Phong Tran

# CREATE A SECURE VOTING SYSTEM USING BLOCKCHAIN

Bachelor's thesis

Bachelor of Engineering

Information Technology

2021

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland University of Applied Sciences

| Author (authors) | Degree title | Time |
|---|---|---|
| Phong Tran | Bachelor of Engineering | January 2021 |
| **Thesis title** | | |
| Create a secure voting system using blockchain technology | | 70 pages<br>15 pages of appendices |
| **Commissioned by** | | |
| **Supervisor** | | |
| Reijo Vuohelainen | | |

**Abstract**

The main goal of this thesis was to study and implement a voting system using blockchain technology, specifically Ethereum Smart Contract. Such a system is necessary because the voting results can easily be manipulated if the data is not stored in a secured database. The use of blockchain ensures that the data is immutable, thus providing a transparent result after the voting is over.

The theoretical part of this thesis familiarizes the readers with the definition of blockchain, how the data is stored inside the ledger, how the data is verified using blockchain, and how it is impossible for a malicious user to tamper with data inside the network. Moreover, the theoretical part explains the Ethereum ecosystem and how to interact with Ethereum Smart Contracts.

The practical part provides the specification needed to build this application, along with instructions as to how the application is created. The application was created using ReactJS as a Front-end Framework, NodeJS to build the Back-end server, and Solidity language to build the Ethereum Smart Contract.

After the practical part, the outcomes and the results of the application are discussed in the result and the discussion section. Possible considerations and improvements are also discussed, along with one downside of the application. Finally, the conclusion of this thesis discusses how the developed code solves the original tasks of secure voting. One alternative method is also mentioned in this section and why the alternative may not be the better option. Lastly, further improvement, updates, and thoughts are also mentioned in the conclusion as well.

**Keywords**

blockchain, ethereum, smart contract, reactjs, nodejs

**CONTENTS**

# 1 INTRODUCTION

Voting is a symbol of democracy. The birth of this concept suggested that an empire, a country, or a government needed to hear its citizens. As idealistic as it sounds, no ideas are perfect, and there are a lot of methods to manipulate the voting result. Considering the US Presidential Election in 2016, there is still a rumor suggesting that the election result was tampered with, which resulted in the victory of Donald Trump over Hillary Clinton (CNN, 2016). Although this story can either be true or false, the fact that there was no proof supporting the trustworthiness of the election means that the transparency of the election remains a question.

One solution to solve this problem is the use of blockchain. According to IBM, Blockchain is a shared, immutable ledger that facilitates the process of recording transactions and tracking assets in a business network. Blockchain ensures three important aspects: decentralization, immutability, and transparency. With its attribute, the use of blockchain is perfect for creating a voting system because the result cannot be tampered with if it is saved inside a blockchain.

In the following sections, an in-depth explanation about how blockchain operates will be provided along with other relevant technologies. There will also be an implementation part where a decentralized application will be presented, and there will also be a step-by-step instruction on how to build it. There will also be a discussion section where the result and the feasibility of the application will be discussed. The reason why blockchain and its characteristics are perfect for solving the problem of manipulating voting result is that there is no way to freely mutate the result inside the blockchain without complying with certain rules defined inside the blockchain application.

# 2 BLOCKCHAIN

Blockchain technology has been around since 2009, along with the introduction of the famous cryptocurrency, Bitcoin. Blockchain is a distributed network that uses cryptographic as its fundamental protection. It is essentially a ‘chain of

blocks' where each block stores data or the transactions that have been made and the entire network is shared among the participants using a peer-to-peer network.

Each block inside the blockchain contains verified hashed transactional data and other attributes that make up the entire block which will be explained in later chapters. When a transaction is created, the transaction will be signed by the person who made the transaction, then propagated to the network. Each node inside the network will get a copy of that transaction, then the nodes, which are also called miners, will add the transaction into their own 'block'. After that, they have to solve a very complicated hash algorithm which is called "Proof of Work" in order to find the right hash output. After the miner found the correct hash solution, his 'block' will be propagated to other nodes to ensure the validity of the block. Once the block is verified, it will be included in the blockchain network.

## 2.1 Hash Algorithms

Above was a short explanation as to how blockchain operates, but when talking about hashing, which hashing algorithms does the blockchain utilize? In Bitcoin, when a user first registers an account, a private key is assigned to the user. The private key is a random 256-bit number between 1 and n -1, where n is a constant (n = 1.158 * 10^77, slightly less than 2^256) (Antonopoulos 2015, p.60 – 63). After the number is picked, a public key will be generated using a one-way hash algorithm called Elliptic Curve Algorithm. After the public key is generated, the address of the user will be calculated using SHA256 and RIPEMD160 algorithms to produce a 160-bit number. The formula will be:

*A = RIPEMD160(SHA256(PK))*

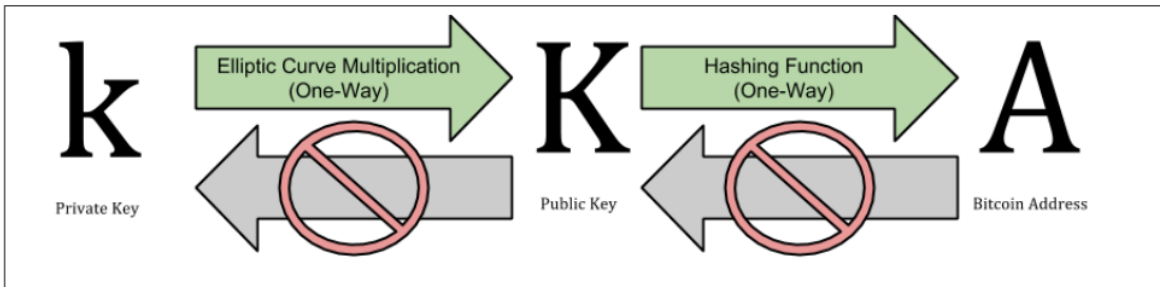



Figure 1 *The relation between the keys (Antonopoulos 2015, p.63)*

The public key (PK) is hashed to a 160-bit number is because the public key has an extremely large memory (256-bit) which would create an overflow if the public key is being used as an address. The creation process is shown briefly in Figure 1.

The reason why the Elliptic Curve Algorithm was used to generate the public key is because the Elliptic Curve Algorithm is a one-way hash algorithm which is not only impossible to reverse, but there are also no two messages that will produce the same result. The way Elliptic Curve Algorithm works is by picking a point on a graph called *k*. After that, a tangent line will be drawn from point *k* and intersected at a point on the graph. From that intersected point, the inverse point will be taken into use and from that point, the process will repeat for a number of *G* times. The final point will be the inverse point of the last intersected point of the last line on the graph. The formula for this algorithm is:

$K = k*G$

where K is the resulted public key, k is the private key, and G is the number of times. The illustration of this algorithm is illustrated in Figure 2.

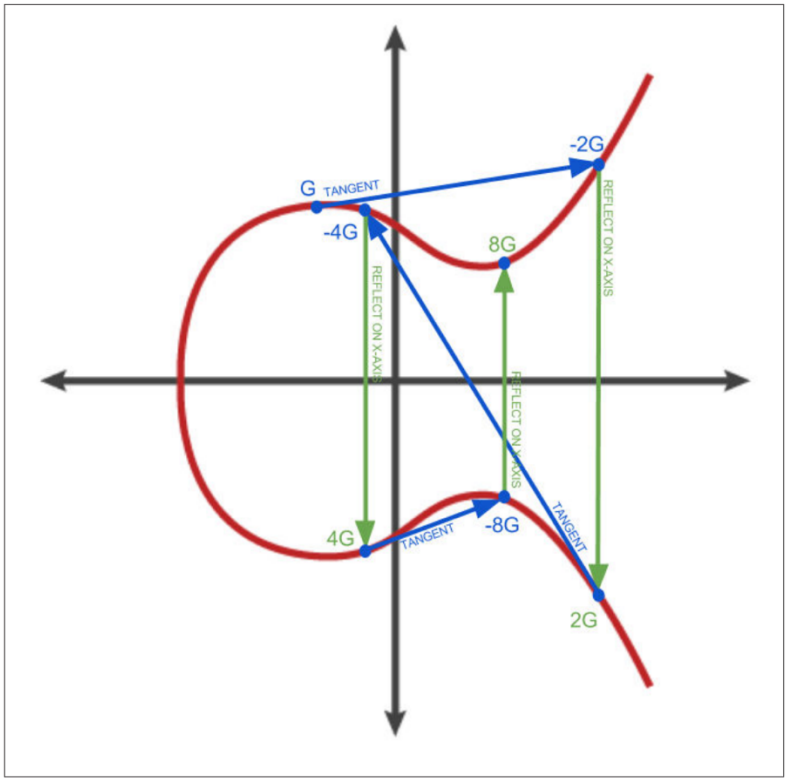



Figure 2 Elliptic Curve Algorithm (Antonopoulos 2015, p.70)

The reason why this algorithm is irreversible is because if the resulted point was given, it would be impossible to backtrack to the starting point.

## 2.2 Block

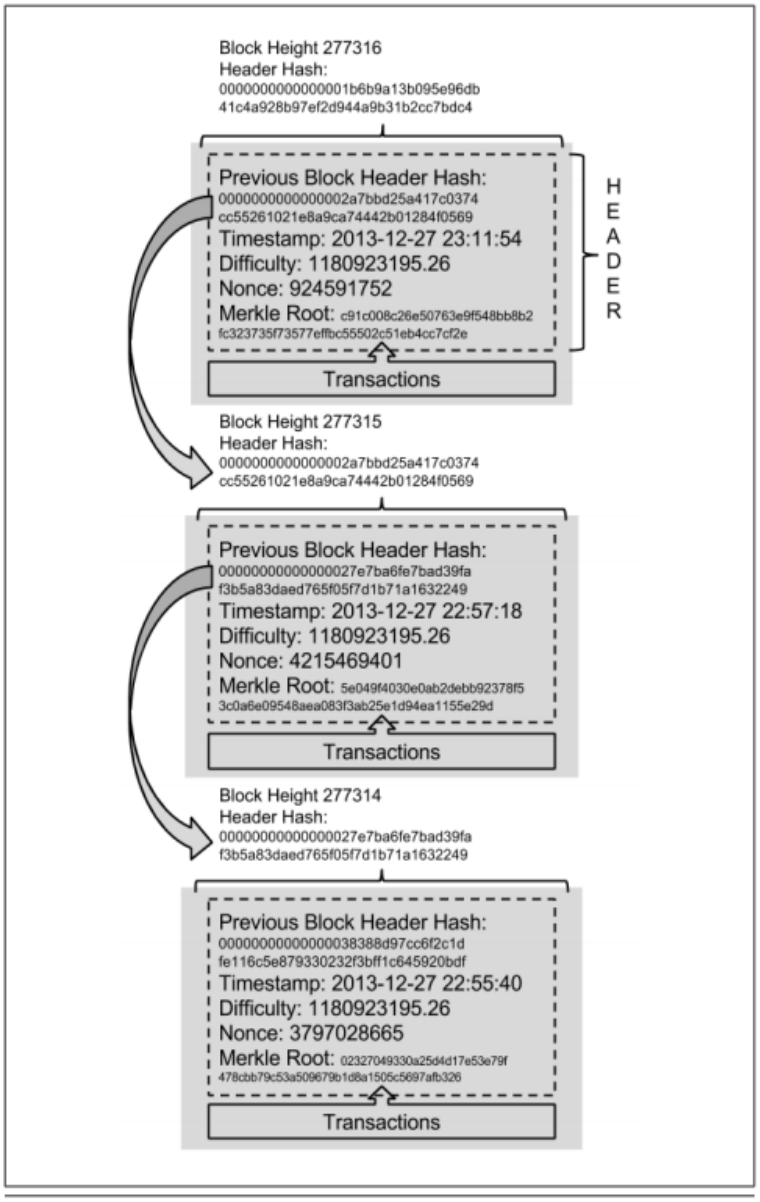



Figure 3 Blocks linked in chain (Antonopoulos 2015, p.169)

As mentioned above, blockchain is essentially a chain of “blocks”. When blockchain is deployed for the first time, the first block of the blockchain is called a “Genesis block”, which contains no transaction data. After that, other subsequent blocks will link to their previous block by the attribute “previous block hash”. Aside from that attribute, a block can contain other attributes such as:

merkle root, difficulty, nonce, and a list of transaction. The reason why blockchain is immutable is that each data in the block has been hashed by SHA256, and when data is mutated by just a little bit, the produced hash result will be completely different. Even if the hacker managed to find the right hash result to mutate that entire block by changing every single data, the hash result will be completely different than the following block's previous block hash. There is also a rule in blockchain called the "51% rule such that if 51% of the participants in the network agrees that a block is validated and trustworthy, then the block is essentially valid and will be deployed onto the ledger. The 51% rule also applied to the blockchain ledger such that if at least 51% of the computers in the network verify that the blockchain on the ledger has not been tampered with, then the blockchain is still valid. However, with the evolution of computer power, the 51% rule for computer power can be easily overruled. Hence the birth of "Proof of Stake"; instead of using computer power to mine the block and find the correct hash result, the validators in this case will use their tokens to "stake" and validate the transactions. More on the concept of "Proof of Stake" in the later chapter.

### 2.3 Merkle Tree

Another attribute that is often overlooked when talking about blockchain is Merkle Tree. According to SelfKey (2019), Merkle Tree is a way of structuring data that allows a large body of information to be verified for accuracy both extremely efficiently and quickly. Each block inside the blockchain contains a large amount of transaction ID, and to iterate over the total number of transactions to verify whether or not the transaction is valid takes an enormous amount of computer power. In order to mitigate the use of computer power in order to look for the transaction ID in the blockchain, a binary tree-based search method was introduced: the Merkle Tree. Merkle Tree helps reduce the amount of computing power needed in order to find the transaction ID inside the blockchain.

Merkle Trees are created by hashing each pair of hashed transaction id until it becomes one hash called Merkle Root as shown in the figure 4.

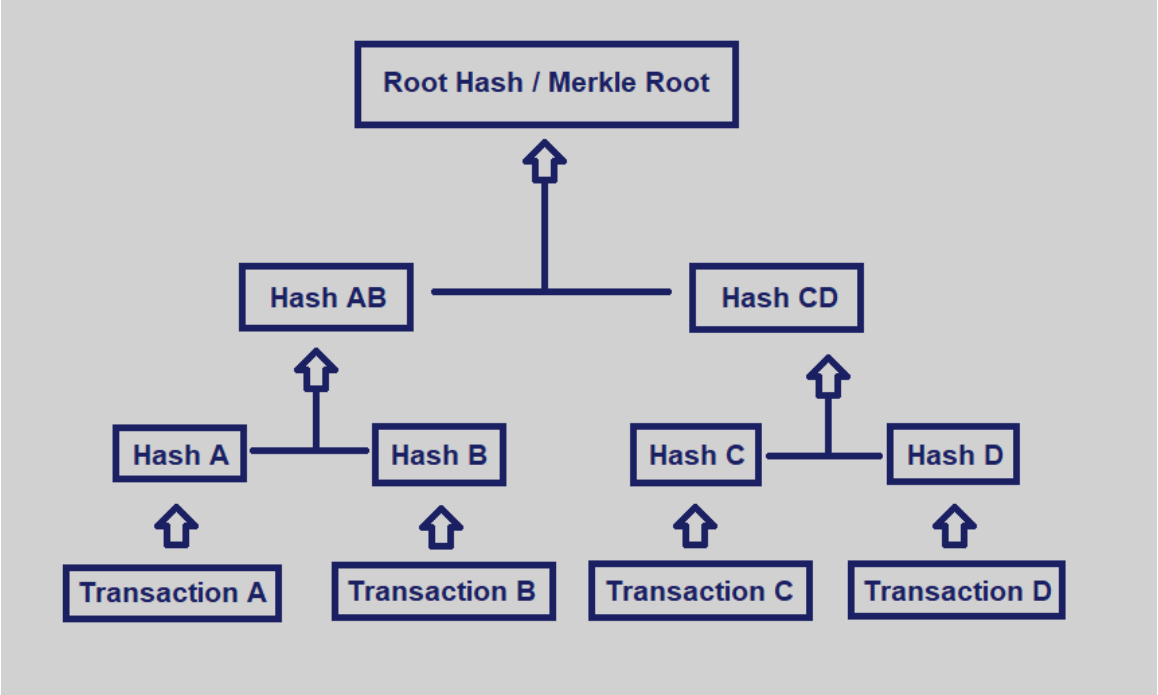



Figure 4 *Merkle Tree (Jain 2018)*

For example, if a user needs to verify the validity of Transaction B, the user will need to know Hash A, and Hash CD. Hash B, and Hash AB can be computed with the needed information. If the result of hashing Hash AB and Hash CD is equal to Merkle Root, it means that Transaction B is entirely valid. Merkle Tree is used in various blockchain networks, including Bitcoins and Ethereum.

## 2.4 Mining

The term most familiar to anyone when talking about blockchain is mining. Mining is the process where miners need to perform very complicated computational calculations to be able to find the correct hash in that block mining process. This is also the concept of Proof-of-Work. Essentially, Proof-of-Work is where there is a specified amount of difficulty such that a miner needs to find the nonce number where the resulting hash of every attributes inside the block and the nonce has to have greater than or equal *k* number of leading zeros, and *k* = *difficulty.* For example, if the difficulty is equal to 5, then the resulting hash must be 00000asjdhkeaah23132432dj… The difficulty is periodically reset so that the average time for a block to be added onto the blockchain network is about 5 to 10 minutes.

The fastest miner who finds the correct hash will be able to include their block onto the public ledger. After a block is added to the network, a specific amount of token, i.e., Bitcoin or Eth, will be awarded to that specific miner, along with the transaction fees in that mining cycle. The purpose of this reward is to motivate other miners to join the blockchain network and help with the block adding process, thus making the blockchain network even more secure. There is also a case where multiple miners come up with the answers at the same time, and the blockchain network has a way to handle that situation. Essentially, if there are three miners coming up with the answer at the same time, the chain will split into three sequences, and each sequence needs to compete with each other to form the longest sequence. This action is also known as “forking”. It means that in the subsequent mining processes, the other miners will choose one sequence and mine on until the longest sequence is found. When that happens, the other sequences will be dropped from the blockchain. This can lead to a problem when a transaction is already verified on the blockchain and then suddenly get unverified, but most of the time those transaction will be reverified rather quickly so this is not so much of an issue in the blockchain network.

### 2.5 Validator

There is another concept used by other blockchain networks such as Ethereum 2.0 in order to add blocks onto the blockchain network, and it is called “Proof of Stake”. As explained above, Proof of Stake is also a consensus mechanism where the validators will be rewarded after pushing a block onto the blockchain. The biggest difference between Proof of Work and Proof of Stake is that instead of relying on the computer power, Proof of Stake is relying on economic power of its validator. According to Wackerow (2020), the validators in Proof of Stake do not need an enormous amount of computer power and they are not necessarily competing in order to add new blocks onto the network. Instead, the validators will need to stake their coins in order to participate in the staking process. The validators will be chosen either randomly or deterministically by the network and their duty is to add new block onto the network and other validators will be able to attest the chosen validator. If the validators attest to a malicious block or if they include an invalid transaction into the block, they will ultimately lose a part of their

stakes. The purpose of Proof of Stake is to improve energy efficiency when the participants do not need computer power to participate and improve immunity to centralization when there will be more nodes on the network. The birth of Proof of Stake also addresses the “51% attack” when there is one node with enough computer power to tamper the entire blockchain network. With Proof of Stake, every effort of tampering with the network will result in the loss of money, so it essentially reduces the temptation of trying to pollute the blockchain network.

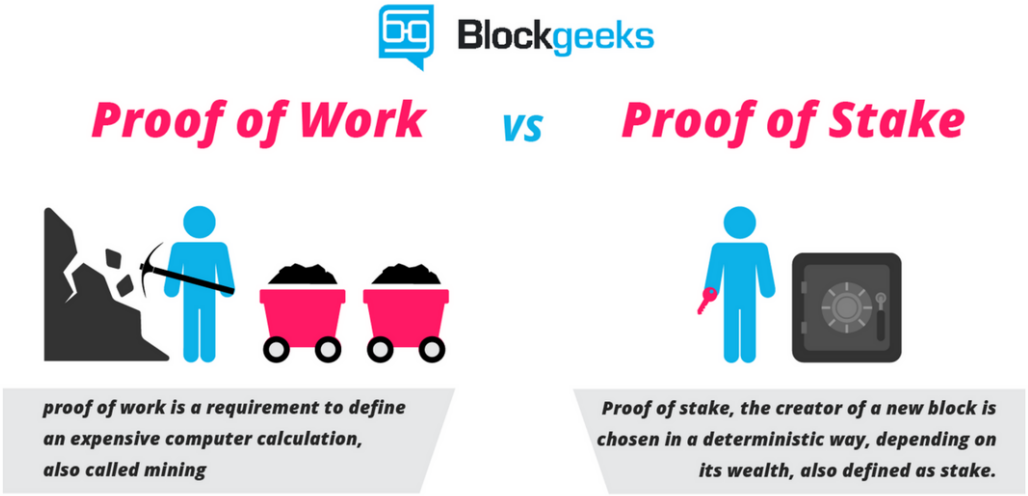



Figure 5 Proof of Work vs Proof of Stake (Use the Bitcoin 2020)

## 3 ETHEREUM

This thesis will make use of the Ethereum blockchain to create a Decentralized Application. Ethereum was found in 2013 by Vitalik Buterin, and it is often described as “the world computer”. Ethereum is an open-source, globally decentralized infrastructure that executes the program called “Smart Contract” (Lavayssière 2018). The cryptocurrency used in Ethereum blockchain is ether.

### 3.1 Smart Contract

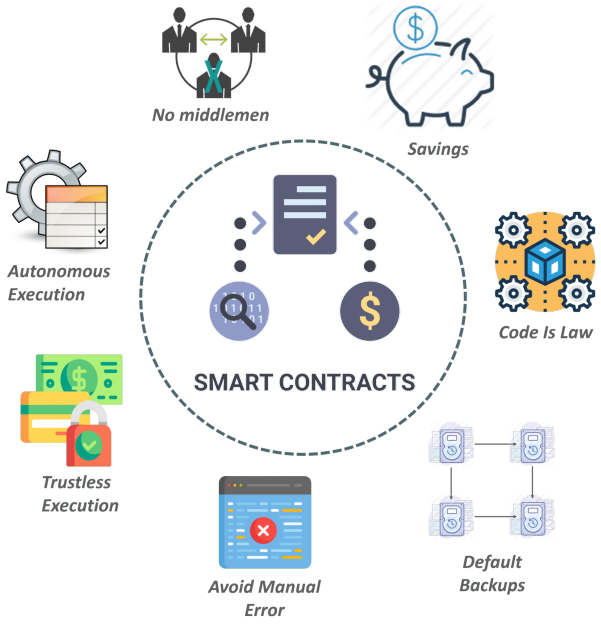



Figure 6 *Smart Contract (Shashank, 2019)*

Smart Contract is a program that runs inside the Ethereum Blockchain executed by Ethereum Virtual Machine. Smart Contract is an immutable program, meaning once the code is written and deployed to a blockchain, it cannot be updated or rewritten. Vitalik describes this concept as follows: “code is law”. The language used in Ethereum Smart Contract is Solidity language. After the code is written and ready to be deployed, the developers have the option to deploy it to Mainnet, which is the real network, and it uses real Ether. If the developers want to test their Smart Contract, they can deploy their Smart Contract to four of the Ethereum Testnets: Ropsten, Kovan, Rinkeby, and Goerli. These Testnets do not use real Ether. Instead, the developers can ask for Ether from one of these Testnets’ faucets.

A Smart Contract consists of state variables, events, modifiers, and functions. Each function call that mutates the state variables inside the Smart Contract will be a transaction, and each transaction will cost a certain amount of “gas”. The amount of gas spent will depend on the complexity and the memory of the function. Other functions such as return function or pure function do not consume

gas as long as they are not called from another mutative function and the return function does not mutate the values of the state variables.

The biggest advantage when using Smart Contract is that there is practically no downtime since the blockchain is maintained by millions of users. As long as the Ethereum blockchain network is still up, the Smart Contract will still be valid.

### 3.2 Decentralized Application

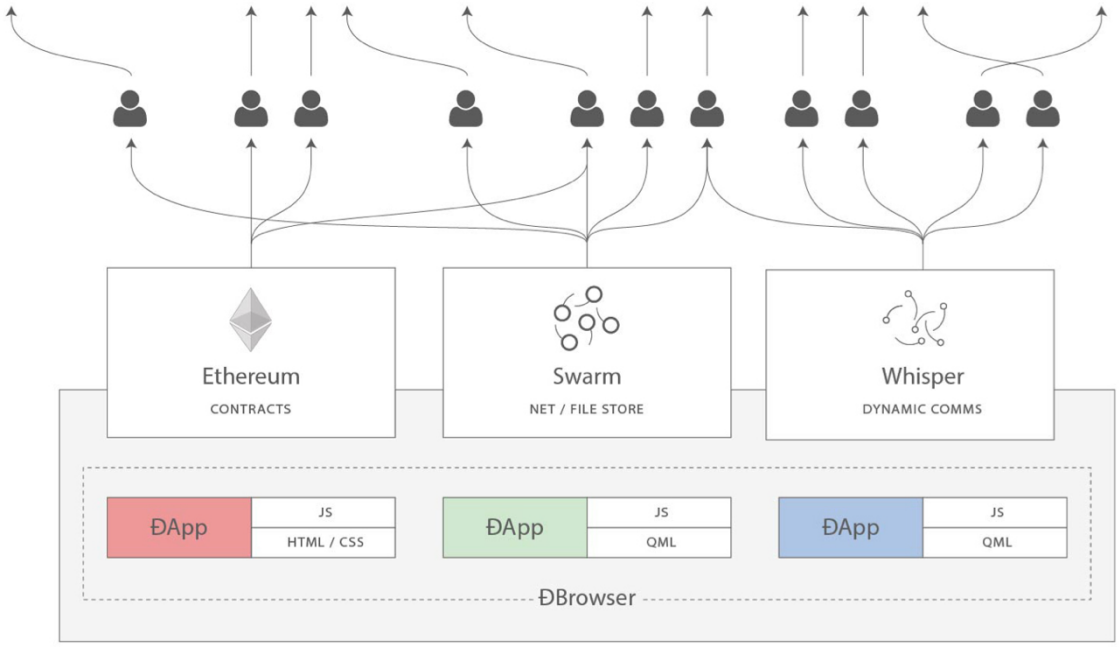



Figure 7 *Typical architecture of a decentralized application (*Lavayssière 2018)

Decentralized Application is an application that is mostly decentralized. An application mostly consists of: Frontend, Backend, and Data Storage. The Frontend will most likely be hosted on a centralized server, or a mobile app that runs on the end-users' devices. The Backend can be entirely decentralized using Smart Contract. As mentioned above, since Smart Contract is entirely stored on the blockchain, the user will experience no downtime and the service will continue to be available as long as the blockchain network still exists (Lavayssière, 2018). For Data Storage, unfortunately, Smart Contract does not have the ability to store bulky data or a complicated data structure. The best option for a developer is to make use of centralized servers, or they can also make use of decentralized servers such as InterPlanetary File System (IPFS) and Swarm. However, if the developers only wish to store simple data such as

primitive data types, mappings, or arrays, a Smart Contract will be the best data storage in this case due to its decentralized nature.

# 4 DEVELOPMENT TOOLS AND SERVICES

This section lists out the necessary tools that are used inside the application which will be implemented later inside this thesis.

## 4.1 ReactJS

ReactJS is a Frontend JavaScript framework and developed by Facebook. As of 2020, ReactJS ranked number one for the best JavaScript library to build user interface, according to Stackoverflow (Donovan 2019). It is a versatile library, and it also has a short learning curve. ReactJS has four important features, which are JSX, Component Lifecycle, Props, and States.

### 4.1.1 JSX

React provides syntax extension called JSX. It is not a string nor Hypertext Markup Language (HTML). JSX expressions are compiled into JavaScript function calls that evaluate JavaScript objects – React elements (React Documentation 2020). The example of JSX is shown below.

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

Figure 8 React element declare with JSX (upper) and without JSX (lower) (React Documentation 2020)

The first declaration uses JSX syntax, and it is internally compiled to the second declaration. It is entirely possible to write React without JSX but it is not common. It is much easier for developers to visualize the layout of the webpage by using JSX syntax.

### 4.1.2 Component Lifecycle Methods

Component lifecycle methods are also essential parts of the React ecosystem. There are different lifecycle methods that are triggered during different stages of the component, either when the component is being mounted onto the Document Object Model (DOM), or unmounted from the DOM. There are 5 most common lifecycle methods in React: render, constructor, componentDidMount, componentDidUpdate, and componentWillUnmount.

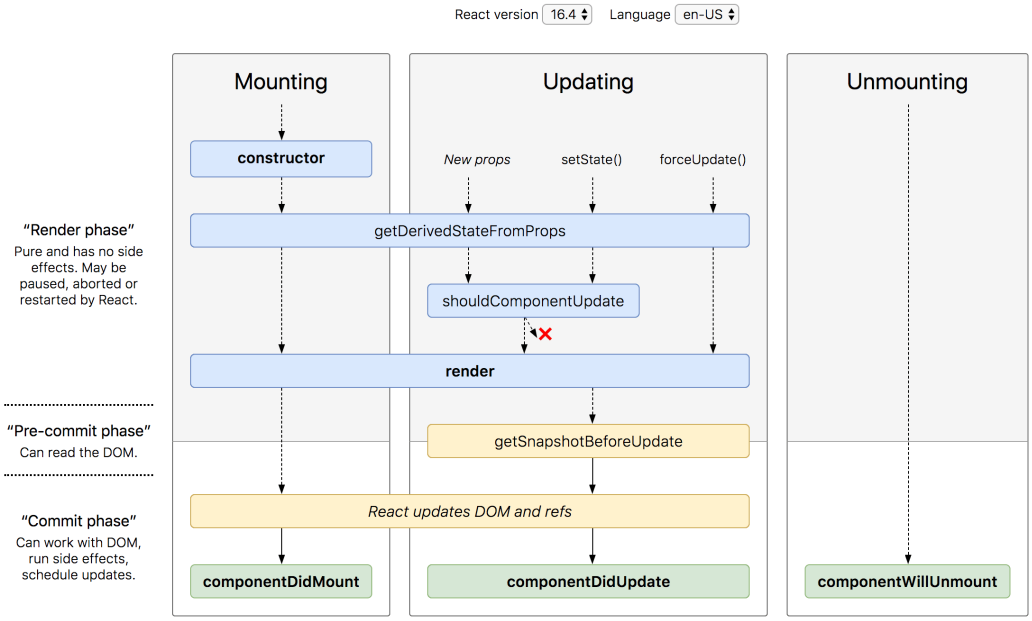



Figure 9 React component lifecycle (React Documentation 2020)

The lifecycle methods can be explained as the following:

- render(): the render method is required in a React class based component. The method returns JSX and renders the component onto the DOM.
- constructor(): the constructor method only runs once and it is used to initialize state variables and to bind event methods to the instance of the component. The method is called right before the component is mounted to the DOM.
- componentDidMount(): the componentDidMount method triggers right after the component is mounted onto the DOM. The method is usually used to perform API calls or adding subscription.
- componentDidUpdate(): the componentDidUpdate method triggers every time the states or the props of the component are changed. The method

comes with previous state and previous props as arguments, and it is possible to compare them with the current state or props to perform certain logics.

- componentWillUnmount(): the componentWillUnmount method triggers right before the component is unmounted from the DOM. The method is mostly used for clean-up and unsubscribing from any subscription.

Aside from lifecycle events, the following concepts are also essential in the React ecosystem.

### 4.1.3 Props

Props is one of the most important aspects of the ReactJS ecosystem. React's architecture is similar to Tree structure. In a React application, there will always be an outer most component that wraps the entire application. Most API calls and subscriptions will be triggered from that component. If there is a data that its child components need to make use of, that data can be passed from the parent component as props.

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

Figure 10 Props example (React Documentation, 2020)

The above example makes use of props in App component. In this case, App is the parent component and Welcome is the child component. Welcome is making use of "props.name", and App passes names to each Welcome component in App.

### 4.1.4 States

Along with props, the concept of states is also an important aspect of React. Each component will have its own states that can be passed to other components as props. The reason why states are so important is that they make the webpage dynamic by re-rendering the components every time their states are being updated.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Figure 11 State example (React Documentation, 2020)

There are some caveats when using states in React that most beginner developers do not know about. Those caveats are:

- Do not mutate the state directly, meaning they cannot just write *this.state.date = new Date().* Instead, they have to write *this.setState({ date: new Date() }).*
- State update maybe asynchronous, it is not safe to use the value of state directly like:
  “this.setState({ counter: this.state.counter + this.props.increment });”.
  Instead, setState also accepts a function rather than an object as an argument, and the function will receive the previous states and previous props as argument. The second form of setState can be written like this:
  “this.setState((state, props) => ({ counter: state.counter + props.increment }));”

State is one of the most useful but also the most confusing aspect in the React ecosystem, and a lot of beginner developers are struggling with the concept of states.

## 4.2 Web3

Web3 is a JavaScript library that includes functionality for the Ethereum ecosystem (Web3 Documentation, 2020). This is the best library to interact with Ethereum Smart Contracts inside a JavaScript application. The library includes various methods establishing a connection to Ethereum's Mainnet and Testnets, creating a transaction, signing a transaction, sending a transaction to the network, and listening to any Smart Contract events.

## 4.3 NodeJS

NodeJS is an open-source, cross-platform, JavaScript runtime environment. It uses Chrome's V8 engine and executes JavaScript codes outside of browser environment (NodeJS Documentation, 2020). It has built-in library to create web servers, and it also comes with node package manager (npm). It allows developers to install external libraries and import those libraries to their application. One of the most popular npm libraries is "express", which is a NodeJS framework, and it allows a developer to write Backend API with ease and automate a lot of details for the developers.

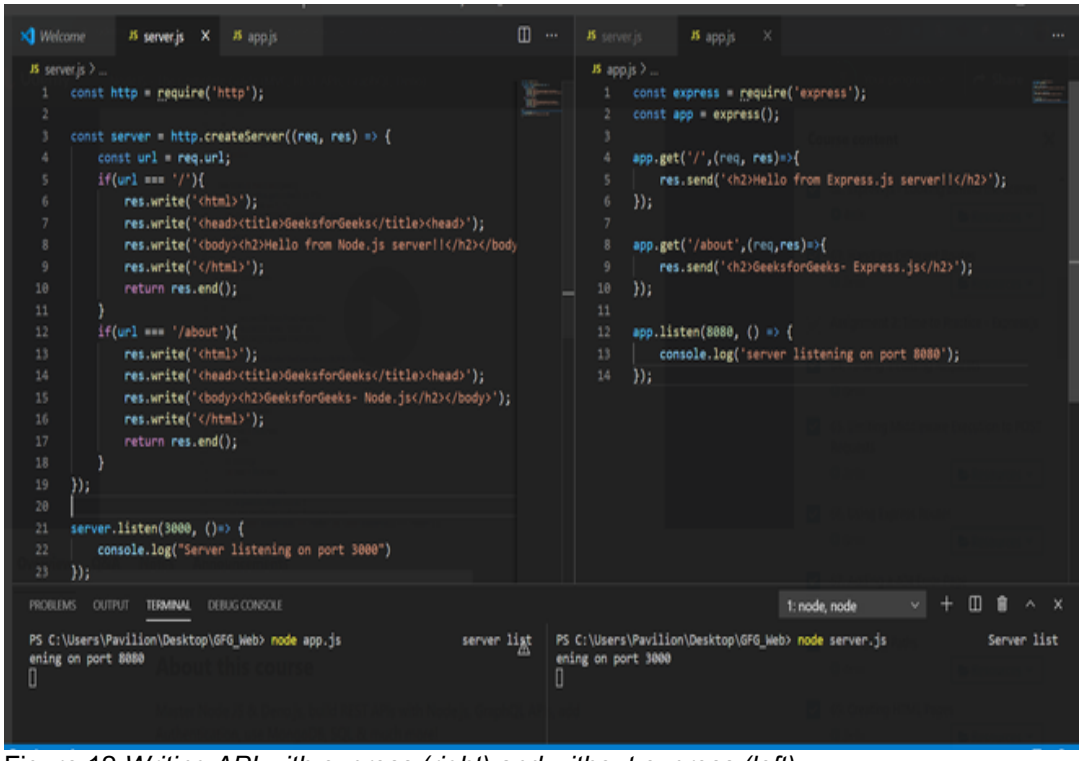



Figure 12 *Writing API with express (right) and without express (left)*

The figure above shows that express abstracts a lot of excess code that the developer must write when not using express. The only method that the user has to write is “res.send()” and the content will be sent to the web browser. Moreover, the developer can create routes for POST, PUT, and DELETE with ease by typing “app.post()” or “app.put()”, whereas without express, it is complicated to set up the route handler for each type of request because the developers will have to write many if-statements to handle those requests.

### 4.4 PostgreSQL

PostgreSQL is one of the most popular relational databases and it is going to be used in this application. The database is being interfaced by Structured Query Language (SQL) and it is a versatile database to retrieve data from and create relationships between each table inside the database. The database supports various data types including text, number, Boolean, JSON, and Dates object.

## 5 IMPLEMENTATION

In order to implement this application, the following tools are required:

- Node version 12.6.3
- Npm version 6.14.5
- Truffle version 5.1.27
- Yarn version 1.22.4
- PostgreSQL 12.3

The architecture of the application is shown in figure 13.

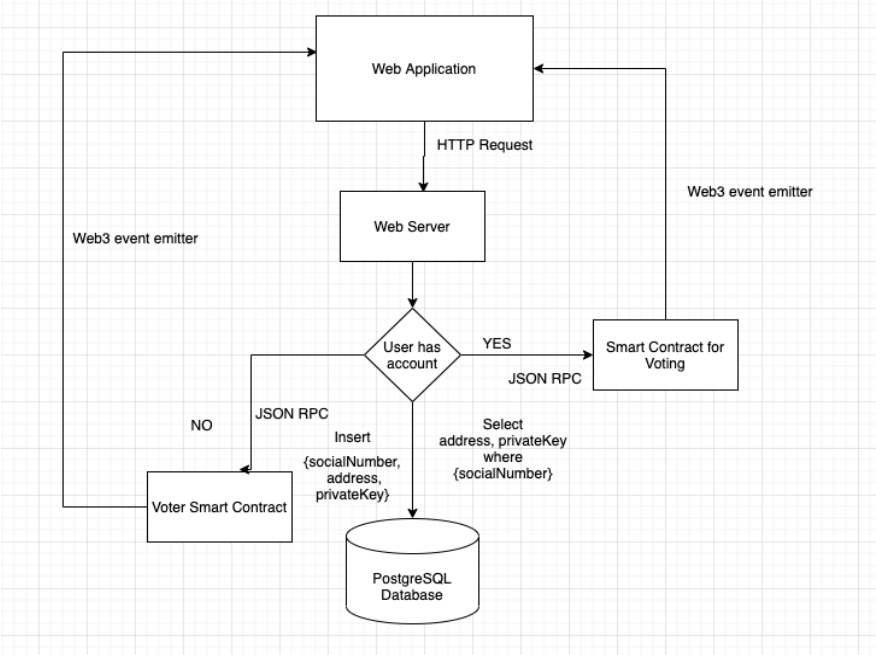



Figure 13 Project architecture

There will be a web application for the users to interface with. The users will first need to register an account in order to vote using their Social Security Number. After they have registered an account, they can select which petition they want to vote in. All projects or services inside this thesis will be hosted inside a common root project directory.

## 5.1 Voter Smart Contract

Voter Smart Contract will be the first thing to be implemented. Open the terminal/console and type in the following:

```
mkdir VoterSmartContract
truffle init
npm init -y
npm install @openzepplelin/contracts @truffle/hdwallet-
provider truffle-hdwallet-provider
```

The first command creates a new directory called “VoterSmartContract”. The second command is to initiate and scaffold a new Smart Contract project using truffle. After that, the third command is to initiate a node project within the directory using npm. The purpose of this command is mainly for creating a new package.json file. Lastly, the final command is used to install necessary

dependencies to develop and deploy the Smart Contract. After successfully running the commands, the project structure should look like figure 14

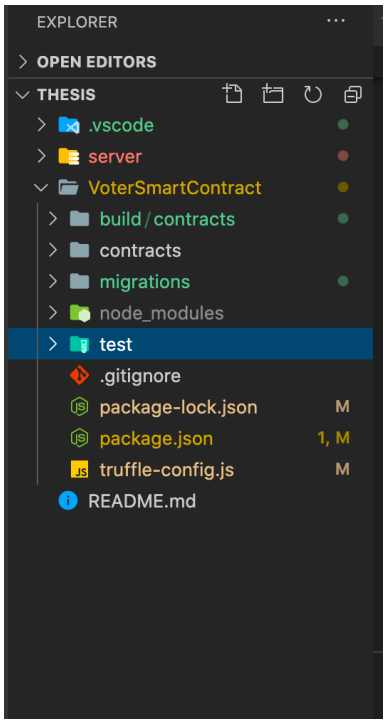



Figure 14 Project structure after running "truffle init"

Inside the contract repository, create a new file called Voter.sol. Then, type code 1 onto the newly created file:

```
pragma solidity >=0.4.21;
import "@openzeppelin/contracts/ownership/Ownable.sol";
```

Code 1

Code 1 snippet specifies the solidity version and importing the Ownable contract from *openzeppelin* library that was installed earlier. After typing in the following code, initiate the contract by typing code 2.

```
contract Voter is Ownable {

}
```

Code 2

Code 2 is similar to declaring a class in other Object-Oriented Programming (OOP) languages. The type “contract” is essentially a “class”, and “is” is the same as “extends”. The above code declares a new contract called “Voter” and the contract inherits from the other contract called “Ownable”. This code is essential in declaring the contract definition and later used as an artifact so that truffle can deploy this contract to Ethereum’s Mainnet or Testnets. In between the curly braces, type code 3.

```
mapping(string => address) voters;
    event RegisterVoter(address voter, string socialNumber);

    function _minting(address _voter) private {
        address payable voter = address(uint160(_voter));
        voter.transfer(msg.value);
    }

    function registerVoter(address _voter, string calldata _socialNumber)
        external
        payable
        onlyOwner
    {
        require(
            voters[_socialNumber] == address(0x000),
            "The voter is already registered"
        );
        voters[_socialNumber] = _voter;
        _minting(_voter);
        emit RegisterVoter(_voter, _socialNumber);
    }

    function kill() public onlyOwner {
        selfdestruct(address(uint16(owner())));
    }
```

Code 3

From code 3, the first line defines a mapping, which is the same as Dictionary in other programming languages. The mapping voters will store the voters in string to address basis, meaning the key will be the string, or the social security of the voters, and the value will be their Ethereum address. The next line defines a “type” event “RegisterVoter”, which will be emitted after the voter is successfully registered. Skip over the “_minting” function and go straight to the “registerVoter” function, the function has the arguments “_voter” of type address and

“_socialNumber” of type string. The function “registerVoter” also has the keyword “payable”, which is the function where ether can be sent to and processed by the Smart Contract. The voter will be added to the “voters” mapping, and the voter will be minted 0.0001 ether using the “_minting” function. After the operation is successful, the “RegisterVoter” event will be emitted. The reason for minting the registered voters is because the voters will not know the process of creating the Ethereum account and using it to vote. All of this process including the creation of Ethereum accounts will be abstracted from the users and handled in the backend server which is built later in the thesis. When the Ethereum account is first created, those accounts will not have any ethers, and ethers is required to call any mutable function inside Smart Contract. That is the reason why minting some ethers to voters is important since they would require ether to call the vote function in the next section.

After the code for Smart Contract is written, some tests were written to ensure the behavior of the Smart Contract is as expected. The test file is shown in figure 15.

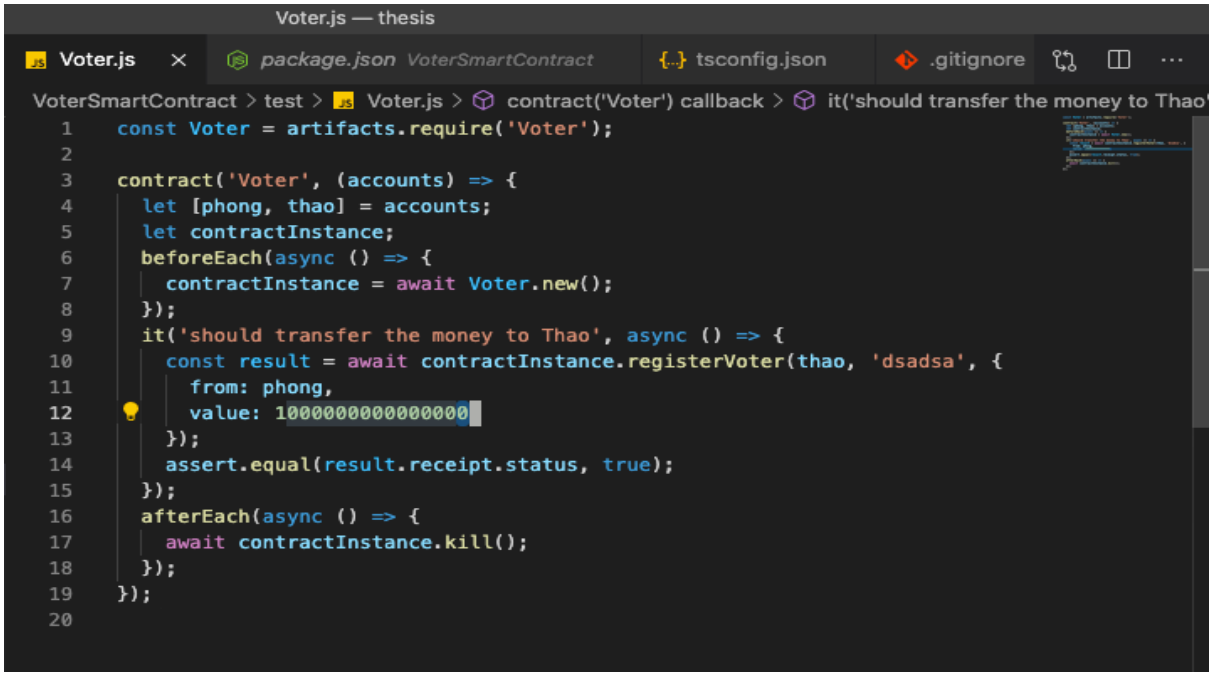



Figure 15 Test suite for Voters' SC

The test is basically checking if the function that was declared above run properly providing correct arguments and ethers. If the function is successfully run, the receipt status of the function would return true.

Next, install ganache cli into the project by typing the following command into the console:

```
npm install -D ganache-cli
```

After the package is successfully installed, navigate to package.json and add code 4.

```
"scripts": {
    "local-blockchain": "ganache-cli -p 7545 -s plutx"
  },
```

Code 4

Run the following command into the console:

```
npm run local-blockchain
```

In another console, run:

```
truffle test
```

The test should pass as in figure 16.

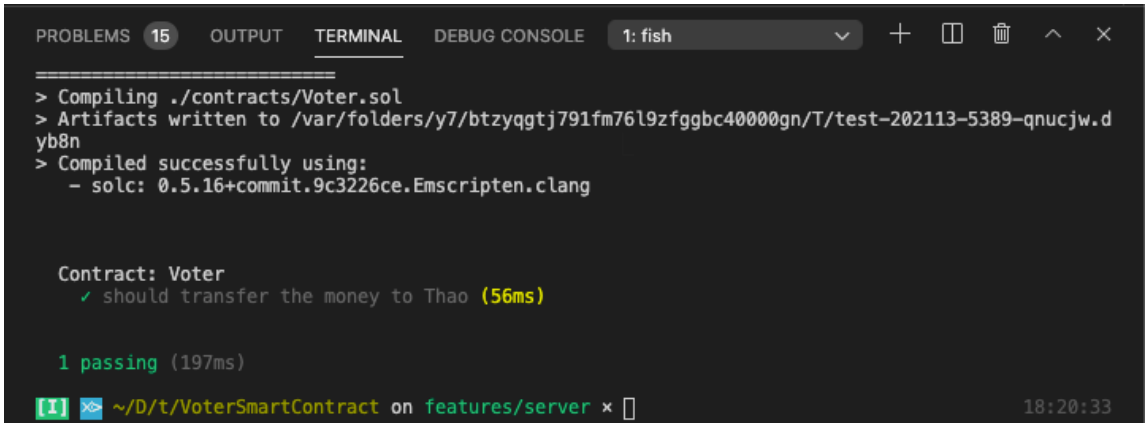



Figure 16 Voters' SC test pass

## 5.2 US Election Vote Contract

After implementing the voter contract is finished, the actual contract to vote will be the next thing to implement. I created a directory and gave it an arbitrary name such as “UsElectionVoteContract”. The setup and initialization shall be the same as above.

After the project is successfully initiated, navigate into the contract folder and create the following file: UsElectionVoteContract.sol.

Inside the file, saturate the file with code 5.

```
pragma solidity >=0.4.21;
import "@openzeppelin/contracts/ownership/Ownable.sol";

contract UsElectionVote is Ownable {
    function kill() public onlyOwner {
        selfdestruct(address(uint16(owner())));
    }
}
```

Code 5

In every contract that will be written, code 5 will be the base of the contract. The code snippet includes the version number of the contract, which is required for the contract to run inside the Ethereum Virtual Machine. The contract also includes the importation of Ownable contract, which is required for the ability of destroying the contract from the creator of the contract. Then there is the contract declaration, as explained above, and the kill function which can only be called by

the owner to destroy the contract from the Ethereum Blockchain network. Inside the contract declaration, type code 6.

```
mapping(string => address[]) candidateVotes;
    mapping(address => bool) alreadyVote;
    uint256 totalVotes;
    event Vote(address voter, string candidate);

    function vote(address _voter, string calldata _candidate) external {
        require(alreadyVote[_voter] == false, "the voter has already voted");
        require(
            keccak256(abi.encodePacked(_candidate)) ==
                keccak256(abi.encodePacked("Donald Trump")) ||
                keccak256(abi.encodePacked(_candidate)) ==
                keccak256(abi.encodePacked("Joe Biden")),
            "Invalid Candidate"
        );
        candidateVotes[_candidate].push(_voter);
        alreadyVote[_voter] = true;
        totalVotes++;
        emit Vote(_voter, _candidate);
    }

    function getVote(string calldata _candidate)
        external
        view
        returns (uint256)
    {
        return candidateVotes[_candidate].length;
    }
```

Code 6

In this contract, two mappings will be created. The mapping “candidateVotes” will store the array of addresses, or voters, for each candidate in the US Election. For example, if ‘candidateVotes[“Donald Trump”]’ was given, the result will be the array of addresses that vote for Trump. This allows for easy look up by just inserting the candidate’s name into the function call “getVote” and the number of votes will be returned immediately. The second mapping is “alreadyVote”, which is for keeping track of all voters so that no voters can vote more than once. This is important because it is not a good practice to allow voters to spam the vote button and count those votes. This is also the whole purpose of using blockchain technology to build a voting system because since there is no function to alter the “alreadyVote” mapping, there is no way to hack into the contract on the

blockchain to remove the address from “alreadyVote” mapping. The next variable is “totalVotes”, which keeps track of the total number of votes. Then the event “Vote” is declared so that later when the user has successfully voted, the event “Vote” will be emitted with the voter’s address and the candidate. After all the variables are declared, the main function of the contract, the “vote” function is created. The “vote” function takes the voter’s address and the candidate as the arguments. The function makes sure that the voter has not voted once before and the voter chooses the right candidates by including the two required statements in the function. After the requirements suffice, the mapping “candidateVotes” with the key of the candidate will push the voter’s address into its array, the voter will be marked as voted, and the event Vote will be emitted as mentioned above. The next function “getVote” with the parameter candidate returns the number of voters of each candidate. After the contract is written, the contract is tested, the code for testing this contract is shown in figure 17.

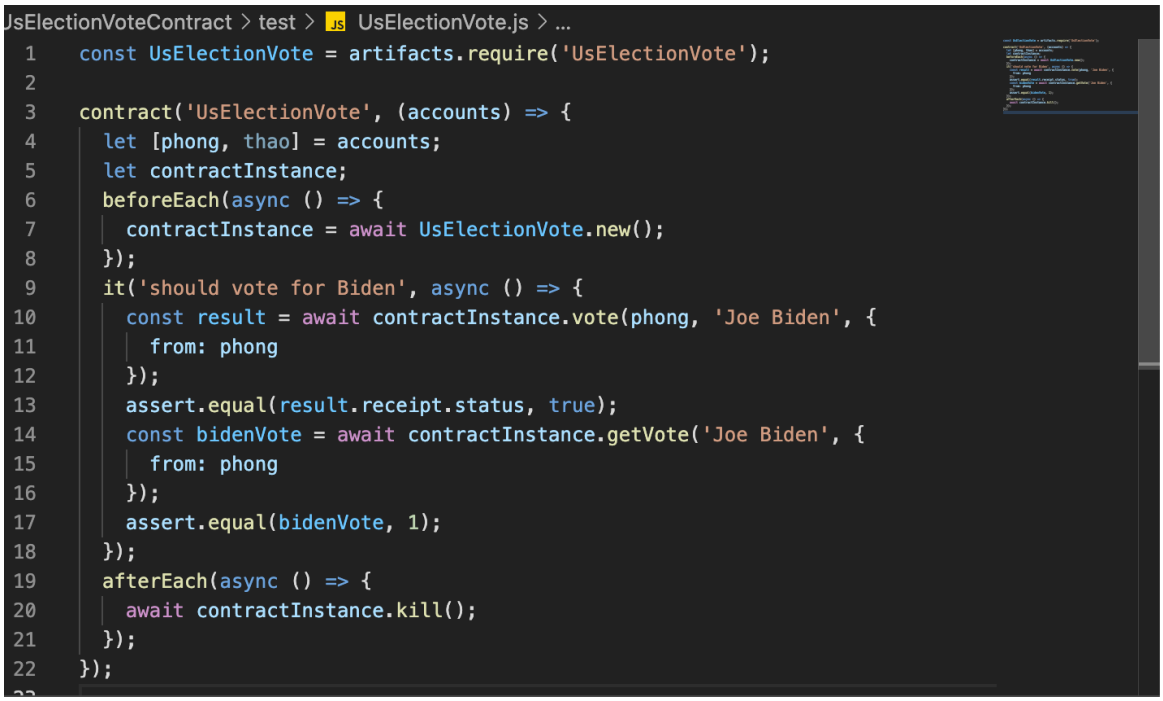



Figure 17 US Vote SC test suites

The purpose of the test was to make sure the function works properly, and the variable is saved correctly. Essentially, the candidate will vote for Joe Biden by calling the function vote and providing the appropriate arguments to the function call. After that, the getter function “getVote” is called to get Joe Biden’s result. The result should return 1 in this case since the vote function is only called one

time for Joe Biden. If the code was written correctly, the test will pass after typing in “truffle test” into the console.

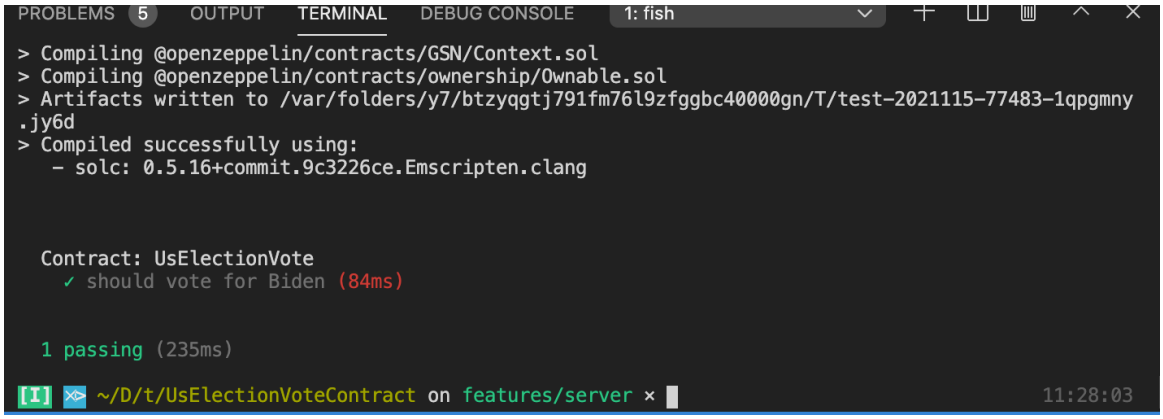



Figure 18 US Vote SC test pass

### 5.3 Backend Server

After writing the two contracts to actually handle the registering of voters and voting, it is time to actually implement a backend server to interface with the Smart Contracts that were just created. From the root project directory, type the following:

```
mkdir server
cd server
yarn init -y
```

After typing those commands, a folder named folder will be generated and a package.json file will appear inside the folder. One thing that is special about this backend server is that this server will be written in TypeScript instead of JavaScript. TypeScript is a subset of JavaScript; it is essentially the same as JavaScript except it allows type-safe when writing “JavaScript” code and better Intellisense (autocompletion and recommendation) when writing codes inside Visual Studio Code. Moreover, TypeScript catches common errors such as typos and other errors such as wrong typing when passing arguments into function call and it won’t allow the application to run if those errors exist. TypeScript is better than JavaScript in general and it enables developers to write better codes.

To use TypeScript in this service, a “tsconfig.json” file is required. Create a tsconfig.json file inside the server folder and paste code 7 into the file.

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "lib": ["dom", "es6", "es2017", "esnext.asynciterable"],
    "skipLibCheck": true,
    "sourceMap": true,
    "outDir": "./dist",
    "moduleResolution": "node",
    "removeComments": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "strictFunctionTypes": true,
    "noImplicitThis": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "allowSyntheticDefaultImports": true,
    "esModuleInterop": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "resolveJsonModule": true,
    "baseUrl": "."
  },
  "exclude": ["node_modules"],
  "include": ["./src/**/*.tsx", "./src/**/*.ts"]
}
```

Code 7

This will enable certain TypeScript rules which will be checked during the development process of this server project. When the code is written and some of the code goes against these rules, the IDE will indicate the error by having the red line right below the code. After that, an npm package called “typescript” is required to compile the TypeScript file into JavaScript. The reason for that is because TypeScript is used for developers to write better code and it enables type-safe. TypeScript is not used to run the actual NodeJS program, nor does it use to run on the browser because some code in TypeScript cannot be interpreted in the NodeJS or the browser environment. It is only used solely for

development purposes, and some of the code in TypeScript will not be compiled into JavaScript. That is why in every TypeScript project, there is always a build script to compile TypeScript into JavaScript and then after that run the built JavaScript file.

To download “typescript”, run the following command:

```
yarn add typescript
```

The package will be downloaded and added into the package.json file. After that, open package.json file and add the following into the script section of the file:

```
"build": "tsc",
```

That allows compiling every single TypeScript file in the “src” folder of the server project into JavaScript and output the result into the “dist” folder. All of these settings and configurations are available in “tsconfig.json” above.

### 5.3.1 Initiate the server

After setting up TypeScript for the Backend service, run the command *“`yarn add express`”* to install ExpressJS into the project. Since the project will be written in TypeScript, the declarative type for ExpressJS is also required. Run the command *“`yarn add -D @types/express`”* to install the module as a development dependency. After it is installed, create a source folder to host the TypeScript files. Run the command *“`mkdir src`”* from the backend service project folder and create an “index.ts” file. Inside the file, saturate the file by typing code 8.

```
import express, { Response, Request } from 'express';

const app = express();

const start = () => {
  app.get('/', (req: Request, res: Response) => {
    res.send('<h1>Hello</h1>');
  });

  app.listen(5000);
```

```
};

start();
```

Code 8

The first line imports “express” and other interfaces from the ExpressJS module that was installed using “yarn”. The second line of the code initialize the express application by invoking the “express()” function and assign it to a variable “app”. After that, a “start” function is defined and inside the function, the “app” will request an HTTP GET request to the “/” route to the server, and when that happens, the server will respond with “<h1>Hello</h1>”. Lastly, the “app” will listen at port 5000. At the end of the file, the “start” method is invoked. After writing out the codes, go to package.json and add the following scripts under the build script:

```
"watch": "tsc -w",
    "start": "node dist/index.js",
    "dev": "nodemon dist/index.js",
```

The “watch” script will watch for the changes in the TypeScripts file and update the JavaScript outputs after the files have been saved. The “start” script is for running the server using the JavaScript output that the “watch” or the “build” script produce. The downside of this script is that every time the file is changed in the project, meaning if the “watch” script is catching something from the TypeScript files and update it accordingly, the “start” script needs to be restarted every time for the changes to get reflected on the application. That is why inside the development environment, “nodemon” is commonly used. “nodemon” allows the application to restart every time there is any file changes in the application. Navigate to the command prompt and run the following:

```
yarn watch
```

In another terminal, navigate into the server folder and run the command:

```
yarn dev
```

After running that, navigate into the browser and navigate to http://localhost:5000


Figure 19 Hello world express

The webpage shall display like figure 19.

### 5.3.2 Connect to Ethereum network and Smart Contracts

Now that the server is up and running, it is time to interact with the Smart Contracts using a library called Web3.js. to install Web3.js, open another terminal tab and type in:

```
yarn add web3 ethereumjs-tx dotenv
yarn add -D @types/ethereumjs-tx
```

This will install Web3.js, EthereumJS, dotenv, and the type declaration for EthereumJS into the server folder. EthereumJS is for simplifying the way to sign the transactions produced by the Web3.js library. Dotenv is handy for accessing environment variables from the ".env" file which will be created after this step. Inside the root of the server service folder, create a file called ".env" file and add code 9 into the file.

```
FULL_NODE_URL=https://rinkeby.infura.io/v3/b6ea1cd9d4a64650a661639e777e6919
VOTER_SC_ADDRESS=0xA9FdDeBfA9C160e013BFdE473E83b2A6292eE98d
US_ELECTION_VOTE_SC_ADDRESS=0x761D1a23484f32D36b7038Fdf6BA46b5247Ab255
ADMIN_PRIVATE_KEY=e75593c07554c41cc48971a4454dcaf21da3616ad55c3d3e8747f8acd0715
e19
```

```
ADMIN_ADDRESS=0xbc5aC9e4bEe4aAE9F0D97F27d9e81B3eBDC8a39a
DB_URL=postgresql://postgres:2606@localhost:5432/thesis-project
ENCRYPTED_KEY=FoCKvdLslUuB4y3EZlKate7XGottHski1LmyqJHvUhs=
```

*Code 9*

This is the information that will be needed for the connection of the Smart Contracts and calling methods from them. In order to get access to these variables using "*process.env.**", add code 10 into "index.ts".

```
import dotenv from 'dotenv'
dotenv.config()
```

*Code 10*

It will allow to get access to environment variables such as "FULL_NODE_URL" using "process.env.FULL_NODE_URL". Inside the "src" folder, create a "utils" folder and inside that folder, create a file called "web3Instance.ts". Inside the file, type code 11.

```
export const web3Instance = () => {
  const web3 = new Web3(
    new Web3.providers.HttpProvider(process.env.FULL_NODE_URL)
  );
  const voterAbi = Voter.abi as any;
  const usElectionVoteAbi = UsElectionVoteContract.abi as any;
  const voterContract = new web3.eth.Contract(
    voterAbi,
    process.env.VOTER_SC_ADDRESS
  );
  const usElectionVoteContract = new web3.eth.Contract(
    usElectionVoteAbi,
    process.env.US_ELECTION_VOTE_SC_ADDRESS
  );
  return { web3, voterContract, usElectionVoteContract };
};
```

*Code 11*

Code 11 allows the server application connects with the Smart Contracts using an Ethereum Node Provider called given the "FULL_NODE_URL". After that, the Smart Contracts will be connected by Web3.js using its "ABI" and the Smart Contracts' addresses which is provided in the environment variables. The

contracts' "ABIs" can be extracted by navigating into the contracts' folders, and do the following:

- "`npm i @truffle/hdwallet-provider`" to install the necessary dependencies.
- Go to "truffle-config.js" and uncomment the following line: *const HDWalletProvider = require('@truffle/hdwallet-provider');*
- After that, inside the network object, comment everything beside the development object.
- In terminal, run "`truffle build`" and the necessary "ABIs" will be available in the build folder.
- Copy that "ABIs" file, inside the server folder, create a folder called "abis" inside the "src" folder and paste the "ABIs" into that folder.

After the Web3.js has successfully connected to the "FULL_NODE_URL" and the Smart Contracts, the function "web3Instance" will return the "web3" object which is used to interact with Ethereum network and creating transactions, and the instance of the two contracts which will be used to invoke methods inside the two contracts.

### 5.3.3 Connect to PostgreSQL

After establishing connection to the Smart Contracts, it is a good time to establish a connection to a PostgreSQL database to store the social number, the address, and the private key of the users. The reason for that is because the user will enter their social security number when they want to vote for a candidate, and that social number will be used to get accessed to the user's address and private key. In order to establish a connection to the Postgres database, navigate into the terminal and type the following:

```
yarn add pg reflect-metadata typeorm
```

This command installs the Postgres plugin for NodeJS (pg), TypeORM (which is used to interface with the database) and Reflect Metadata into the project. Navigate to "index.ts" and add the following import statements:

```
import 'reflect-metadata';
import { createConnection } from 'typeorm';
import path from 'path';
```

At the beginning of the “start” function, type code 12.

```
const connection = await createConnection({
    type: 'postgres',
    url: process.env.DB_URL,
    logging: true,
    synchronize: false,
    migrations: [path.join(__dirname, './migrations/*')],
    entities: [Voter]
  });
  await connection.runMigrations();
```

*Code 12*

The connection variable is for establishing the connection to the Postgres database in the using the given URL, in this case is the URL provided by the environment variables (for anyone who is reading this and are trying to reproduce the steps in this thesis, use your own DB_URL for connecting to the database). This also enables logging when performing any query with the database, configuring migration folders to host any migration files, an entities array which will be discussed later. The last line is used to run any existing migrations. To create the Voter entity, create a folder called “entities” inside the “src” folder and inside “entities”, create “Voter.ts”. Type code 13 into the file.

```
import { BaseEntity, Column, Entity, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class Voter extends BaseEntity {
  @PrimaryGeneratedColumn('uuid')
  id: string;
  @Column()
  socialNumber: string;
  @Column()
  address: string;
  @Column()
  privateKey: string;
}
```

*Code 13*

Code 13 is for creating and defining the Voter table into the Postgres database. The equivalent SQL table of the above code is:

| id | Type: uuid |
|---|---|
| socialNumber | Type: text |
| address | Type: text |
| privateKey | Type: text |

Import the file into “index.ts” file.

```
import { Voter } from './entities/Voter'
```

Look at the “`yarn dev`” terminal and the database logging should be displayed, and it looks similar to figure 20.

```
[I] ~/D/t/server on master x yarn dev
yarn run v1.22.4
$ nodemon dist/index.js
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node dist/index.js`
query: SELECT * FROM "information_schema"."tables" WHERE "table_schema" = current_schema() AND "table_name" = 'migrations'
query: SELECT * FROM "migrations" "migrations"  ORDER BY "id" DESC
```

Figure 20 Sample Postgres Logging

To tell the database to create the voter tables, create a file called “ormconfig.json” outside of “src” folder. Type the code 14.

```
{
    "type": "postgres",
    "url": "postgresql://postgres:2606@localhost:5432/thesis-project",
    "logging": true,
    "synchronize": false,
    "migrations": ["dist/migrations/*.js"],
    "entities": ["dist/entities/*.js"]
}
```

*Code 14*

After that, navigate to the terminal and type the following command:

```
npx typeorm migration:generate -n Voter
```

This will generate a migration file that has the code to create the voter table via migrate up and drop the table via migrate down. Inside “src” folder, create “migrations” folder and paste the generated migration file into migration folder. After doing that, the server should restart, and the log from figure 21 should show up.

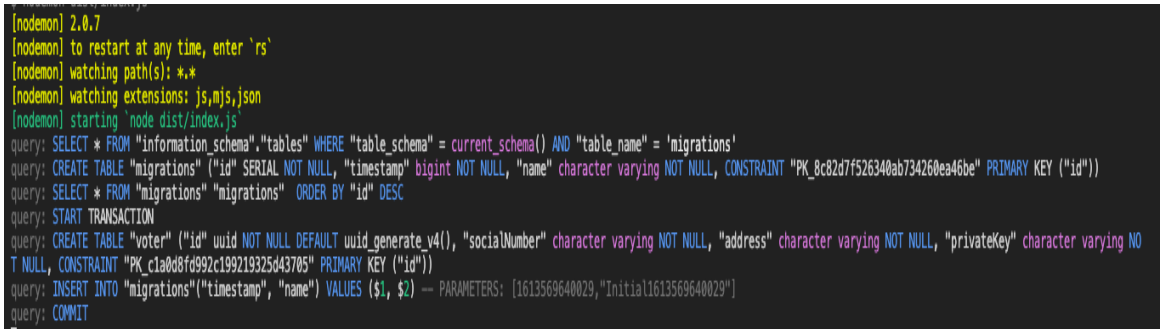



Figure 21 After migration

### 5.3.4 Create Route Handler for Registering Voter

After establishing a connection to the Postgres database, it is time to create a route to handle registering voters. In the “src” folder, create a file called “registerVoters.ts”. Inside the file, there is a route handler for registering a voter:

```
app.post('/register-voter', async (req: Request, res: Response) => {
const { socialNumber } = req.body;
    const existingSocialNumber = await Voter.findOne({ socialNumber });
    if (existingSocialNumber) {
      return res
        .status(400)
        .send({ error: 'The social number is already registered' });
    }
```

Code 15

There is a lot going with code 15. Essentially, when the user sends a POST request to localhost:5000/register-voter with his or her social number, the server will first check inside the database whether the user has already registered an

account. If the social number exists inside the database, the server will send a response with a status code of 400 and an error message. If the user has not registered an account, the server will generate an Ethereum address and private key using Web3 library. Then the next part is for creating Transaction object for creating a voter in Voter Smart Contract and signing that transactions using the Admin Private Key (Smart Contract Owner). All of that logic is expressed in code 16.

```
const account = web3.eth.accounts.create();
const data = await voterContract.methods.registerVoter(
     account.address,
     socialNumber
   );
   const txObj: TxObj = {
     from: process.env.ADMIN_ADDRESS,
     data: data,
     to: process.env.VOTER_SC_ADDRESS,
     value: 1000000000000000
   };
   const adminPrivateKey = process.env.ADMIN_PRIVATE_KEY;
   let tx = await createTransaction(txObj);
   let signedTx = await signTransaction(tx, adminPrivateKey);
   return await web3.eth
     .sendSignedTransaction(signedTx)
```

*Code 16*

After the transaction has been sent, the server will listen to two events: confirmation, and error. Error events means that there is something wrong when sending the transaction to the blockchain, i.e., network error, … The confirmation event is tracking whether the transaction is being processed and added onto the blockchain ledger. Inside the confirmation block, the server is checking if the transaction is successful. If it is not successful, the server will send a 400 response with an error message indicating that the Transaction failed. If it is succeeded, the server will encrypt the generated private key, and save the social number, the address, and the encrypted private key onto the database. After that, the server will response with a 200 status code and the response body will be the confirmation number, status, and a message saying that the account has been successfully registered. All these logics are implemented in code 17.

```
.on('confirmation', async (confirmationNumber, receipt) => {
        confirmNum++;
        if (confirmNum === 2) {
          if (!receipt.status) {
            res.status(400).send({ error: 'Transaction failed' });
          } else {
            const cipher = crypto.createCipher(
              'aes-128-cbc',
              process.env.ENCRYPTED_KEY
            );
            let ciphertext = cipher.update(
              account.privateKey,
              'utf8',
              'base64'
            );
            ciphertext += cipher.final('base64');
            await Voter.create({
              socialNumber,
              address: account.address,
              privateKey: ciphertext
            }).save();
            console.log('Send Request');
            res.send({
              confirmationNumber,
              status: receipt.status,
              message: 'Account has been registered'
            });
          }
        }
      })
```

*Code 17*

### 5.3.5 Create route handler for voting and get votes

After creating a route handler for registering a voter, let's start creating a route to do the actual voting. Inside the "src" folder, create a file called "usElectionVote.ts". The "usElectionVote.ts" file will have two route handlers. One is for handling the voting:

```
app.post('/us-election-vote', async (req: Request, res: Response) => {
```

and handling the fetching of the vote:

```
app.get('/vote-result', async (_req, res: Response) => {
```

The route that handling the voting receives a POST request that includes the user's social number and the candidate's name. The server will check if the voter has already been registered. If the user has not registered, the server will response with a status code of 400. If the user has registered, then the Transaction object is defined to create a transaction, and the server will sign that transaction using the voter's private key.

```
  const { candidate, socialNumber } = req.body;
    const voter = await Voter.findOne({ socialNumber });
    if (!voter || !voter.address) {
      res
        .status(400)
        .send({ error: 'The voter does not exist or not yet registered' });
    }
```

*Code 18*

Code 18 is checking if the voter has already registered by checking the database if there is an entry with the social number. If the user has not registered, an error will be sent to the user.

```
else {
      const data = await usElectionVoteContract.methods.vote(
        voter.address,
        candidate
      );
      const txObj: TxObj = {
        from: voter.address,
        data: data,
        to: process.env.US_ELECTION_VOTE_SC_ADDRESS
      };
      const encryptedPrivateKey = voter.privateKey;
      const decipher = crypto.createDecipher(
        'aes-128-cbc',
        process.env.ENCRYPTED_KEY
      );
      let privateKey = decipher.update(encryptedPrivateKey, 'base64', 'utf8');
      privateKey += decipher.final('utf8');
      console.log('private key: ', privateKey);
      let tx = await createTransaction(txObj);
      let signedTx = await signTransaction(tx, privateKey);
      await web3.eth
        .sendSignedTransaction(signedTx)
```

*Code 19*

In code 19, if the voter is found, the voting operation begins by creating the transaction, decrypting the private key from the database, and use that private key to sign the transaction. After doing all of those, the transaction will be sent to the blockchain. Similar to register voter route handler, the server will listen to two events: confirmation and error. If there is any error event, most likely the error is due to a connection problem or something catastrophic has happened in the blockchain. If the confirmation event is received, the server will send a 400 status if the transaction is failed or a 200 status if the transaction is succeeded.

```
.on('confirmation', async (confirmationNumber, receipt) => {
          confirmNum++;
          if (confirmNum === 2) {
            if (!receipt.status) {
              res.status(400).send({ error: 'You have already voted' });
            } else {
              res.send({
                confirmationNumber,
                status: receipt.status,
                message: `You have voted for ${candidate}`
              });
            }
          }
        })
```

*Code 20*

The last route is the get vote route handler. This route is simple. It receives a GET request to “/vote-result” and the server will make a request to the blockchain server to fetch the voting data of the two candidates. After that the server will send a response with a status code of 200 and the data that the server fetched.

```
app.get('/vote-result', async (_req, res: Response) => {
    const joeBidenVote = await usElectionVoteContract.methods
      .getVote('Joe Biden')
      .call();
    const donaldTrumpVote = await usElectionVoteContract.methods
      .getVote('Donald Trump')
      .call();
    res.send({ joeBidenVote, donaldTrumpVote });
  });
```

*Code 21*

## 5.4 Client

After building everything the foundation of the application, it is a good time to develop the User Interface so that the users can register the voter account and vote for their candidates. In the root project folder, type the following into the terminal:

```
npx create-react-app client –template typescript
```

This will create a folder named “client” and scaffold the ReactJS project into that folder, and “—template typescript” added TypeScript support into the project. Navigate into the folder and start removing everything in the “client/src” folder beside the “index.tsx” file. In the terminal, type the following:

```
yarn add axios @chakra-ui/react
```

This installs Axios (a library for HTTP request) and Chakra UI (a custom styling library). Inside “index.tsx”, code 21 is used to make the project utilize Chakra’s components.

```
const colors = {
  brand: {
    900: '#1a365d',
    800: '#153e75',
    700: '#2a69ac'
  }
};
const theme = extendTheme({ colors });
ReactDOM.render(
  <React.StrictMode>
    <ChakraProvider theme={theme}>
      <App />
    </ChakraProvider>
  </React.StrictMode>,
  document.getElementById('root')
);
```

*Code 22*

Code 22 will allow React to use components from Chakra UI and save a lot of time from styling the elements using CSS.

Inside the “src” folder, create “App.tsx” file. Inside the file, code 22 is used to fetch the candidates’ voting result to the application.

```
const [donaldTrumpVote, setDonaldTrumptVote] = useState<number | null>(null);
  const [joeBidenVote, setJoeBidenVote] = useState<number | null>(null);
  const [voted, setVoted] = useState(false);
  useEffect(() => {
    axios
      .get<IVoteResultResponse>('http://localhost:5000/vote-result')
      .then(({ data: { joeBidenVote, donaldTrumpVote } }) => {
        setDonaldTrumptVote(donaldTrumpVote);
        setJoeBidenVote(joeBidenVote);
      });
  }, [voted]);
```

*Code 23*

Code 23 makes use of “useEffect” hook, which is a powerful feature in React. Essentially, when the app is first loaded onto the browser, the application will fetch the vote result by requesting to *http://localhost:5000/vote-result* using Axios and after that append the result to “donaldTrumpVote” and “joeBidenVote” respectively. This application also makes use of “useState” hook, which is an alternative for the React states using “Class based Components”. At the end of the “useEffect”, there is an array with the state “voted” in it. The array is a dependencies array, by default the “useEffect” only runs once if the array is empty. However, in this case, “voted” is inside the array, meaning if there are any changes in “voted” state, the “useEffect” will be triggered and run again. After that, the application will render these components in code 24.

```
return (
    <Layout>
      <VoteForm voted={voted} setVoted={setVoted} />
      <Flex>
        <Result
          voteNumber={joeBidenVote}
          imageSrc='joe-biden.jpeg'
          imageAlt='Joe Biden'
          candidateHeading='Joe Biden'
        />
        <Result
          voteNumber={donaldTrumpVote}
          imageSrc='donald-trump.jpeg'
```

```
          imageAlt='Donald Trump'
          candidateHeading='Donald Trump'
        />
      </Flex>
    </Layout>
  );
```

*Code 24*

The application will render the form for voting the candidates and the result component for each candidate. There is the “Result” component inside the return statement. The “Result” component has the duty to render the voting result of each candidate accordingly. Another component to notice is the “Layout” component, which is written in “Layout.tsx”. Inside “Layout.tsx”, there is a “NavBar” component according to code 25.

```
export const Layout: React.FC<LayoutProps> = ({ variant, children }) => {
  return (
    <>
      <NavBar />
      <Wrappper variant={variant}>{children}</Wrappper>
    </>
  );
};
```

*Code 25*

Inside “NavBar.tsx”, there is a button that triggers the “SocialNumberForm” (code 26).

```
  <Box ml='auto'>
          <Flex align='center'>
            <Box mr={4}>
              <Button onClick={onOpen} mr={4}>
                Register to the system
              </Button>
            </Box>
            <SocialNumberForm isOpen={isOpen} onClose={onClose} />
          </Flex>
        </Box>
```

*Code 26*

Inside the “SocialNumberForm” component which is written in “SocialNumberForm.tsx”, a form for the user to register their social number into the system is expressed in code 27.

```
<Modal isOpen={isOpen} onClose={onClose}>
        <ModalOverlay />
        <ModalContent>
          <ModalHeader>Register to voting system</ModalHeader>
          <ModalCloseButton />
          <ModalBody>
            <form onSubmit={onSubmit}>
              <FormControl>
                <FormLabel>Social Number: </FormLabel>
                <Input
                  value={socialNumber}
                  onChange={(e) => {
                    setSocialNumber(e.target.value);
                  }}
                />
                <Button isLoading={loading} mt={4} type='submit'>
                  Submit
                </Button>
              </FormControl>
            </form>
          </ModalBody>
        </ModalContent>
      </Modal>
```

*Code 27*

The form essentially asks for the user's social security number, and the user can click submit to send the request onto the backend server. The request is being handled in code 28.

```
const [socialNumber, setSocialNumber] = useState<string>('');
  const [loading, setLoading] = useState<boolean>(false);
  const onSubmit = async (e: SyntheticEvent) => {
    console.log('submitting');
    e.preventDefault();
    try {
      setLoading(true);
      await axios.post('http://localhost:5000/register-voter', {
        socialNumber
      });
      toastSuccess(`${socialNumber} has been registered`);
    } catch (e) {
      console.log(e.response.data.error);
      toastError(e.response.data.error);
    }
    setSocialNumber('');
    setLoading(false);
    onClose();
```

```
  };
```

*Code 28*

There is a “useState” definition for “socialNumber”, which was plugged into the form input and it recorded the value from the input accordingly. After that, there is an “onSubmit” handler, which will trigger when the user submits the form. Inside the function, there is an “Axios” POST request call to the server at *https://localhost:5000/register-voter* with the “socialNumber” as the request body. After the request is succeeded or failed, the corresponding “Toast” message will be displayed onto the screen.

Another important to look at is “VoteForm”, which is rendered in “App.tsx”. The “VoteForm” component is written inside of “VoteForm.tsx” (code 29).

```
return (
    <>
      <Heading as='h3' size='lg'>
        Here is the US Election vote, choose your candidate
      </Heading>
      <form onSubmit={onSubmit}>
        <FormControl mt={4}>
          <FormLabel>Your Social Number</FormLabel>
          <Input
            value={socialNumber}
            onChange={(e) => setSocialNumber(e.target.value)}
          />
        </FormControl>
        <FormControl>
          <FormLabel mt={4}>Choose one</FormLabel>
          <Select
            defaultValue=''
            onChange={(e) => setCandidate(e.target.value)}
          >
            <option value=''></option>
            <option value='Joe Biden'>Joe Biden</option>
            <option value='Donald Trump'>Donald Trumpt</option>
          </Select>
        </FormControl>
        <Button
          colorScheme='yellow'
          type='submit'
          isLoading={loading}
          mt={4}
          mb={4}
        >
```

```
          Vote
        </Button>
      </form>
    </>
  );
```

*Code 29*

In code 29, there is a form that asks for the user's social security number and the candidate to vote. The user will input the social number to the text input and there is a select input so that the user can choose their candidate to vote for. After the user has finished inputting the values into the form, the user can hit submit and their input will be sent to the server so that the server can later make request onto the Smart Contracts. The code for handling the submit is in code 29.

```
const [socialNumber, setSocialNumber] = useState('');
  const [candidate, setCandidate] = useState('');
  const [loading, setLoading] = useState(false);
  const onSubmit = async (e: SyntheticEvent) => {
    e.preventDefault();
    if (!candidate) {
      console.log('Must select a candidate');
    }
    try {
      setLoading(true);
      await axios.post('http://localhost:5000/us-election-vote', {
        socialNumber,
        candidate
      });
      toastSuccess(`${socialNumber} has voted for ${candidate}`);
      setVoted(!voted);
    } catch (e) {
      console.log(e.response.data.error);
      toastError(e.response.data.error);
    }
    setSocialNumber('');
    setLoading(false);
  };
```

*Code 30*

After putting everything together, the application shall look like the following figures.

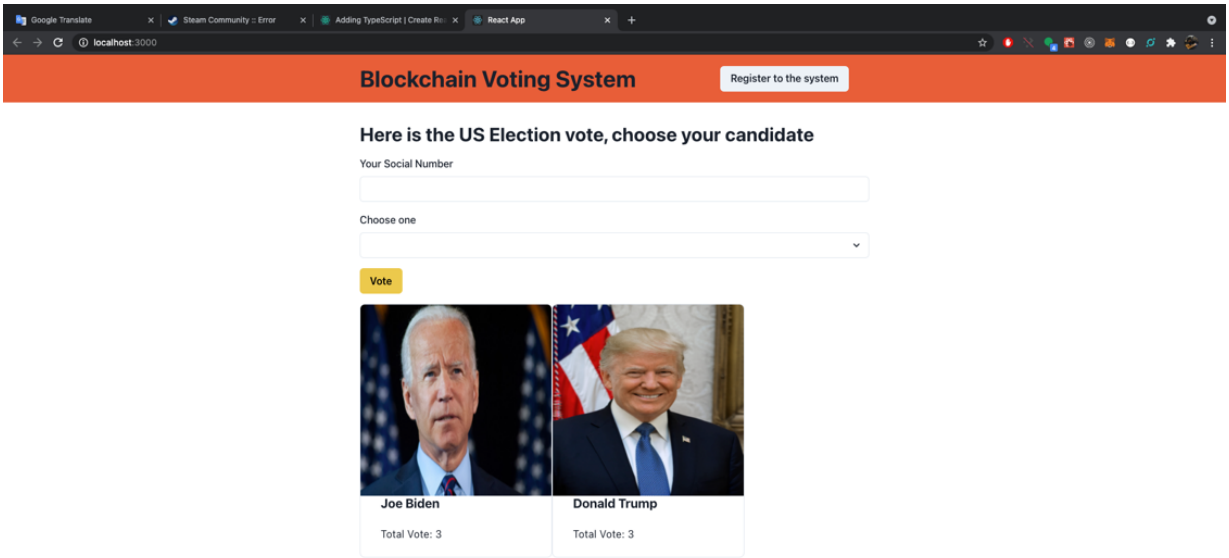



Figure 22 Application first startup

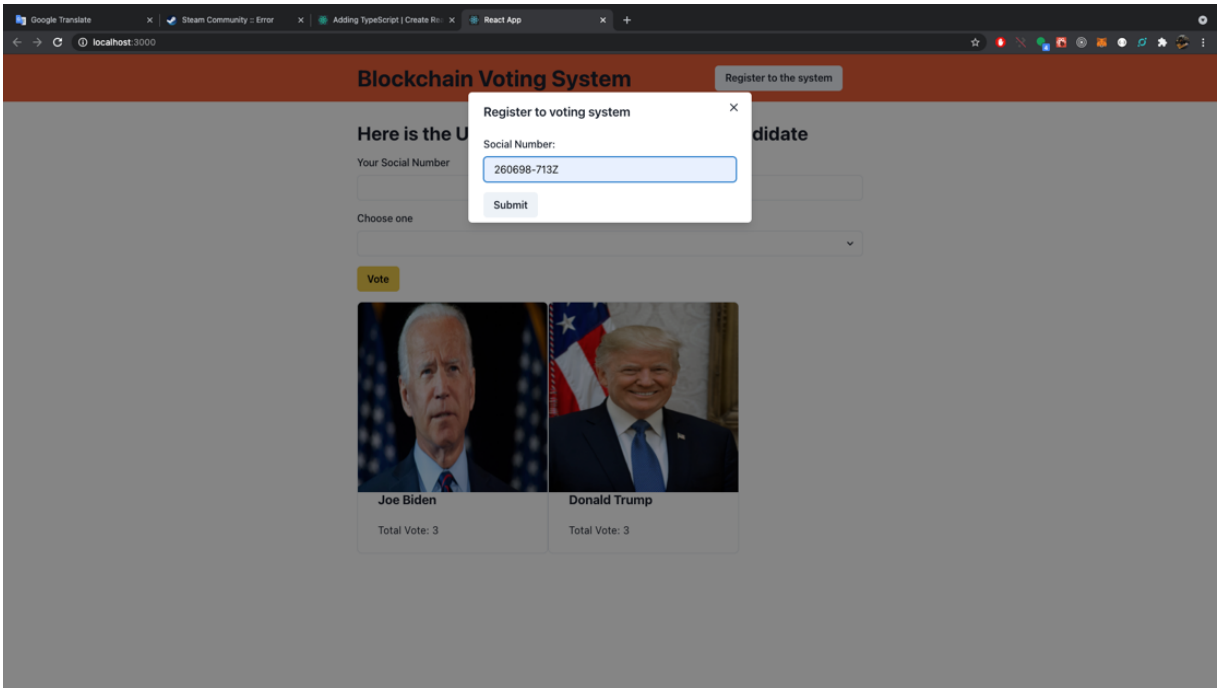



Figure 23 Register voter form

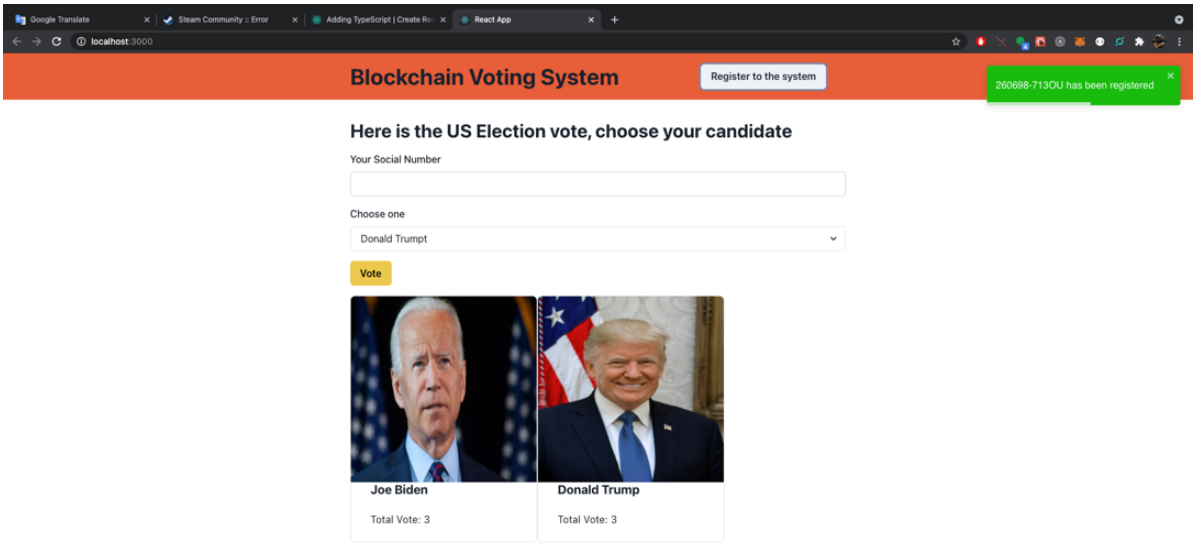



Figure 24 Successful registration

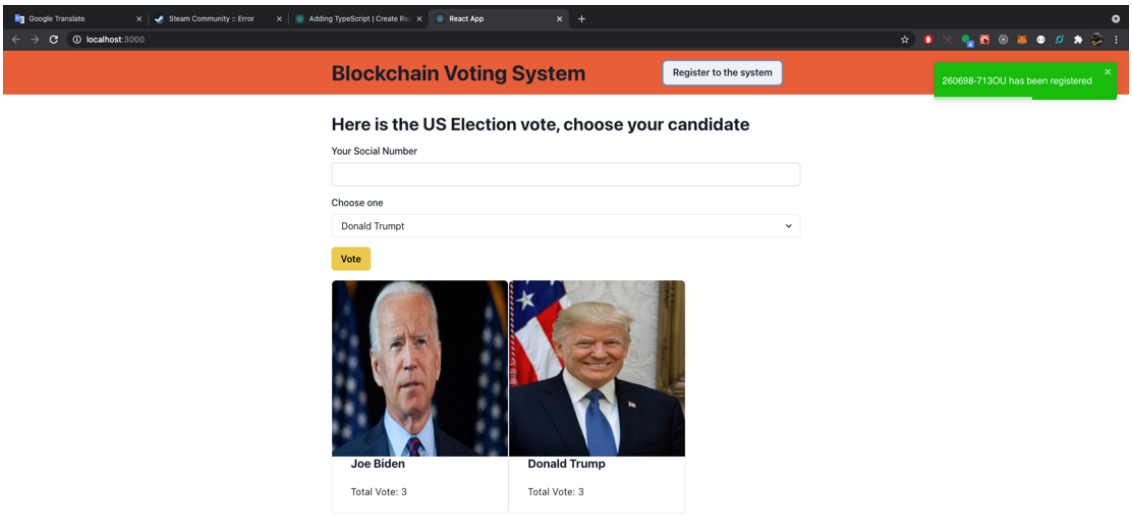



Figure 25 Successful Vote

## 6 RESULTS AND DISCUSSION

The application turns out to be a success and the users can use their social security number to vote. According to the application, every single voting request that is sent to the Smart Contracts (code 31) required the transaction to be signed by the voter. Moreover, the voters need to abide by the rules that are

defined in the Smart Contract (code 32) so that the voters do not vote twice and the voter vote for the right candidates. This is to avoid any possible mutation by a third or direct party who is managing the system so that the result of the voting will always be transparent.

```
const txObj: TxObj = {
        from: voter.address,
        data: data,
        to: process.env.US_ELECTION_VOTE_SC_ADDRESS
      };
      const encryptedPrivateKey = voter.privateKey;
      const decipher = crypto.createDecipher(
        'aes-128-cbc',
        process.env.ENCRYPTED_KEY
      );
      let privateKey = decipher.update(encryptedPrivateKey, 'base64', 'utf8');
      privateKey += decipher.final('utf8');
      console.log('private key: ', privateKey);
      let tx = await createTransaction(txObj);
      let signedTx = await signTransaction(tx, privateKey);
```

Code 31

```
        require(alreadyVote[_voter] == false, "the voter has already voted");
        require(
            keccak256(abi.encodePacked(_candidate)) ==
                keccak256(abi.encodePacked("Donald Trump")) ||
                keccak256(abi.encodePacked(_candidate)) ==
                keccak256(abi.encodePacked("Joe Biden")),
            "Invalid Candidate"
        );
```

Code 32

If the voter is voting more than once, the result will be display as in figure 26

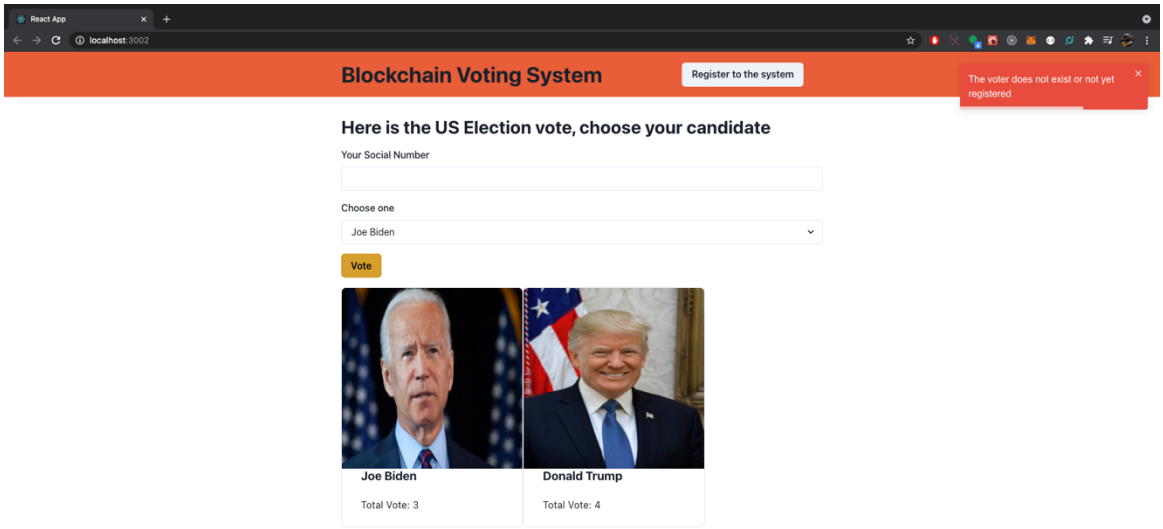



Figure 26 Failed transaction

One way to improve this application is to actually check whether or not the user actually provides the correct social security number. Currently in the application, there is no mechanism to check the validity of the social security number, so a user can register multiple times and vote for a candidate. In order to do that, access to the government database is required, and it is currently outside the scope of this application. The good thing about this application is that all voting choices are completely anonymous, and there is no way to actually see the voter's identity inside the Smart Contracts since there is no function to get those values. Another possibility to consider in order to improve this application is to make this website a little bit more responsive and improve the UI a little bit. The purpose of this thesis is about blockchain introduction and implementation, so styling in this application is outside the scope, but it is a consideration if this application needs further improvement. Finally, currently in the application, the voters do not need to pay any money to vote for the candidate, and it is the contract owner who send ETH to the voters so that the voters can pay the gas fees and vote for the candidates. It is one downside about this application because it would require the government or anyone who is using this application to spend money to pay for the vote of the voters. The application can be downloaded from *https://github.com/phongtra/thesis/tree/master*. Read the instruction in the README.md section to run the application and test it out.

# 7 CONCLUSION

The application that was implemented solved the original task of building a secure voting system. This application does not allow any mutations to the voting result, and it is the original goal of this application. The code from the application works well in real life, it is only required for the Smart Contracts to be on Mainnet, and an access to the government central database to check the validity of the social security numbers. One other way to do this project is to build a secure distributed system and provide restriction to the database so that only authorized personnel can access and query the database. Even that cannot prevent any mutation to the database because the authorized personnel can tamper with the vote result themselves and it defeats the purpose of the application in the first place. A background check is required to elect the trusted personnel to manage the database which stores the voting results, and even that is risky because there is no way to know if there is any outside influence on that person that could make him change his mind. With blockchain and Smart Contracts involved, the system can be decentralized and there is no need to worry about finding a trustworthy person to manage the database.

## REFERENCES

Antonopoulos, AA., (). Mastering Bitcoin: Unlocking Digital Cryptocurrencies. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media.

CNN., (2020). 2016 Presidential Campaign Hacking Fast Facts [online]. CNN editorial research. [Viewed 15 March 2021]. Available at: https://edition.cnn.com/2016/12/26/us/2016-presidential-campaign-hacking-fast-facts/index.html

Donovan., (2019) The Top 10 Frameworks and What Tech Recruiters Need to Know About Them. [Viewed 24 March 2021] Available from: https://stackoverflow.blog/2019/12/17/the-top-10-frameworks-and-what-tech-recruiters-need-to-know-about-them/

IBM., () What is Blockchain Technology? [Viewed 1 May 2021]. Available at: https://www.ibm.com/topics/what-is-blockchain#:~:text=Blockchain%20is%20a%20shared%2C%20immutable,Start%20now%20on%20IBM%20Blockchain

Jain., (2018). Merkle-Tree [online]. GitHub. [Viewed 15 March 2021]. Available at: https://github.com/anudishjain/Merkle-Tree

Lavayssière., (2018) Mastering Ethereum. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media.

React Documentation., (2020) The Official React Documentation [Viewed 24 March 2020] Available at: https://reactjs.org/docs

Shashank., (2019) What are Smart Contracts? [Viewed 15 March 2021]. Available at: https://www.edureka.co/blog/smart-contracts/

Selfkey., (2019). What is a Merkle Tree and How Does it Affect Blockchain Technology? [online]. Selfkey. [Viewed 15 March 2021]. Available at:

https://selfkey.org/what-is-a-merkle-tree-and-how-does-it-affect-blockchain-technology/

Use The Bitcoin., (2020). Ethereum's Switch to Proof of Stake – Better Than Proof of Work? [online]. Use the Bitcoin. [Viewed 15 March 2021]. Available at: https://usethebitcoin.com/ethereums-switch-proof-work-proof-stake/

Wackerow., (2020). PROOF-OF-STAKE (POS) [online]. Ethereum. [Viewed 15 March 2021]. Available at: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/

Web3 Documentation., (2020) The Official Web3 Documentation [Viewed 24 March 2020] Available at: https://web3js.readthedocs.io/en/v1.3.0/getting-started.html

# APPENDICES

## Vote Contract

Voter.sol

```
pragma solidity >=0.4.21;
import "@openzeppelin/contracts/ownership/Ownable.sol";

contract Voter is Ownable {
    mapping(string => address) voters;
    event RegisterVoter(address voter, string socialNumber);

    function _minting(address _voter) private {
        address payable voter = address(uint160(_voter));
        voter.transfer(msg.value);
    }

    function registerVoter(address _voter, string calldata _socialNumber)
        external
        payable
        onlyOwner
    {
        require(
            voters[_socialNumber] == address(0x000),
            "The voter is already registered"
        );
        voters[_socialNumber] = _voter;
        _minting(_voter);
        emit RegisterVoter(_voter, _socialNumber);
    }

    function kill() public onlyOwner {
        selfdestruct(address(uint16(owner())));
    }
}
```

test/Voter.js

```
const Voter = artifacts.require('Voter');

contract('Voter', (accounts) => {
  let [phong, thao] = accounts;
  let contractInstance;
  beforeEach(async () => {
    contractInstance = await Voter.new();
  });
  it('should not register voter if the user has already registered', async () => {
    await contractInstance.registerVoter(thao, 'dsadsa', {
      from: phong,
      value: 1000000000000000      You, a month ago • Squashed commit of the following:
    });
    try {
      await contractInstance.registerVoter(thao, 'dsadsa', {
        from: phong,
        value: 1000000000000000
      });
      assert(false);
    } catch (e) {
      assert(true);
    }
  });
  it('should transfer the money to Thao', async () => {
    const result = await contractInstance.registerVoter(thao, 'dsadsa', {
      from: phong,
      value: 1000000000000000
    });     any
    assert.equal(result.receipt.status, true);
  });
  afterEach(async () => {
    await contractInstance.kill();
  });
});
```

## US Election Vote Contract

UsElectionVoteContract.sol

```
pragma solidity >=0.4.21;
import "@openzeppelin/contracts/ownership/Ownable.sol";

contract UsElectionVote is Ownable {
    mapping(string => address[]) candidateVotes;
    mapping(address => bool) alreadyVote;
    uint256 totalVotes;
    event Vote(address voter, string candidate);

    function vote(address _voter, string calldata _candidate) external {
        require(alreadyVote[_voter] == false, "the voter has already voted");
        require(
            keccak256(abi.encodePacked(_candidate)) ==
                keccak256(abi.encodePacked("Donald Trump")) ||
                keccak256(abi.encodePacked(_candidate)) ==
                keccak256(abi.encodePacked("Joe Biden")),
            "Invalid Candidate"
        );
        candidateVotes[_candidate].push(_voter);
        alreadyVote[_voter] = true;
        totalVotes++;
        emit Vote(_voter, _candidate);
    }

    function getVote(string calldata _candidate)
        external
        view
        returns (uint256)
    {
        return candidateVotes[_candidate].length;
    }

    function kill() public onlyOwner {
        selfdestruct(address(uint16(owner())));
    }
}
```

test/UsElectionVote.js

```
const UsElectionVote = artifacts.require('UsElectionVote');

contract('UsElectionVote', (accounts) => {
  let [phong, thao] = accounts;
  let contractInstance;
  beforeEach(async () => {
    contractInstance = await UsElectionVote.new();
  });
  it('should vote for Biden', async () => {
    const result = await contractInstance.vote(phong, 'Joe Biden', {
      from: phong
    });
    assert.equal(result.receipt.status, true);
    const bidenVote = await contractInstance.getVote('Joe Biden', {
      from: phong
    });
    assert.equal(bidenVote, 1);
  });
  afterEach(async () => {
    await contractInstance.kill();
  });
});
```

## Server

index.ts

```ts
import 'reflect-metadata';
import dotenv from 'dotenv';
import express, { Response, Request, NextFunction } from 'express';
import cors from 'cors';      You, a month ago • Squashed commit of the following:
import { registerVoter } from './routers/registerVoters';
import { createConnection } from 'typeorm';
import path from 'path';
import { Voter } from './entities/Voter';
import { usElectionVote } from './routers/usElectionVote';

dotenv.config();

const app = express();

const start = async () => {
  const connection = await createConnection({
    type: 'postgres',
    url: process.env.DB_URL,
    logging: true,
    synchronize: false,
    migrations: [path.join(__dirname, './migrations/*')],
    entities: [Voter]
  });
  await connection.runMigrations();
  app.use(express.json());
  app.use(cors());
  registerVoter(app);
  usElectionVote(app);
  app.use((err: Error, _req: Request, res: Response, _next: NextFunction) => {
    console.error(err.stack);
    res.status(500).send(err.message);
  });
  app.listen(5000);
};

start();
```

entities/Voter.ts

```ts
import { BaseEntity, Column, Entity, PrimaryGeneratedColumn } from 'typeorm';


@Entity()
export class Voter extends BaseEntity {
  @PrimaryGeneratedColumn('uuid')
  id: string;
  @Column()
  socialNumber: string;
  @Column()
  address: string;
  @Column()
  privateKey: string;
}
```

routers/RegisterVoter.ts

```
You, a month ago | 1 author (You)
import { Request, Response, Express } from 'express';
import crypto from 'crypto';
import { Voter } from '../entities/Voter';
import { TxObj } from '../types/TxObj';
import { createTransaction } from '../utils/createTransaction';
import { signTransaction } from '../utils/signTransaction';
import { web3Instance } from '../utils/web3Instance';
const { web3, voterContract } = web3Instance();
export const registerVoter = (app: Express) => {
  app.post('/register-voter', async (req: Request, res: Response) => {
    let confirmNum = 0;
    const { socialNumber } = req.body;
    const existingSocialNumber = await Voter.findOne({ socialNumber });
    if (existingSocialNumber) {
      return res
        .status(400)
        .send({ error: 'The social number is already registered' });
    }
    const account = web3.eth.accounts.create();
    console.log('address: ', account.address);
    console.log('privateKey: ', account.privateKey);
    const data = await voterContract.methods.registerVoter(
      account.address,
      socialNumber
    );
    const txObj: TxObj = {
      from: process.env.ADMIN_ADDRESS,
      data: data,
      to: process.env.VOTER_SC_ADDRESS,      You, a month ago • Squashed commit of the following:
      value: 1000000000000000
    };
    const adminPrivateKey = process.env.ADMIN_PRIVATE_KEY;
    let tx = await createTransaction(txObj);
    let signedTx = await signTransaction(tx, adminPrivateKey);
    return await web3.eth
      .sendSignedTransaction(signedTx)
      .on('transactionHash', (txHash) => {
        console.log(txHash);
      })
      .on('confirmation', async (confirmationNumber, receipt) => {
        confirmNum++;
        if (confirmNum === 2) {
          if (!receipt.status) {
            res.status(400).send({ error: 'Transaction failed' });
          } else {
            const cipher = crypto.createCipher(
              'aes-128-cbc',
              process.env.ENCRYPTED_KEY
            );
            let ciphertext = cipher.update(
              account.privateKey,
              'utf8',
              'base64'
            );
            ciphertext += cipher.final('base64');
            await Voter.create({
              socialNumber,
              address: account.address,
              privateKey: ciphertext
            }).save();
            console.log('Send Request');
            res.send({
              confirmationNumber,
              status: receipt.status,
              message: 'Account has been registered'
            });
          }
        }
      })
      .on('error', async (error) => {
        console.error(error.stack);
      });
  });
};

```

routers/usElectionVote.ts

```ts
import { Request, Response, Express } from 'express';
import { Voter } from '../entities/Voter';
import { TxObj } from '../types/TxObj';
import { createTransaction } from '../utils/createTransaction';
import { signTransaction } from '../utils/signTransaction';
import { web3Instance } from '../utils/web3Instance';
import crypto from 'crypto';

const { web3, usElectionVoteContract } = web3Instance();
export const usElectionVote = (app: Express) => {
  app.post('/us-election-vote', async (req: Request, res: Response) => {
    let confirmNum = 0;
    const { candidate, socialNumber } = req.body;
    const voter = await Voter.findOne({ socialNumber });
    if (!voter || !voter.address) {
      res
        .status(400)
        .send({ error: 'The voter does not exist or not yet registered' });
    } else {
      const data = await usElectionVoteContract.methods.vote(
        voter.address,
        candidate
      );
      const txObj: TxObj = {
        from: voter.address,
        data: data,
        to: process.env.US_ELECTION_VOTE_SC_ADDRESS
      };
      const encryptedPrivateKey = voter.privateKey;
      const decipher = crypto.createDecipher(
        'aes-128-cbc',
        process.env.ENCRYPTED_KEY
      );
      let privateKey = decipher.update(encryptedPrivateKey, 'base64', 'utf8');
      privateKey += decipher.final('utf8');
      console.log('private key: ', privateKey);
      let tx = await createTransaction(txObj);
      let signedTx = await signTransaction(tx, privateKey);
      await web3.eth
        .sendSignedTransaction(signedTx)
        .on('transactionHash', (txHash) => {
          console.log(txHash);
        })
        .on('confirmation', async (confirmationNumber, receipt) => {
          confirmNum++;
          if (confirmNum === 2) {
            if (!receipt.status) {
              res.status(400).send({ error: 'You have already voted' });
            } else {
              res.send({
                confirmationNumber,
                status: receipt.status,
                message: `You have voted for ${candidate}`
              });
            }
          }
        })
        .on('error', async (error) => {
          console.error(error.stack);
        });
    }
  });
  app.get('/vote-result', async (_req, res: Response) => {      You, a month ago • Squashed commit of the following:
    const joeBidenVote = await usElectionVoteContract.methods
      .getVote('Joe Biden')
      .call();
    const donaldTrumpVote = await usElectionVoteContract.methods
      .getVote('Donald Trump')
      .call();
    res.send({ joeBidenVote, donaldTrumpVote });
  });
};

```

types/TxObj.ts

```ts
export interface TxObj {
  from: string;
  data: any;
  to: string;
  value?: number;
  nonce?: string;
  (property) TxObj.gasLimit?: string | undefined
  gasLimit?: string;
}

```

types/env.d.ts

```ts
declare namespace NodeJS {

  export interface ProcessEnv {
    FULL_NODE_URL: string;
    VOTER_SC_ADDRESS: string;
    US_ELECTION_VOTE_SC_ADDRESS: string;
    ADMIN_PRIVATE_KEY: string;
    ADMIN_ADDRESS: string;
    DB_URL: string;
    ENCRYPTED_KEY: string;      You, a month ago • Squashe
  }
}

```

utils/createTransaction.ts

```ts
import { TxObj } from '../types/TxObj';
import { web3Instance } from './web3Instance';

const { web3 } = web3Instance();
export const createTransaction = async (txObj: TxObj) => {
  const txnCount = await web3.eth.getTransactionCount(txObj.from, 'pending');
  txObj.nonce = txObj.nonce ? txObj.nonce : web3.utils.numberToHex(txnCount);

  if (!txObj.gasPrice) txObj.gasPrice = await web3.eth.getGasPrice();

  txObj.gasLimit = web3.utils.numberToHex(300000);
  txObj.gasPrice = txObj.gasPrice
    ? web3.utils.numberToHex(txObj.gasPrice)
    : undefined;
  txObj.data = txObj.data.encodeABI();
  return txObj;
};

```

utils/signTransaction.ts

```ts
import { TxObj } from '../types/TxObj';
import { web3Instance } from './web3Instance';
import * as EthereumTx from 'ethereumjs-tx';

const { web3 } = web3Instance();
export const signTransaction = async (txObj: TxObj, privateKey: string) => {
  try {      You, a month ago • Squashed commit of the following:
    const chain = await web3.eth.net.getId();
    const tx = new EthereumTx.Transaction(txObj, { chain: chain });
    const key = Buffer.from(
      privateKey.length === 64 ? privateKey : privateKey.substring(2, 66),
      'hex'
    );
    tx.sign(key);
    return '0x' + tx.serialize().toString('hex');
  } catch (e) {
    throw e;
  }
};

```

utils/web3Instance.ts

```ts
import Web3 from 'web3';      You, a month ago • Squashed commit of the following:
import Voter from '../abis/Voter.json';
import UsElectionVoteContract from '../abis/UsElectionVote.json';
import dotenv from 'dotenv';
dotenv.config();

export const web3Instance = () => {
  const web3 = new Web3(
    new Web3.providers.HttpProvider(process.env.FULL_NODE_URL)
  );
  const voterAbi = Voter.abi as any;
  const usElectionVoteAbi = UsElectionVoteContract.abi as any;
  const voterContract = new web3.eth.Contract(
    voterAbi,
    process.env.VOTER_SC_ADDRESS
  );
  const usElectionVoteContract = new web3.eth.Contract(
    usElectionVoteAbi,
    process.env.US_ELECTION_VOTE_SC_ADDRESS
  );
  return { web3, voterContract, usElectionVoteContract };
};

```

.env

```
FULL_NODE_URL=https://rinkeby.infura.io/v3/b6ea1cd9d4a64650a661639e777e6919
VOTER_SC_ADDRESS=0xA9FdDeBfA9C160e013BFdE473E83b2A6292eE98d
US_ELECTION_VOTE_SC_ADDRESS=0x761D1a23484f32D36b7038Fdf6BA46b5247Ab255      You, a month ago • Squashed commit of the following:
ADMIN_PRIVATE_KEY=e75593c07554c41cc48971a4454dcaf21da3616ad55c3d3e8747f8acd0715e19
ADMIN_ADDRESS=0xbc5aC9e4bEe4aAE9F0D97F27d9e81B3eBDC8a39a
DB_URL=postgresql://postgres:2606@localhost:5432/thesis-project
ENCRYPTED_KEY=FoCKvdLslUuB4y3EZlKate7XGottHski1LmyqJHvUhs=
```

ormconfig.json

```
You, a month ago | 1 author (You)
{       You, a month ago • Squashed commit of the following:
    "type": "postgres",
    "url": "postgresql://postgres:2606@localhost:5432/thesis-project",
    "logging": true,
    "synchronize": false,
    "migrations": ["dist/migrations/*.js"],
    "entities": ["dist/entities/*.js"]
}
```

package.json

```
{
  "name": "server",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "dependencies": {
    "body-parser": "^1.19.0",
    "concurrently": "^6.0.1",
    "cors": "^2.8.5",
    "dotenv": "^8.2.0",
    "ethereumjs-tx": "^2.1.2",
    "express": "^4.17.1",
    "pg": "^8.5.1",
    "reflect-metadata": "^0.1.13",
    "typeorm": "^0.2.31",
    "typescript": "^4.1.3",
    "web3": "^1.3.1"
  },
  "devDependencies": {
    "@types/cors": "^2.8.9",
    "@types/ethereumjs-tx": "^2.0.0",
    "@types/express": "^4.17.9",
    "@types/node": "^14.14.20",
    "nodemon": "^2.0.7"
  },
  ▷ Debug
  "scripts": {
    "build": "tsc",
    "watch": "tsc -w",
    "start": "node dist/index.js",
    "dev": "nodemon dist/index.js",
    "dev:build": "concurrently \"yarn watch\" \"yarn dev\""      You, an hour ago • updated readme
  }
}
```

tsconfig.json

```
{      You, a month ago • Squashed commit of the following:
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "lib": ["dom", "es6", "es2017", "esnext.asynciterable"],
    "skipLibCheck": true,
    "sourceMap": true,
    "outDir": "./dist",
    "moduleResolution": "node",
    "removeComments": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "strictFunctionTypes": true,
    "noImplicitThis": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "allowSyntheticDefaultImports": true,
    "esModuleInterop": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "resolveJsonModule": true,
    "baseUrl": "."
  },
  "exclude": ["node_modules"],
  "include": ["./src/**/*.tsx", "./src/**/*.ts"]
}
```

## Client

index.tsx

```tsx
import React from 'react';      You, a month ago • Squashed commit of the following:
import 'react-toastify/dist/ReactToastify.css';
import ReactDOM from 'react-dom';
import App from './App';
import { ChakraProvider, extendTheme } from '@chakra-ui/react';
import reportWebVitals from './reportWebVitals';
const colors = {
  brand: {
    900: '#1a365d',
    800: '#153e75',
    700: '#2a69ac'
  }
};
const theme = extendTheme({ colors });
ReactDOM.render(
  <React.StrictMode>
    <ChakraProvider theme={theme}>
      <App />
    </ChakraProvider>
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();

```

App.tsx

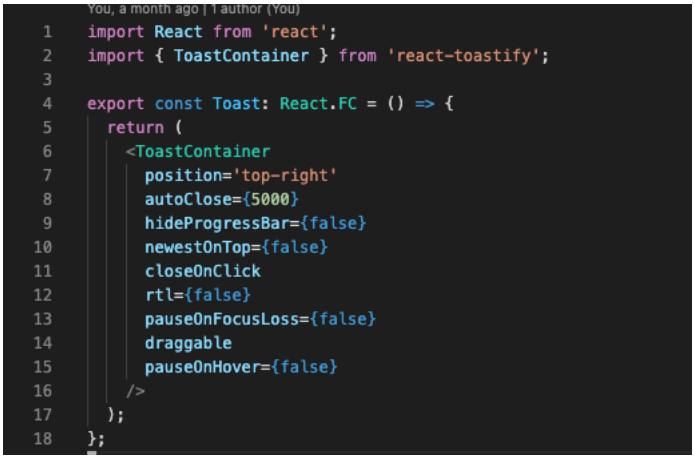
```tsx
import React, { useState, useEffect } from 'react';
import axios from 'axios';
import { VoteForm } from './components/VoteForm';
import { IVoteResultResponse } from './types';
import { Result } from './components/Result';
import { Layout } from './components/Layout';
import { Flex } from '@chakra-ui/layout';

const App: React.FC = () => {
  const [donaldTrumpVote, setDonaldTrumptVote] = useState<number | null>(null);
  const [joeBidenVote, setJoeBidenVote] = useState<number | null>(null);
  const [voted, setVoted] = useState(false);
  useEffect(() => {
    axios
      .get<IVoteResultResponse>('http://localhost:5000/vote-result')
      .then(({ data: { joeBidenVote, donaldTrumpVote } }) => {
        setDonaldTrumptVote(donaldTrumpVote);
        setJoeBidenVote(joeBidenVote);
      });
  }, [voted]);
  return (
    <Layout>
      <VoteForm voted={voted} setVoted={setVoted} />
      <Flex>
        <Result
          voteNumber={joeBidenVote}
          imageSrc='joe-biden.jpeg'
          imageAlt='Joe Biden'
          candidateHeading='Joe Biden'
        />
        <Result
          voteNumber={donaldTrumpVote}
          imageSrc='donald-trump.jpeg'
          imageAlt='Donald Trump'
          candidateHeading='Donald Trump'
        />
      </Flex>
    </Layout>
  );      You, a month ago • Squashed commit of the following:
};

export default App;

```

components/common/Toast.tsx



```tsx
import React from 'react';
import { ToastContainer } from 'react-toastify';

export const Toast: React.FC = () => {
  return (
    <ToastContainer
      position='top-right'
      autoClose={5000}
      hideProgressBar={false}
      newestOnTop={false}
      closeOnClick
      rtl={false}
      pauseOnFocusLoss={false}
      draggable
      pauseOnHover={false}
    />
  );
};
```

components/Layout.tsx

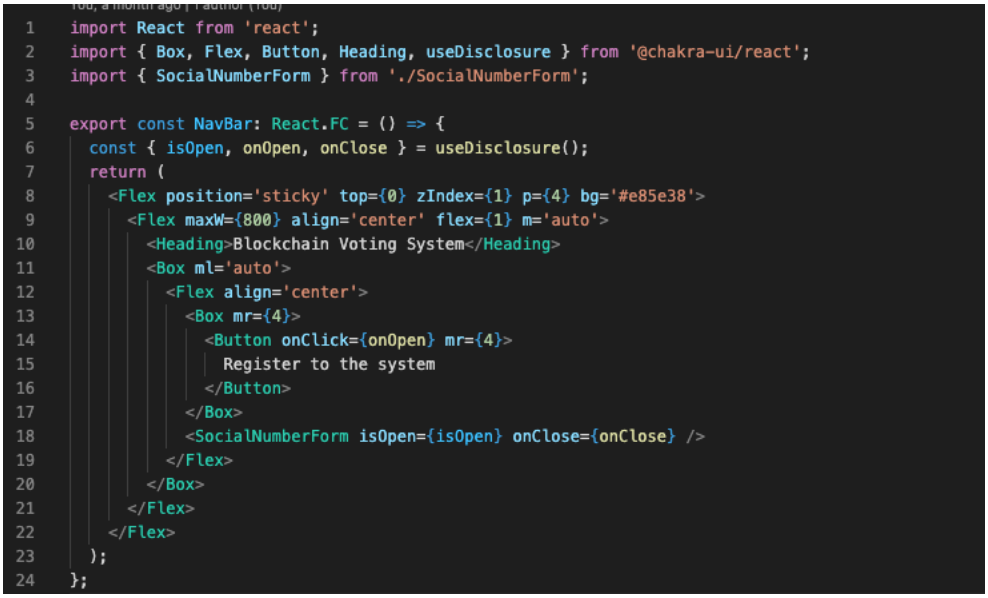
```tsx
import React from 'react';
import { NavBar } from './NavBar';
import { WrapperVariant, Wrappper } from './Wrapper';


interface LayoutProps {
  variant?: WrapperVariant;
}

export const Layout: React.FC<LayoutProps> = ({ variant, children }) => {
  return (
    <>
      <NavBar />
      <Wrappper variant={variant}>{children}</Wrappper>
    </>
  );
};

```

components/NavBar.tsx



```tsx
import React from 'react';
import { Box, Flex, Button, Heading, useDisclosure } from '@chakra-ui/react';
import { SocialNumberForm } from './SocialNumberForm';

export const NavBar: React.FC = () => {
  const { isOpen, onOpen, onClose } = useDisclosure();
  return (
    <Flex position='sticky' top={0} zIndex={1} p={4} bg='#e85e38'>
      <Flex maxW={800} align='center' flex={1} m='auto'>
        <Heading>Blockchain Voting System</Heading>
        <Box ml='auto'>
          <Flex align='center'>
            <Box mr={4}>
              <Button onClick={onOpen} mr={4}>
                Register to the system
              </Button>
            </Box>
            <SocialNumberForm isOpen={isOpen} onClose={onClose} />
          </Flex>
        </Box>
      </Flex>
    </Flex>
  );
};
```

components/NavBar.tsx

```tsx
import { Box, Image, Heading } from '@chakra-ui/react';
import React from 'react';
import { IResult } from '../types';

const Result: React.FC<IResult> = ({
  imageSrc,
  imageAlt,
  voteNumber,
  candidateHeading
}) => {
  return (
    <>
      <Box maxW='sm' borderWidth='1px' borderRadius='lg' overflow='hidden'>
        <Image boxSize='300px' src={imageSrc} alt={imageAlt} />
        <Heading as='h4' size='md' ml='8'>
          {candidateHeading}
        </Heading>
        <Box p='6'>
          <Box d='flex' alignItems='baseline'>
            <Box letterSpacing='wide' fontSize='md' ml='2'>
              Total Vote: {voteNumber}
            </Box>      You, a month ago • Squashed commit of the following:
          </Box>
        </Box>
      </Box>
    </>
  );
};

export { Result };
```

components/SocialNumberForm.tsx

```tsx
import React, { SyntheticEvent, useState } from 'react';

import axios from 'axios';
import {
  Button,
  FormControl,
  FormLabel,
  Input,
  Modal,
  ModalBody,
  ModalCloseButton,
  ModalContent,
  ModalHeader,
  ModalOverlay
} from '@chakra-ui/react';
import { toastSuccess } from '../utils/toastSuccess';
import { toastError } from '../utils/toastError';
import { Toast } from './common/Toast';

You, a month ago | 1 author (You)
interface ModalProps {
  isOpen: boolean;
  onClose: () => void;
}

const SocialNumberForm: React.FC<ModalProps> = ({ isOpen, onClose }) => {
  const [socialNumber, setSocialNumber] = useState<string>('');
  const [loading, setLoading] = useState<boolean>(false);
  const onSubmit = async (e: SyntheticEvent) => {
    console.log('submitting');
    e.preventDefault();
    try {
      setLoading(true);
      await axios.post('http://localhost:5000/register-voter', {
        socialNumber
      });
      toastSuccess(`${socialNumber} has been registered`);
    } catch (e) {
      console.log(e.response.data.error);
      toastError(e.response.data.error);
    }
    setSocialNumber('');
    setLoading(false);
    onClose();
  };      You, a month ago • Squashed commit of the following:
  return (
    <>
      <Modal isOpen={isOpen} onClose={onClose}>
        <ModalOverlay />
        <ModalContent>
          <ModalHeader>Register to voting system</ModalHeader>
          <ModalCloseButton />
          <ModalBody>
            <form onSubmit={onSubmit}>
              <FormControl>
                <FormLabel>Social Number: </FormLabel>
                <Input
                  value={socialNumber}
                  onChange={(e) => {
                    setSocialNumber(e.target.value);
                  }}
                />
                <Button isLoading={loading} mt={4} type='submit'>
                  Submit
                </Button>
              </FormControl>
            </form>
          </ModalBody>
        </ModalContent>
      </Modal>
      <Toast />
    </>
  );
};

export { SocialNumberForm };
```

components/VoteForm.tsx

```tsx
import { Button } from '@chakra-ui/button';
import axios from 'axios';
import { FormControl, FormLabel } from '@chakra-ui/form-control';
import { Input } from '@chakra-ui/input';
import { Heading } from '@chakra-ui/layout';
import { Select } from '@chakra-ui/select';
import React, { SyntheticEvent, useState } from 'react';
import { toastSuccess } from '../utils/toastSuccess';
import { toastError } from '../utils/toastError';

You, a month ago | 1 author (You)
interface IProps {
  setVoted: (voted: boolean) => void;
  voted: boolean;
}

const VoteForm: React.FC<IProps> = ({ setVoted, voted }) => {
  const [socialNumber, setSocialNumber] = useState('');
  const [candidate, setCandidate] = useState('');
  const [loading, setLoading] = useState(false);
  const onSubmit = async (e: SyntheticEvent) => {
    e.preventDefault();
    if (!candidate) {
      console.log('Must select a candidate');
    }
    try {
      setLoading(true);
      await axios.post('http://localhost:5000/us-election-vote', {
        socialNumber,
        candidate
      });
      toastSuccess(`${socialNumber} has voted for ${candidate}`);
      setVoted(!voted);
    } catch (e) {
      console.log(e.response.data.error);
      toastError(e.response.data.error);
    }
    setSocialNumber('');
    setLoading(false);
  };
  return (      You, a month ago • Squashed commit of the following:
    <>
      <Heading as='h3' size='lg'>
        Here is the US Election vote, choose your candidate
      </Heading>
      <form onSubmit={onSubmit}>
        <FormControl mt={4}>
          <FormLabel>Your Social Number</FormLabel>
          <Input
            value={socialNumber}
            onChange={(e) => setSocialNumber(e.target.value)}
          />
        </FormControl>
        <FormControl>
          <FormLabel mt={4}>Choose one</FormLabel>
          <Select
            defaultValue=''
            onChange={(e) => setCandidate(e.target.value)}
          >
            <option value=''></option>
            <option value='Joe Biden'>Joe Biden</option>
            <option value='Donald Trump'>Donald Trumpt</option>
          </Select>
        </FormControl>
        <Button
          colorScheme='yellow'
          type='submit'
          isLoading={loading}
          mt={4}
          mb={4}
        >
          Vote
        </Button>
      </form>
    </>
  );
};

export { VoteForm };
```

components/Wrapper.tsx

```tsx
import { Box } from '@chakra-ui/react';

export type WrapperVariant = 'small' | 'regular';

You, a month ago | 1 author (You)
interface WrapperProps {
  variant?: WrapperVariant;
}

export const Wrappper: React.FC<WrapperProps> = ({
  children,
  variant = 'regular'
}) => {
  return (
    <Box
      maxW={variant === 'regular' ? '800px' : '400px'}
      w='100%'
      mt={8}
      mx='auto'
    >
      {children}      You, a month ago • Squashed commit of the following:
    </Box>
  );
};
```

utils/toastError.ts

```ts
import { toast } from 'react-toastify';

export const toastError = (message: string) => {
  toast.error(message, {
    position: 'top-right',
    autoClose: 5000,
    hideProgressBar: false,
    closeOnClick: true,
    pauseOnHover: false,
    draggable: true,
    progress: undefined
  });
};

```

utils/toastSuccess.ts

```ts
import { toast } from 'react-toastify';

export const toastSuccess = (message: string) => {
  toast.success(message, {
    position: 'top-right',
    autoClose: 5000,
    hideProgressBar: false,
    closeOnClick: true,
    pauseOnHover: false,
    draggable: true,
    progress: undefined
  });
};

```

types.ts

```ts
export interface IResult {
  imageSrc: string;
  imageAlt: string;
  voteNumber: number | null;
  candidateHeading: string;
}


export interface IVoteResultResponse {
  joeBidenVote: number | null;
  donaldTrumpVote: number | null;
}

```

package.json

```
{
  "name": "client",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@chakra-ui/react": "^1.3.4",      You, a month ago • Squashed commit of
    "@emotion/react": "^11.1.5",
    "@emotion/styled": "^11.1.5",
    "@testing-library/jest-dom": "^5.11.4",
    "@testing-library/react": "^11.1.0",
    "@testing-library/user-event": "^12.1.10",
    "@types/jest": "^26.0.15",
    "@types/node": "^12.0.0",
    "@types/react": "^17.0.0",
    "@types/react-dom": "^17.0.0",
    "axios": "^0.21.1",
    "framer-motion": "^3.10.0",
    "react": "^17.0.1",
    "react-dom": "^17.0.1",
    "react-scripts": "4.0.3",
    "react-toastify": "^7.0.3",
    "typescript": "^4.1.2",
    "web-vitals": "^1.0.1"
  },
  Debug
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

tsconfig.json

```
{      You, a month ago • Squashed commit of the following:
  "compilerOptions": {
    "target": "es5",
    "lib": [
      "dom",
      "dom.iterable",
      "esnext"
    ],
    "allowJs": true,
    "skipLibCheck": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "noFallthroughCasesInSwitch": true,
    "module": "esnext",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react-jsx"
  },
  "include": [
    "src"
  ]
}
```