate’ and displays the id, name and votecount, using the index from the election.js file.

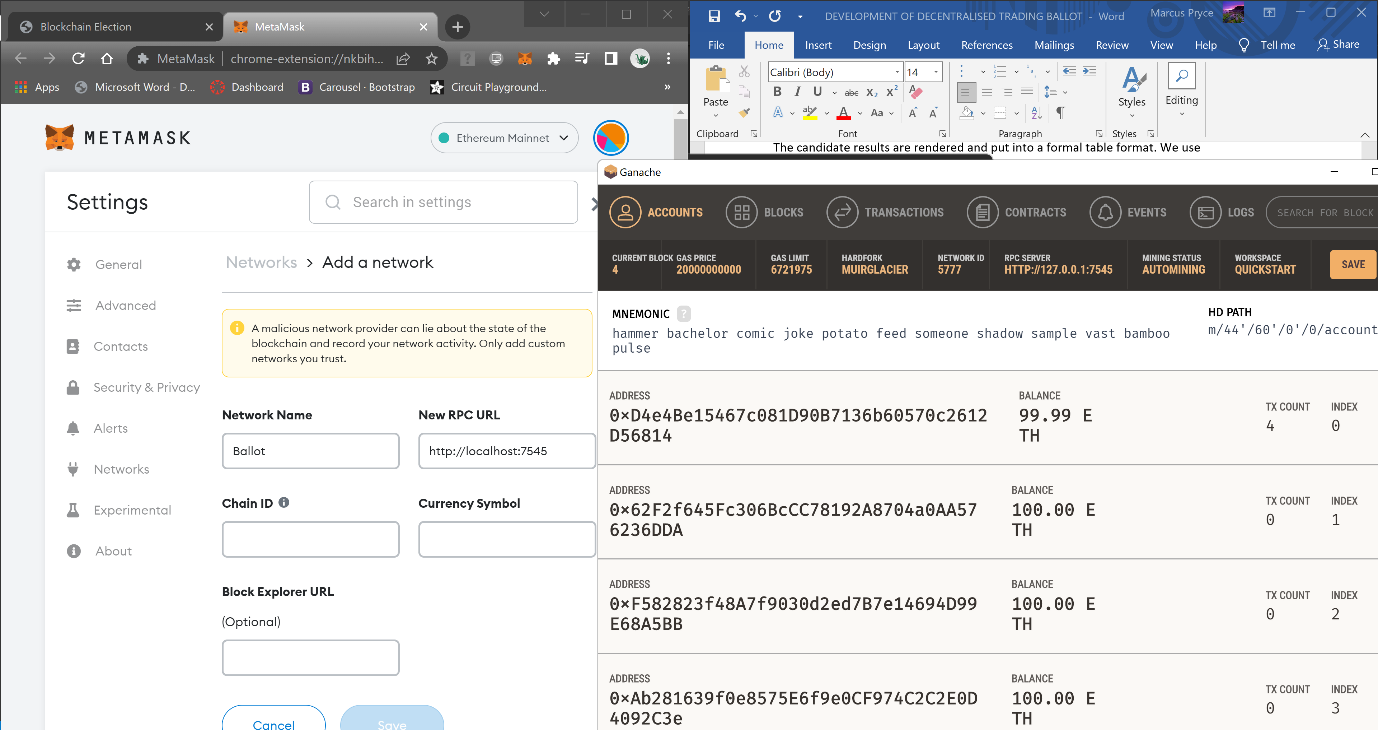
The candidate results are rendered and put into a formal table format. We use the append javascript function to display the HTML content within the candidateTemplate variable.

LOADING CLIENT SIDE APP

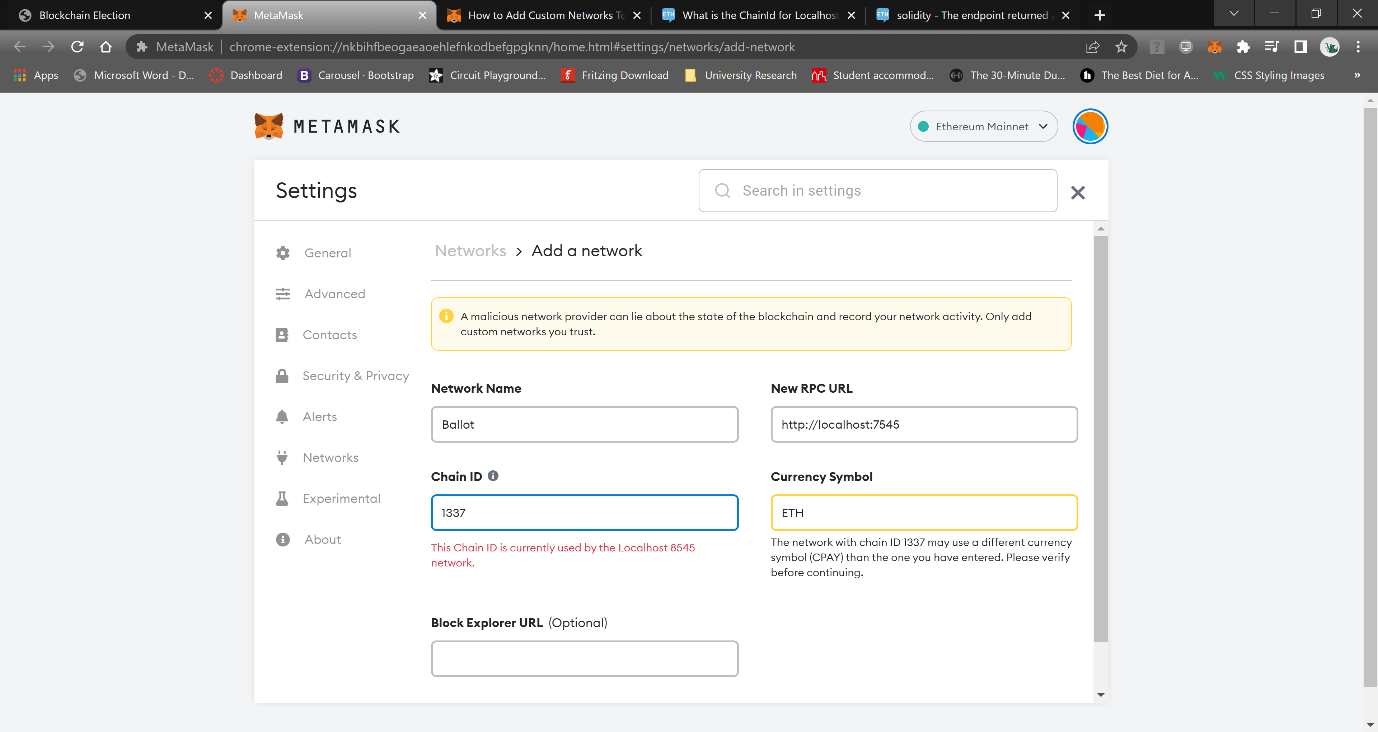
Another feature of the pet-shop truffle package is an included lite server, which serves as a development server for web applications and opens it in the browser.

(INSERT MP4 TEST CLIENT-SIDE1)

Doesn’t work because we haven’t enabled metamask to communicate our browser to our local blockchain.

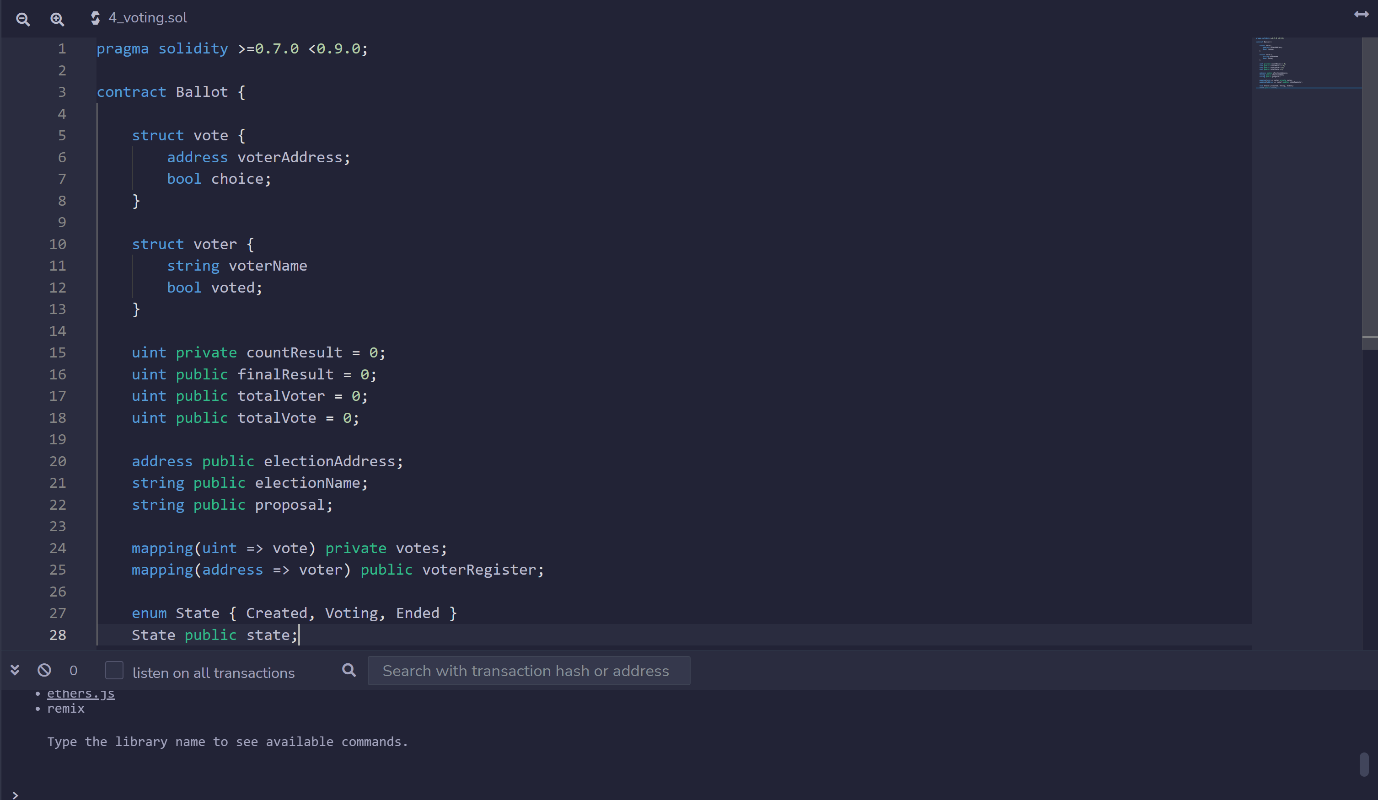


Within the RPC URL place the RPC server node ID which is 7545 at the end of our localhost.



I encountered a chainID error when linking the metamask service with my local ganache blockchain. The chain ID did not correspond to the RPC URL, instead it corresponded with a different port, 8545. I had an alternative network operating on the chosen chainID from a previous project, however it still did not work.

NEW BALLOT SIMPLER



Due to the calamity of the error regarding metamask, as well as time constraints, I have been forced to create a simpler version of a blockchain voting ballot on the remix browser-based IDE. I concluded the JavaScript compatibility with metamask is outdated and the code I implemented was not viable. Therefore I have compiled a number of state variables to begin my project within the Ethereum supported IDE.