Ethocracy - A decentralised blockchain based voting system

Technical Guide

Submitted:

07/05/2021

Contributors:

Bartlomiej Kiraga - 17327333

Rónán Mac Gabhann - 17478562

Supervisor:

Dr. Geoffrey Hamilton

Table of Contents

Section 1: Motivation

Ethocracy was built to redesign the age-old process of voting in modern democratic societies to allow for secure, verifiable, and anonymous online voting. Estimates from MIT put the yearly administrative costs of US elections at $2 billion a year[1]. Voting requires securing and staffing numerous locations across any nation, printing and transporting millions of paper ballots, and manually counting said ballots by hand. These are costs which could be removed or heavily negated through the implementation of an online voting system.

Major issues with online voting systems are security in the form of guaranteeing voter anonymity, ensuring votes are counted correctly, and a lack of public visibility and transparency in the time-consuming tallying process.

A centralised online voting platform may use a server that all voters send their votes to, but the host of the server would have the ability to view and tamper with the ballots, and if the server was to be compromised by a third party they would have similar power. The tallying process would also remain entirely server side, even more hidden from the public eye.

Ethocracy seeks to address these concerns as an alternative to a centralised online voting system. It is a decentralised application that utilises the potential of increasingly popular blockchain technology via the Ethereum network[2]. It aims to provide a decentralised, transparent, online election process while still maintaining standard election practices such as guaranteeing voters anonymity and allowing only registered voters to cast their ballots.

All relevant data is shared with the public on the Ethereum blockchain including the tallying protocol and the anonymous yet verifiable ballots. This full transparency allows anyone and everyone to ensure the election process was upheld to the proper standards.

---

[1] Mohr, Zach, et al. *How Much Are We Spending on Election Administration?* 2017, p. 2. https://electionlab.mit.edu/sites/default/files/2019-01/mohr_et_al_2017summary.pdf

[2] Ethereum is the community-run technology powering the cryptocurrency, ether (ETH) and thousands of decentralized applications https://ethereum.org/en/

Section 2: Research

2.1 Democratic Voting Systems Around the World

https://www.fairvote.org/research_electoralsystems_world

According to fairvote.org, the “Winner Takes All” (First Past the Post) and “Proportional Representation” (Single Transferable Vote / Instant Runoff) make up the majority of democratic systems in the world.

Our research into how these systems work is shown below.

First Past the Post

https://en.wikipedia.org/wiki/First-past-the-post_voting

First Past the Post uses a simple voting style where each voter submits a ballot with only one candidate. If any candidates achieve a majority of the votes they win the election.

Votes may end up being “wasted” using this system, any votes for losing candidates do not carry over to the voter’s next preferred candidate. This may influence voter’s decisions when casting their ballot.

Single Transferable Vote

https://www.electoral-reform.org.uk/voting-systems/types-of-voting-system/single-transferable-vote/

The Single Transferable Vote system is one of the most widespread systems across the English speaking world. It aims to achieve proportional representation by allowing voters to rank their candidates in order of preference. It reduces the number of wasted votes that come up when using the First Past the Post system.

Ireland uses the Single Transferable Vote system for:

- General Elections
- Presidential Elections
- European Elections
- Seanad Elections

2.2 Blockchains: Ethereum vs Alternatives

After deciding on the core idea of our blockchain application, we researched what blockchain solutions were out on the market, and which would be most appropriate for developing a cryptographically secure solution. There were several candidates and our research below outlines why we decided to use Ethereum for the application.

Bitcoin

- Bitcoin is the first blockchain many think of, but Bitcoin is non-turing complete and cannot an application nearly as complex as what we were building.
    - Developer info: https://developer.bitcoin.org/
    - Whitepaper: https://bitcoin.org/bitcoin.pdf

Cardano

- Cardano was a very promising candidate. Seemingly tailor made to handle applications such as ours, with a functional language designed for secure blockchain development, all features are peer reviewed and studied carefully before they're added.
    - Applications: https://cardano.org/enterprise/
    - Protocol: https://cardano.org/ouroboros#vision

- InputOutput is building a blockchain solution using Cardano to government issued ID cards, a testament to the security this platform plans to offer.
    - Enterprise Interest: https://iohk.io/en/enterprise/#digital-identity

- Unfortunately Cardano was, and still is, at an early stage of development and while the community has grown massively since our choice of blockchain, many features are still in the works.

Hyperledger

- Hyperledger is an advanced solution with a wide toolset and large amount of supporting documentation.
    - Website: https://www.hyperledger.org/

- Hyperledger does not have a main network, instead relying on individual, smaller blockchains. To ensure incorruptibility of our platform the larger, Ethereum network was more suitable.

Ethereum

- Ethereum is currently the largest blockchain network offering Turing complete decentralised application support. Its network offered a plethora and the largest developer community available.
    - Website: https://ethereum.org/en/
    - Whitepaper: https://ethereum.org/en/whitepaper/

While we decided on Ethereum for our application, we believe that Cardano may become even more suitable for this style of application given enough time to develop.

2.3 Votereum

https://ieeexplore.ieee.org/document/8713661

Other similar projects have tried developing decentralised blockchain voting systems built on Ethereum. These gave us assurance that the project was feasible.

2.4 Development Related Research

https://ethereum.org/en/developers/

https://docs.soliditylang.org/en/v0.8.0/

https://docs.ethhub.io/

As we developed our project, we researched into utilising additional tools and learning materials to implement into our project as we developed a clearer, applied design. Researching further into areas such as documentation and community developed projects helped build our understanding of blockchain applications.

Section 3: Design

3.1 Development Tools

Primarily, the application was written using React javascript framework together with Solidity, a language designed for writing smart contracts that run on Ethereum.

MetaMask is a Google Chrome and Mozilla Firefox browser extension required to allow any given user to send transactions containing data to the blockchain from the frontend interface.

Truffle is used to compile smart contracts and to simulate a blockchain network locally, instead of paying currency in the form of ETH to deploy it to the live Ethereum network.

Testing of JavaScript files was done through the Jest.js testing framework, testing of the Solidity smart contracts was done through truffle's testing framework based on javascript's mocha and chai libraries.

Below is a full list of the languages and development tools used:

- Solidity
    - Version 0.8.0
    - Used to write our Election and ElectionDeployer smart contracts which receive and handle voter ballots

- JavaScript
    - Version ES6
    - Used with React.js and Node.js as core libraries to develop the frontend of the application

- Jest.js
    - Version 26.x
    - Used to help test our JavaScript code

- Truffle
    - Version 5.1.65
    - Used to compile Solidity smart contracts and to simulate a blockchain network locally

- MetaMask
    - Version 9.4.0
    - Used by all users to interact with the blockchain

3.2 Dependencies

Ethocracy utilises an array of JavaScript libraries. A portion are dedicated to providing vital functionality such as Web3.js which connects the frontend and backend together, or Crypto.js which helps the application guarantee voter anonymity. Another portion assists in the styling of the user interface.

Below is a full list of dependencies:

The following libraries provide important functionality to the application:

- Web3.js
    - Version 1.2.2
    - Used to allow the frontend to interact with the Ethereum blockchain backend

- React.js
    - Version 16.11.0
    - Used as the core library for the user interface

- Node.js
    - Version 12.19.0

    - Used to host a server as an election catalogue

- SHA256
    - Version 0.2.0
    - Used to hash voter IDs to maintain voter anonymity

- Crypto.js
    - Version 4.0.0
    - Used to encrypt and decrypt votes at varying stages in the election process

- Node RSA
    - Version 1.1.1
    - Used to generate RSA keypairs

- React Router
    - Version 16.11.0
    - Used for instantaneous page navigation without refreshing the page or reloading previously loaded UI elements

The remaining libraries are used to assist in the styling of the user interface:

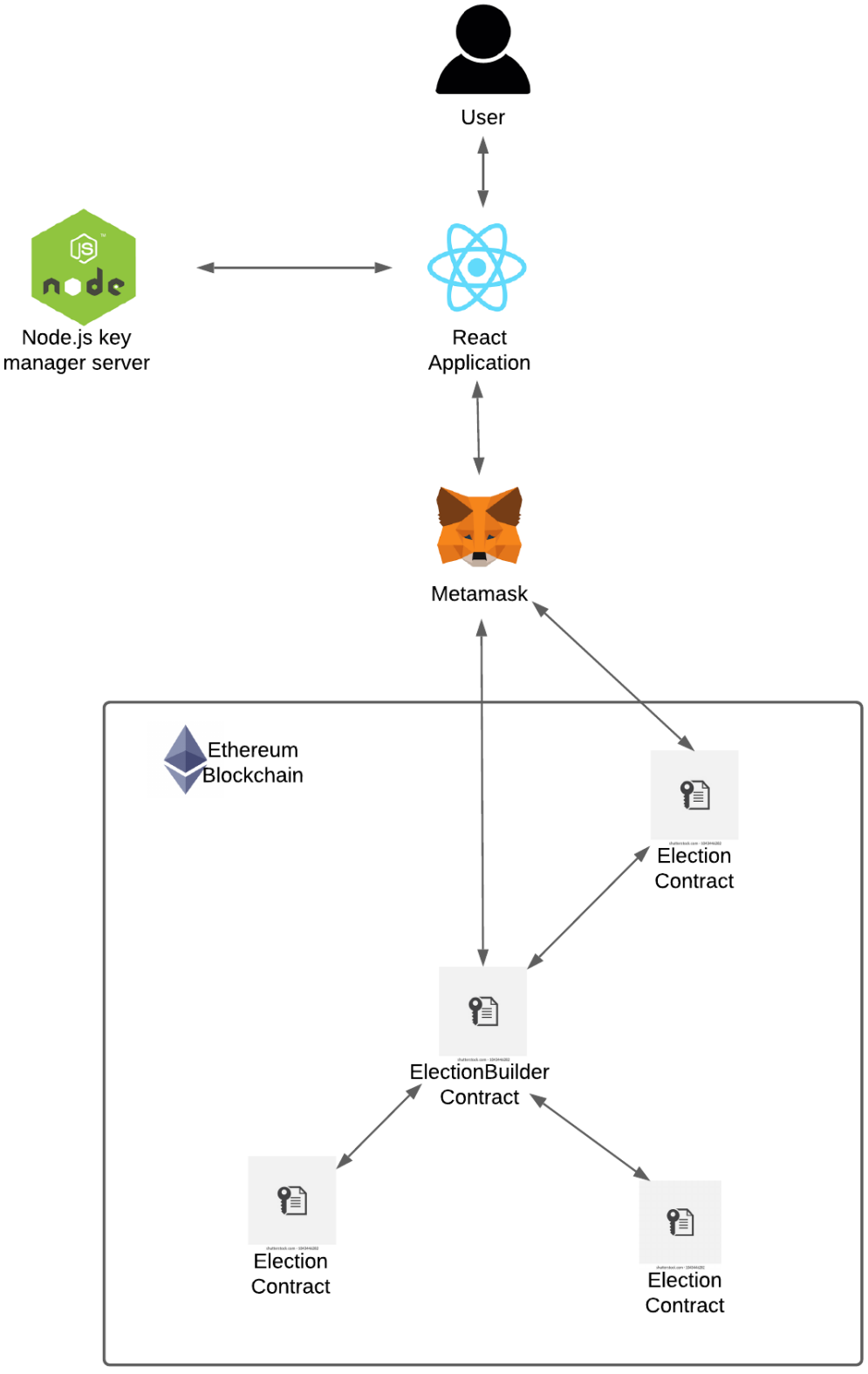
- React Bootstrap
    - Version 1.5.2
    - Used for styling the user interface

- Bootstrap
    - Version 4.6.0
    - Required by React Bootstrap for styling the user interface

- React Datepicker
    - Version 1.5.2
    - Used to style a discrete date selector for the election deadline

- Multiselect React Dropdown
    - Version 1.6.11
    - Used to style a discrete selection interface for use during a Single Transferable Vote ballot submission

3.3 System Architecture



The system architecture of our application differs from most standard web applications as it aims to maximise decentralisation and user transparency in its design.

Most web applications rely on the use of a central server and a database to manage data and authorise users. While this is adequate in the majority of applications, it has some disadvantages associated with using this approach in an election system. One of the main disadvantages is the fact that a centralised system requires users to place more trust in the system administrators. Another potential issue is that a centralised system can often be a target for attacks aimed at disrupting or tampering with the system.

Our application attempts to mitigate these faults by adopting a more decentralised design approach which replaces the traditional server+database backend with smart contracts deployed onto Ethereum blockchain. The main advantage of our design is that through its use of blockchain technology it achieves a greater degree of transparency in the elections. Every cast ballot is visible to every user and after the election deadline when the election result key is released anybody can fetch the ballots of the blockchain and calculate the winner themselves removing the need for a central authority tasked with the tally of votes.

The application is composed of the following components:

3.3.1 React Frontend Application

Software development on the ethereum blockchain has many downsides as the computations performed on it are not only limited by limited resources but they also cost money to execute. This means that most of the functionality of our application was implemented on the frontend making react javascript framework a perfect choice for our project. The react frontend contains not only the user interface but also the majority of the logic involved in deploying election, casting votes and tallying results. The react frontend then communicates with the smart contracts on the blockchain using the web3.js library which acts as a binding for Ethereum's JSON-RPC interface.

3.3.2 Ethereum Smart Contracts

The application relies on the use of Ethereum smart contracts written in Solidity. Solidity is a statically typed programming language designed for contract development for the Ethereum blockchain. The application uses two contracts, the ElectionBuilder and the Election contracts.
The ElectionBuilder contract is responsible for the creation and storage of The Election contracts as well as providing extra functionality such as the mapping of users to all their ballots and elections for easy access.
The Election contract contains all the information of a given election relevant to the voting, tallying and validation processes.

3.3.3 Metamask

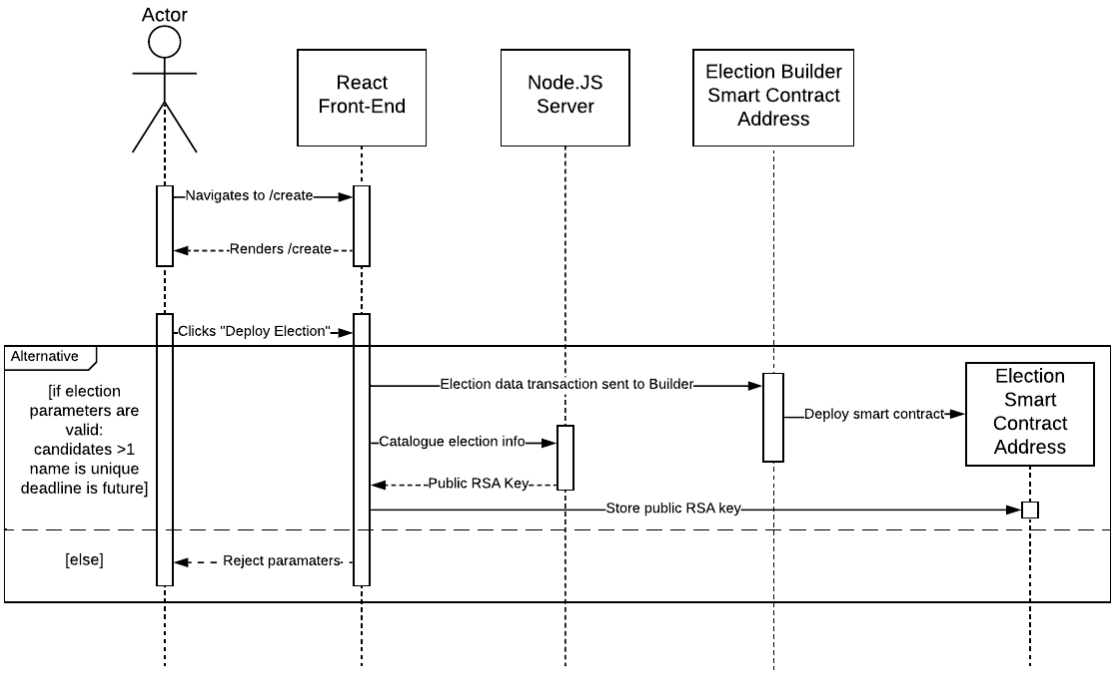
Metamask is a browser plugin that allows any users to connect to the Ethereum blockchain through their browser without running a full ethereum node. It also acts as a secure decentralised authentication system that lets our application identify users balled on the ethereum addresses they created using their metamask wallets.

3.3.4 Node.js key management server

The application makes use of a node js server which has limited functionality. Its purpose is to provide cryptographic key generation and storage functionality for the application. Performing cryptographic functions or the storage of keys on the blockchain is strongly discouraged due to the fact that by design all operations performed on the blockchain are publicly visible to everyone. As decentralised storage tools such as swarm are still in early experimental period of development we opted to sacrifice some decentralisation and use a node server to store and manage the election result keys.

3.4 High Level Design

3.4.1 Deploying an Election

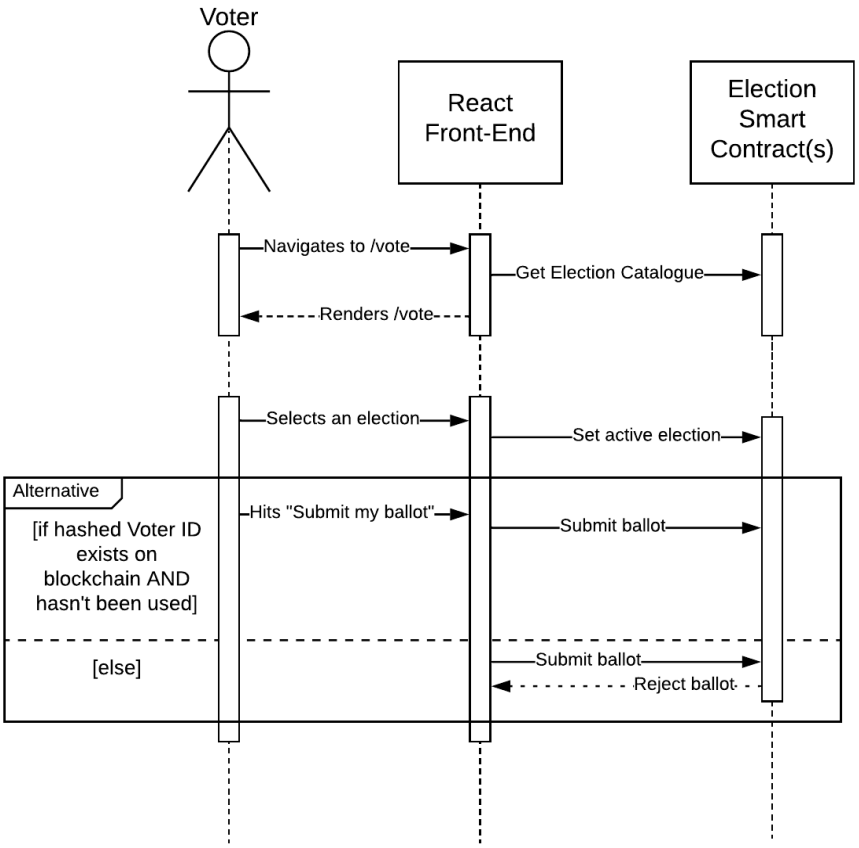


The above sequence diagram outlines the process for deploying an election.

After the User finishes setting election parameters in the frontend interface, MetaMask will send a transaction to the Ethereum Address where the Election Builder contract is stored. If the election parameters are valid, a new instance of an Election contract will be deployed with appropriate parameters such as hashed Voter IDs, candidates, deadline and election name. Cryptographic keys will also be generated. The public key will be sent to the new Election instance for the voter's to encrypt their votes.

This Election Instance will stay live on the blockchain forever (until the blockchain is terminated, in the case of a local test environment), but will only take votes until the deadline.

3.4.2 Cast a Ballot

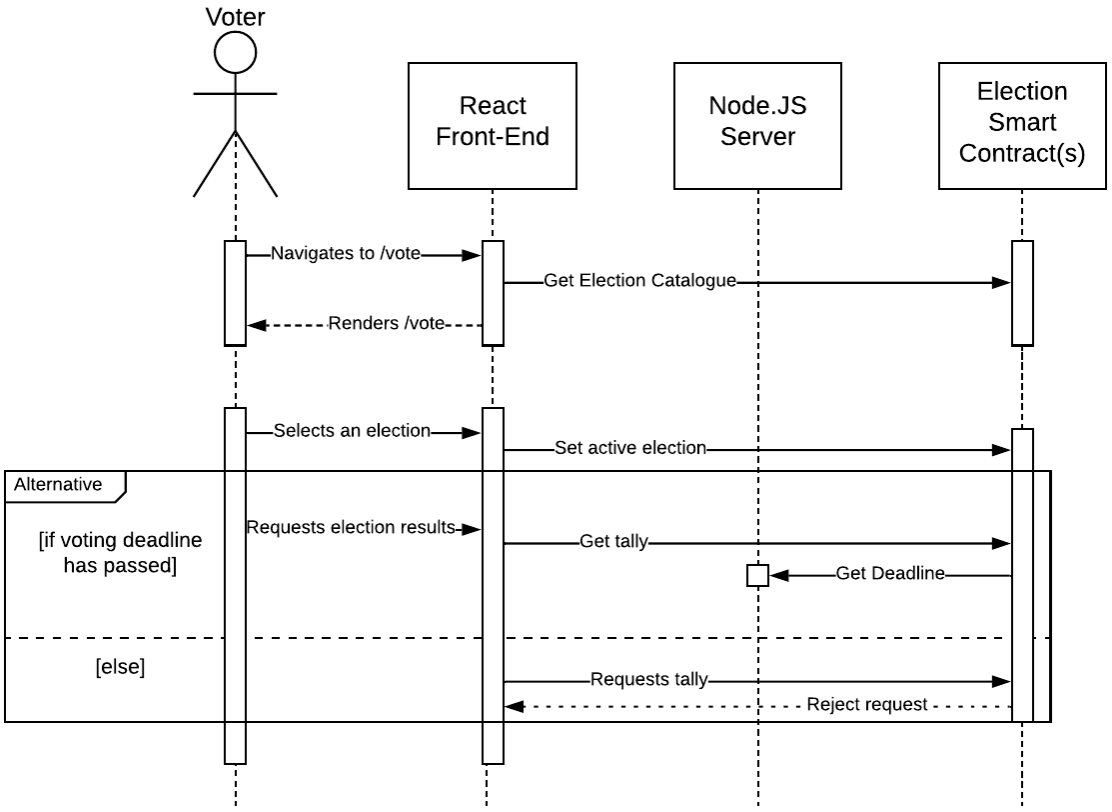


The above sequence diagram outlines the process for casting a ballot.

The Node server provides the user with an array of Election information in the form of a JSON file, the user can select any address in this catalogue and select it, setting it as their active election in the front end interface.

The interface will display all the election info and provide the user with the UI required to vote. Upon ballot submission to the Election Instance's blockchain address it will verify that the voter's hashed voter ID is valid and has not been used to vote before. Upon confirming this the ballot will then be stored on the Election Instance's address.

3.4.3 View Election Results



The above sequence diagram outlines the process for viewing the election results.

Any user can view the results and it does not require them to have been a voter in the election.

Requesting the results from the Election Instance will return a tally if the voting deadline has passed. If this is the first user to request the results, the votes will begin being counted. This process may take some time for large scale elections but should be relatively instant for smaller tallies.

Section 4: Implementation

4.1 Election Creation

```
28
29    await fetch(`/api/generateKeys?name=${encodeURIComponent(this.props.name)}&deadline=${selectedTimestamp}`)
30      .then(response => response.json())
31      .then(data => this.setState(() => {
32        return {
33          electionKey: data.public_key
34        }
35      }));
36      let hashedVoterIds = [];      You, 6 hours ago • reorganised project file structure, addded more tests,
37      for (let i = 0; i < this.props.validVoters.length; i++) {
38        let hash = hashVoterId(this.props.validVoters[i]);
39        hashedVoterIds.push(hash);
40      }
41      const validVoterCount = hashedVoterIds.length;
42      try {
43        await this.props.electionBuilder.methods.deployElection(this.props.name, this.props.candidates, time,
44      } catch (e) {
45        alert("Election not deployed");
46      } finally {
47        this.props.setName("");
48      }
```

Fig1

```
app.get('/api/generateKeys', (req, res) => {
  const electionName = req.query.name;
  const deadline = req.query.deadline;
  if (!electionName) {
    return res.send({
        error: "election name not given to the api"
    })
  }
  const electionKeys = utils.generateKeys();
  elections.addElection({name: electionName, key: electionKeys.private_key, deadline: deadline});

  res.send({
    public_key: electionKeys.public_key
  });
})
```

Fig2

```
const generateKeys = () => {
  const key = new NodeRsa({b:1024});
  const public_key = key.exportKey('public');
  const private_key = key.exportKey('private');
  return {public_key, private_key}
}
```

Fig3

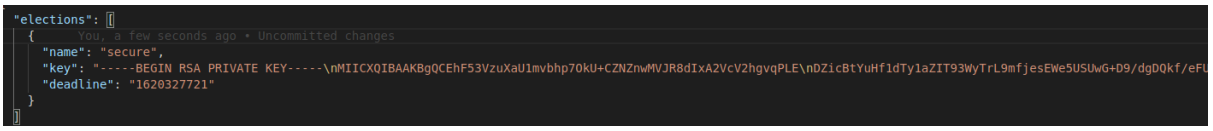


Fig4

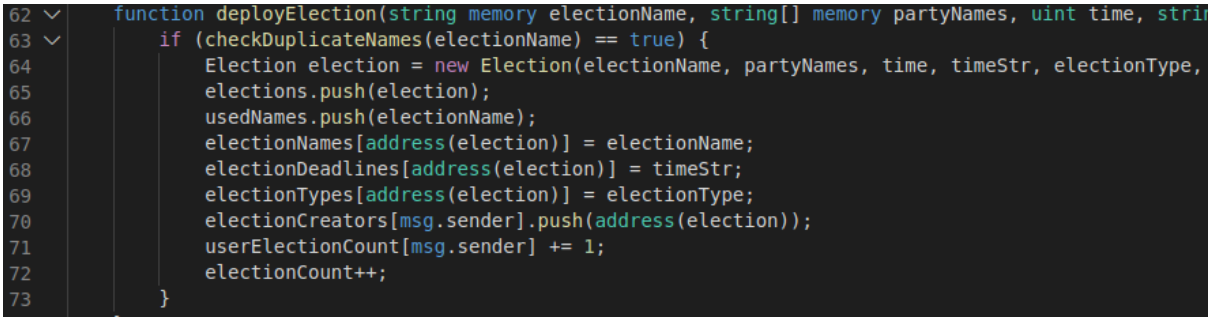


Fig5

In the Fig1 image above the implementation of the deploy election functionality can be seen. In lines 29-35 the express Node js api endpoint is called with the parameters that include the name of the election and its deadline measured as a UNIX timestamp.

The server then generates an RSA keypair and returns the public key back to the client (Fig2,Fig3). The private key is stored by the server in a JSON file as can be seen in Fig4.

In the lines 36-40 of Fig1 every member in the list of valid voter ids which was previously given by the election creator, is hashed using sha256 hashing algorithm. Since the list of valid ids will be deployed onto the blockchain with the election, it is hashed to prevent the public from discovering them and potentially using them to vote in place of someone else.

In line 43 of Fig1 the deployElection ElectionBuilder contract method is called using web3.js. Passed in as the arguments are all the election parameters required for an election. Fig5 shows the deployElection solidity method from the ElectionBuilder

contract. In line 64 onwards the new election contract is created and its information is stored on the ElectionBuilder contract.

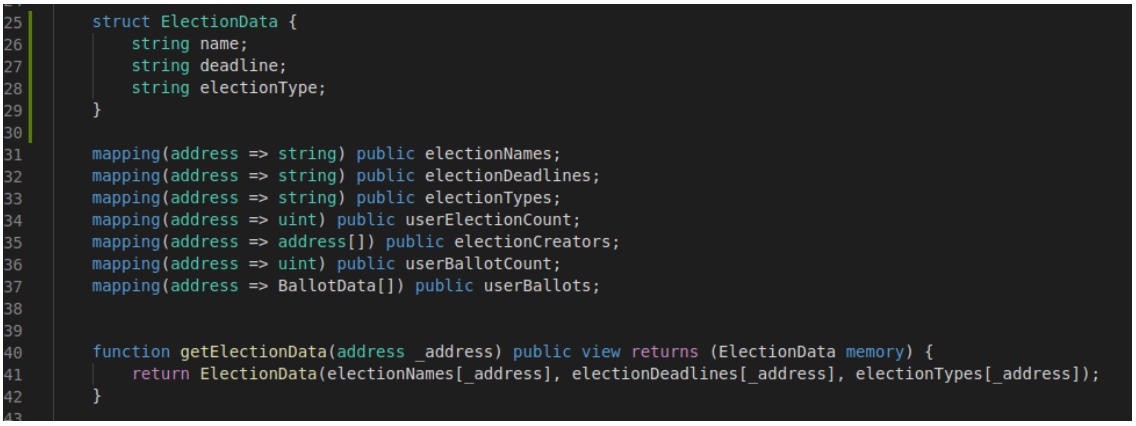
4.2 Election Selection

```
25    async getElections() {
26      let elections = [];
27      const electionCount = await this.props.electionBuilder.methods.electionCount().call();
28      for (let i = 0; i < electionCount; i++) {
29        const electionAddress = await this.props.electionBuilder.methods.elections(i).call();
30        const electionData = await this.props.electionBuilder.methods.getElectionData(electionAddress).call();
31        const election = {name: electionData.name, address: electionAddress, type: electionData.electionType, deadline: electionData.deadline};
32        elections.push(election);
33      }
34      this.setState(() => {
35        return {
36          elections: elections
37        }      You, 8 hours ago • reorganised project file structure, addded more tests, removed unnecessary files
38      })
39    }
```

Fig1

```
14    async handleSelectElection(e) {
15      e.preventDefault();
16      const address = e.target.elements.selectElection.value.trim();
17      let contract;
18      try {
19        contract = await new this.props.web3.eth.Contract(ElectionContract.abi, address);
20        this.setState(() => {
21          return {
22            electionExists: true
23          }
24        })
25      } catch(e) {
26        this.setState(() => {
27          return {
28            electionExists: false
29          }
30        })
31      }
32      if (this.state.electionExists === true) {
33        this.props.setContract(contract);
34        this.props.setSelectedElection(true);
35      } else {
36        this.props.setSelectedElection(false);
37      }
38    }
```

Fig2



```
25      struct ElectionData {
26          string name;
27          string deadline;
28          string electionType;
29      }
30
31      mapping(address => string) public electionNames;
32      mapping(address => string) public electionDeadlines;
33      mapping(address => string) public electionTypes;
34      mapping(address => uint) public userElectionCount;
35      mapping(address => address[]) public electionCreators;
36      mapping(address => uint) public userBallotCount;
37      mapping(address => BallotData[]) public userBallots;
38
39
40      function getElectionData(address _address) public view returns (ElectionData memory) {
41          return ElectionData(electionNames[_address], electionDeadlines[_address], electionTypes[_address]);
42      }
```

Fig3

The images above show the implementation of the select election functionality. In Fig1 the data of all deployed elections is fetched from the ElectionBuilder contract by calling getElectionData as well as calling the mapping getter methods from the ElectionBuilder contract. To reduce the number of calls, the ElectionBuilder implements a getElectionData function which returns a struct that bundles data found in the mappings into a single object.

Figure2 shows the function that runs whenever a user clicks the “Select Election” button from the user interface. The function takes the election address from the user input and in line19 uses the address along with the contract’s ABI to get an instance of the contract. The contract can then be accessed by the application.

4.3 Election Voting

```
componentDidMount = async () => {
  await this.getCandidates();
  const electionKey = await this.props.contract.methods.electionKey().call();
  const electionType = await this.props.contract.methods.electionType().call();
  this.setState(() => {
    return {
      electionKey: electionKey,
      electionType: electionType
    }
  })
}
```

Fig1

```
async handleCastVote() {
  if (this.state.electionType === "FPP") {
    await this.props.contract.methods.castVote(maskBallot(this.state.FPPballot[0].id, this.state.electionKey), "0x" + this.state.voterId).send({from: this.props.accou
  }
  else {
    let formatedSTVballot = "";
    for (let i = 0; i < this.state.STVballot.length; i++) {
      formatedSTVballot += this.state.STVballot[i].id + "|";
    }
    formatedSTVballot = formatedSTVballot.slice(0, -1);
    await this.props.contract.methods.castVote(maskBallot(formatedSTVballot, this.state.electionKey), "0x" + this.state.voterId).send({from: this.props.accounts[0]})
  }
}
```

Fig2

```
export const maskBallot = (ballot, public_key) => {
    const buffer = Buffer.from(ballot);
    const encrypted = crypto.publicEncrypt(public_key, buffer);
    return encrypted.toString("base64");
};

export const unmaskBallot = (ballot, private_key) => {
    let buffer = Buffer.from(ballot, "base64");
    let decrypted = crypto.privateDecrypt(private_key, buffer);
    return decrypted.toString("utf8");
};
```

Fig3

```
88      function castVote (string memory _vote, bytes memory _voterId) public {
89          require(block.timestamp - startTime < allowedTime);
90          require(!voters[msg.sender]);
91          require(!spentVoterIds[_voterId]);
92          require(validateUser(_voterId) == true);
93          ballots.push(_vote);
94          ballotCount++;
95          voters[msg.sender] = true;
96          spentVoterIds[_voterId] = true;
97          electionBuilder.addUserBallot(electionName, msg.sender, _vote);
98      }
```

Fig4

```
78      function validateUser (bytes memory _voterId) public view returns (bool) {
79          bytes32 hashedVoterId = sha256(_voterId);
80          for (uint i = 0; i < validVoterCount; i++) {
81              if (hashedVoterId == hashedVoterIds[i]) {
82                  return true;
83              }
84          }
85          return false;
86      }
```

Fig5

The above images show our implementation of the cast vote functionality. To begin with the user must select an election as outlined in the previous section. With that done, the application fetches the election key, election type and candidate data from the Election contract as can be seen in Fig1. In case the election is of an STV type, the ballots are formatted into a single string in order to simplify their storage on the contract.

The user can then select their candidate choice, encrypt their ballot using the public election key and pass their ballot along with their voter id to the castVote solidity contract method. (Fig2,Fig3).

The solidity castVote Election contract method then checks the timing of the ballot to make sure it is cast within the allowed time of the election(Fig4, line89).

In lines 90 and 91 of Fig4 the contract makes checks to ensure that voter id given by the user and their ethereum address were not previously used to cast a ballot in the election which prevents any possibility of being able to vote twice. In line 92 the validity of the voter id is checked by calling the validatUser method shown in Fig5, which hashes the id using sha256 hashing algorithm and compares it to the hashed list of valid voter ids which are provided by the election creator during the election deployment. If all the previously outlined checks pass the ballot is then accepted and stored on the Election Contract (line93).

The ballot is also added to the ElectionBuilder contract to allow users to fetch and display their ballot receipts without having to search for the election.

4.4 Election Results Tally

```
async getResultKey() {
  await fetch(`/api/getResultKey?name=${encodeURIComponent(this.state.electionName)}`)
  .then(response => response.json())
  .then(data => this.setState(() => {
    return {
      resultKey: data.resultKey
    }
  }));
}

async handleUnlockResults() {
  await this.getResultKey();
  const electionKey = await this.props.contract.methods.electionKey().call();
  const testCiphertext = maskBallot("teststring", electionKey);
  try {
    if (unmaskBallot(testCiphertext, this.state.resultKey) === "teststring") {
      await this.props.contract.methods.releaseResultKey(this.state.resultKey).send({from: this.props.accounts[0]});
    } else {
      console.log("Not a valid key");
    }
  } catch (e) {
    alert("Election still ongoing");
  }
}

async unmaskElectionBallots() {
  let unmaskedBallots = [];
  await this.getBallots();
  const resultKey = await this.props.contract.methods.resultKey().call();
  for (let i = 0; i < this.state.ballotCount; i++){
    let unmaskedBallot = unmaskBallot(this.state.ballots[i], resultKey);
    unmaskedBallots.push(unmaskedBallot);
  }
  this.setState(() => {
    return {
      unmaskedBallotList: unmaskedBallots
    }
  })
}
```

Fig1

```
app.get('/api/getResultKey', (req, res) => {
  const electionName = req.query.name;
  const electionData = elections.lookupElection(electionName);
  const deadline = electionData.deadline;
  const electionFinished = utils.checkElectionComplete(deadline);
  if (electionFinished == true) {
    return res.send({
      resultKey: electionData.key
    });
  } else {
    res.send({
      resultKey: "noKey"
    })
  }
})
```

Fig2

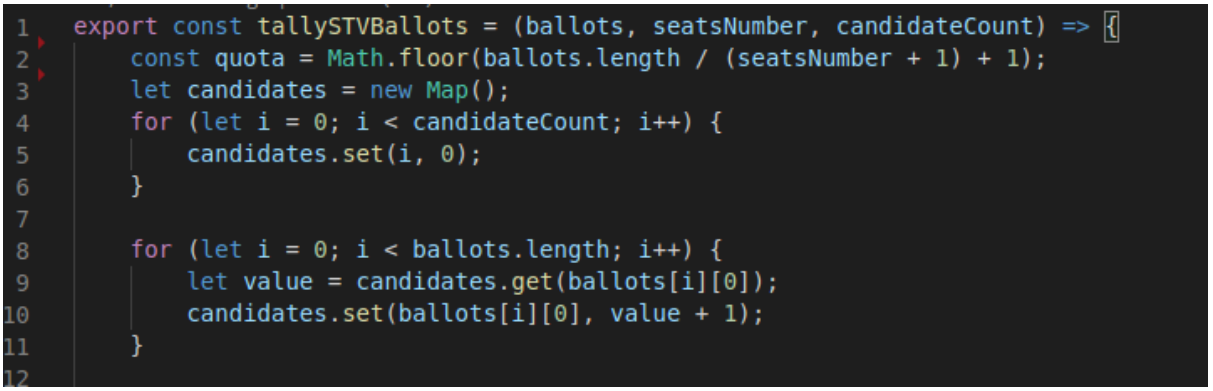
The tally ballot functionality implementation begins with fetching of the result key from the node js server as seen in image Fig1 lines 86-94. The server receives the request along with the name of the election after which it looks up the election data and checks whether the election deadline has passed. Only if the election deadline is over will it release the result key which will then be checked for validity and passed to the Election contract where it can be grabbed by anybody and used to decrypt the ballots.(Fig1, lines 96-108)

```
1   //FPP tally uses a modified selectionsort      You, 20 hours ago •
2
3   export const tallyFPPBallots = (candidates, ballots) => {
4       let resultMap = new Map();
5       for (let i = 0; i < candidates.length; i++) {
6         resultMap.set(candidates[i].name, 0);
7       }
8       for (let i = 0; i < ballots.length; i++) {
9         for (let [key, value] of resultMap.entries()) {
10          if (key === candidates[parseInt(ballots[i], 10)].name) {
11            resultMap.set(key, value+1);
12          }
13        }
14      }
15      let results = [];
16      for (let [key,value] of resultMap.entries()) {
17        results.push({candidate: key, votes: value});
18      }
19
20      for (let i = 0; i < results.length; i++) {
21        let min = i;
22        for (let j = i+1; j < results.length; j++) {
23          if (results[j].votes < results[min].votes){
24            min = j
25          }
26        }
27        if (min !== i) {
28          let tmp = results[i];
29          results[i] = results[min];
30          results[min] = tmp;
31        }
32      }
33      results = results.reverse();
34      return results
35  }
```

Fig3

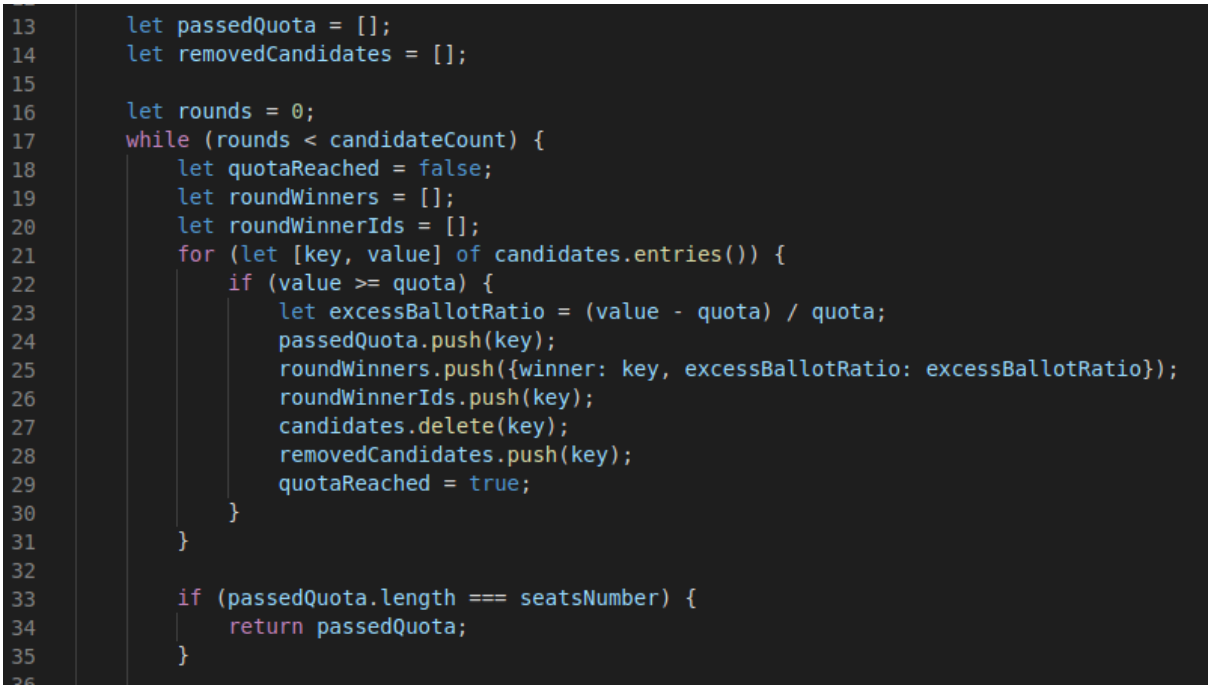
The tallying process differs depending on the type of election. Fig3 shows the implementation for tallying first past the post votes, it maps candidates to the number of

votes they received and then uses a selection sort algorithm to order them based on the number of votes they received.



```
export const tallySTVBallots = (ballots, seatsNumber, candidateCount) => {
    const quota = Math.floor(ballots.length / (seatsNumber + 1) + 1);
    let candidates = new Map();
    for (let i = 0; i < candidateCount; i++) {
        candidates.set(i, 0);
    }

    for (let i = 0; i < ballots.length; i++) {
        let value = candidates.get(ballots[i][0]);
        candidates.set(ballots[i][0], value + 1);
    }
```

Fig4

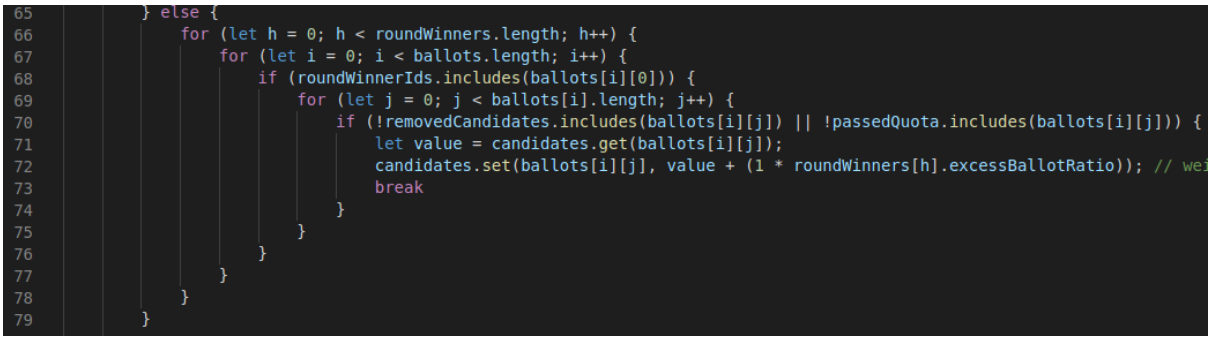


```
    let passedQuota = [];
    let removedCandidates = [];

    let rounds = 0;
    while (rounds < candidateCount) {
        let quotaReached = false;
        let roundWinners = [];
        let roundWinnerIds = [];
        for (let [key, value] of candidates.entries()) {
            if (value >= quota) {
                let excessBallotRatio = (value - quota) / quota;
                passedQuota.push(key);
                roundWinners.push({winner: key, excessBallotRatio: excessBallotRatio});
                roundWinnerIds.push(key);
                candidates.delete(key);
                removedCandidates.push(key);
                quotaReached = true;
            }
        }

        if (passedQuota.length === seatsNumber) {
            return passedQuota;
        }
```

Fig5

```
if (!quotaReached) {
    let lastCandidate;
    let smallestValue;
    for (let [key, value] of candidates.entries()) {
        if (candidates.get(key) < smallestValue || smallestValue === undefined) {
            smallestValue = candidates.get(key);
            lastCandidate = key;
        }
    }
    removedCandidates.push(lastCandidate);
    for (let i = 0; i < ballots.length; i++) {
        if (ballots[i][0] === lastCandidate){
            for (let j = 0; j < ballots[i].length; j++) {
                if (!removedCandidates.includes(ballots[i][j]) || !passedQuota.includes(ballots[i][j])) {
                    let value = candidates.get(ballots[i][j]);
                    candidates.set(ballots[i][j], value + 1);
                    break;
                }
            }
        }
    }
    for (let [key,value] of candidates.entries()) {
        if (value >= quota) {
            passedQuota.push(key);
        }
    }
    candidates.delete(lastCandidate);
```

Fig6



```
} else {
    for (let h = 0; h < roundWinners.length; h++) {
        for (let i = 0; i < ballots.length; i++) {
            if (roundWinnerIds.includes(ballots[i][0])) {
                for (let j = 0; j < ballots[i].length; j++) {
                    if (!removedCandidates.includes(ballots[i][j]) || !passedQuota.includes(ballots[i][j])) {
                        let value = candidates.get(ballots[i][j]);
                        candidates.set(ballots[i][j], value + (1 * roundWinners[h].excessBallotRatio)); // wei
                        break
                    }
                }
            }
        }
    }
}
```

Fig7

The implementation of the STV tallying process proved more complex to implement. The figures 5,6 and 7 show the main steps taken to implement it. The first step in counting STV votes is to count all the first preferences on the ballot just like in the FPP election type. After calculating the first choice ballot totals, they are compared to the election quota. Election quota is calculated by the formula (ballot count/(seats up for election + 1) +1). If the quota is passed the candidates are added to the passedQuota array and their ballot's second choice picks are distributed to other candidates, after being scaled down to be proportional to the excess ballots that went over the quota.(Fig7)

If the quota is not reached the lowest scoring candidate is removed from the election and the second choices on their ballots get distributed to other candidates. This process repeats until all the seats up for election are filled.

Section 5: Results

While we made some changes along the way, our application performs what we set out to build.

The application can deploy Election smart contracts to the Ethereum network or a blockchain, which can be interfaced with through the use of our front end client. The process of tallying is verifiable with all the data publicly available on the blockchain, and voters are provided with anonymity.

It allows elections to be set with the two electoral systems we set out to include in our Functional Specification which are First Past the Post and Single Transferable Vote. Together, they cover the majority of the democratic world.

5.1 Changes from Functional Specification

- The name has been changed to Ethocracy.
    - Why: While a minor issue, vot.ie as a name set false implications
        - It implied the app is Irish and might only be for Irish elections, when it is designed for elections across the democratic world.
        - It implied that the app is a Web 2.0 site, when it is a Web 3.0 app.

- The “Validation” contract which handled blind signature cryptography is removed: cryptography in our application does not involve blind signatures to allow for the masking of a voter’s ballot.
    - Why: With the use of Voter IDs to verify a voter is on the registry, blind signatures serve no purpose, granted that the user does not share both their voter ID and their MetaMask account address they are anonymous by default.

- The “Voting Contract” is no longer an element in the design.
    - Why: The Election contract handles the receiving and tallying of the votes, a separate Voting contract is not required.

- The live “Statistics Engine”, which was to provide live statistics on votes cast in an election, has been removed from the design.
    - Why: Live statistics would violate standard election protocol, compromising voter privacy and providing late voters with extra information earlier voters did not have.
    - Statistics are still provided when the election has ended.

- “Voter Guides” have not been included.
    - Why: Intelligent, user friendly design allows users to infer all they need to know on how to operate the website.
    - There is also a User Manual available. Which serves this purpose otherwise.
    - Teaching users the methodology of different electoral systems is easily accessible in other public websites already, and is outside of the scope of this project.

5.2 User Testing

User testing took place informally as one on one calls with users taking remote desktop control of a developer machine. Users were not required to download the source code or install any tools/dependencies.

Users were instructed to set up an election instance and cast a vote in the created election instance.

4 users were consulted. They provided feedback in the form of questionnaires and conversationally.

The questionnaire used to survey users:

Open Voting Questionnaire

How was your experience setting up an election?

very difficult 1 2 3 4 5 *very easy*

How was your experience setting up an election?

very difficult 1 2 3 4 5 *very easy*

How was your experience voting in an election?

very difficult 1 2 3 4 5 *very easy*

How clear were the results?

very confusing 1 2 3 4 5 *very clear*

Please state your level of agreement for the following:

Use of terms throughout the app are consistent

strongly disagree 1 2 3 4 5 *strongly agree*

Position of messages on the screen is consistent

strongly disagree 1 2 3 4 5 6 7 *strongly agree*

Prompts for inputs are clear

strongly disagree 1 2 3 4 5 6 7 *strongly agree*

What, if anything, would you change about the functionality of the app?

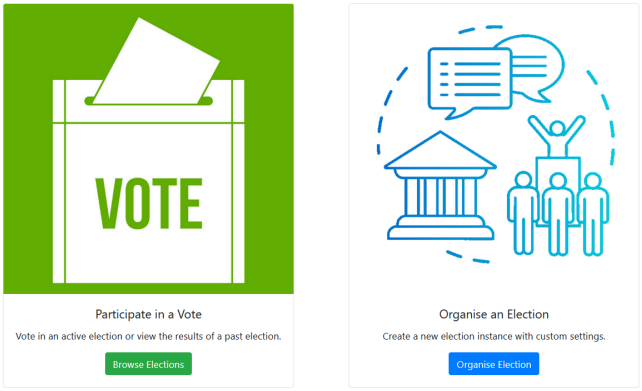
_____
_____
_____
_____

What, if anything, would you change about the design of the app?

_____
_____
_____
_____

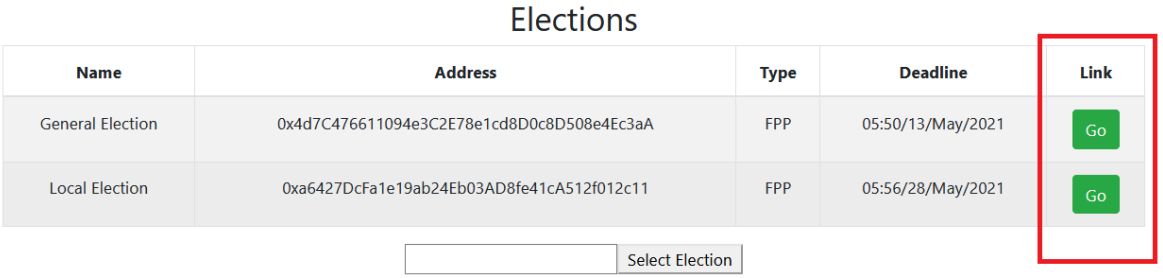
5.2 User Issues

- Users were unsure about the next step to take after arriving on the landing page.
    - Implemented Solution: Descriptive card UI elements added with informative text and photos



- Users were confused about the purpose of the MetaMask extension.
    - Implemented Solution: None. MetaMask could be replaced with other browser-enabled wallets and they are outside of the scope of the project.
    - As the Web3.0 standards become commonplace, browsers should begin to have this functionality built-in with more user friendly UI.

- Users were confused on how to select an election from the election list under “/vote”.
    - Implemented Solution: A user friendly “Go” to election button in line with each election.

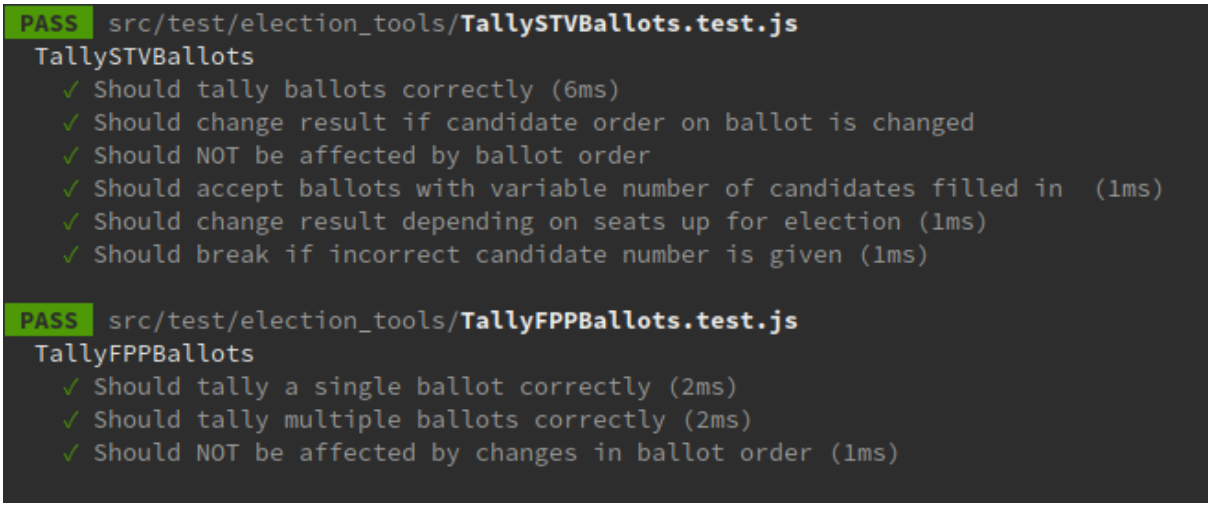


5.3 Unit Testing

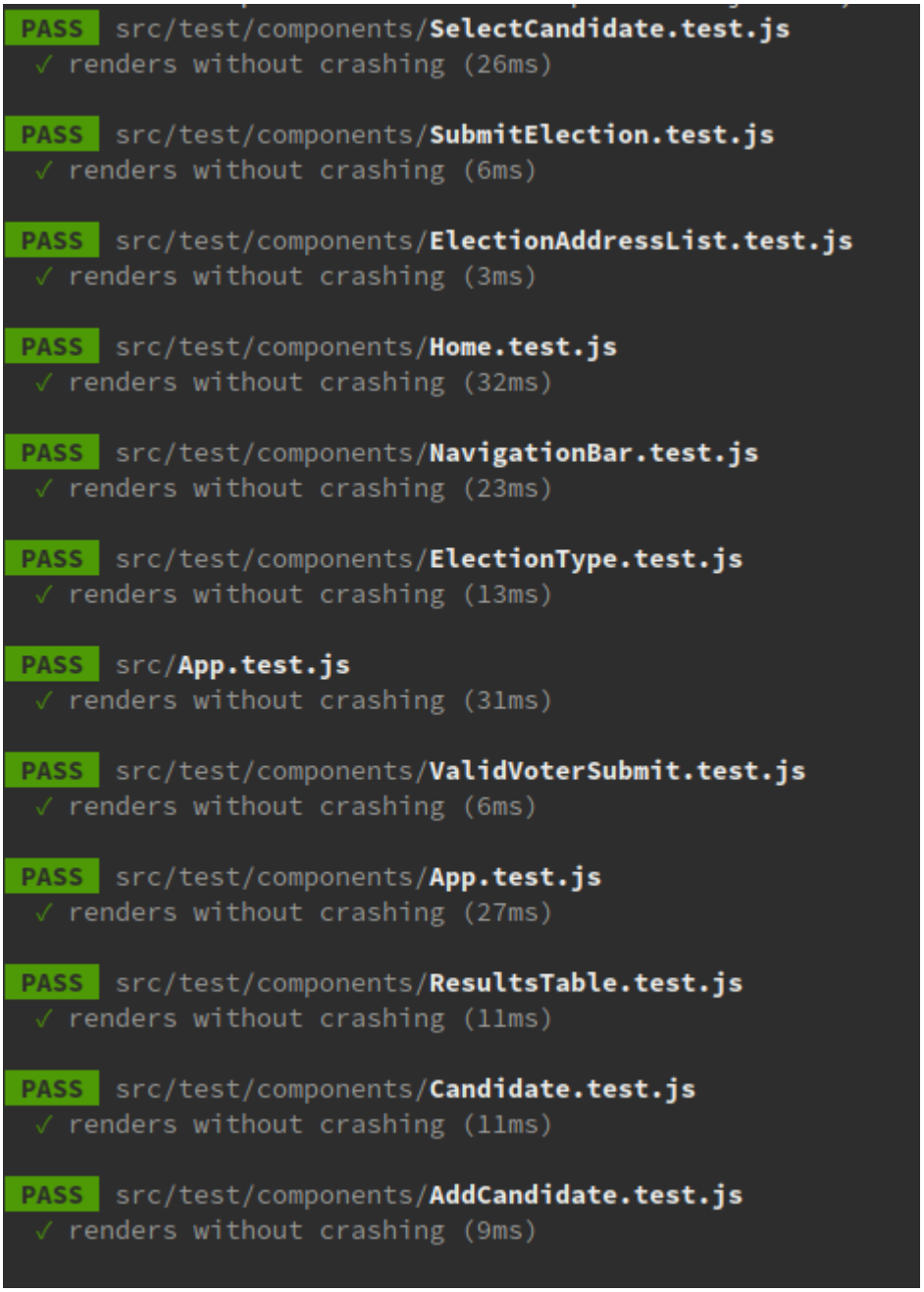
Our testing mainly consisted of ad hoc and unit tests using Jest and truffle test frameworks. We focused our attention on testing critical components found on the solidity contract as well as our tally algorithms. Simple tests were also made to makes sure react components render properly.

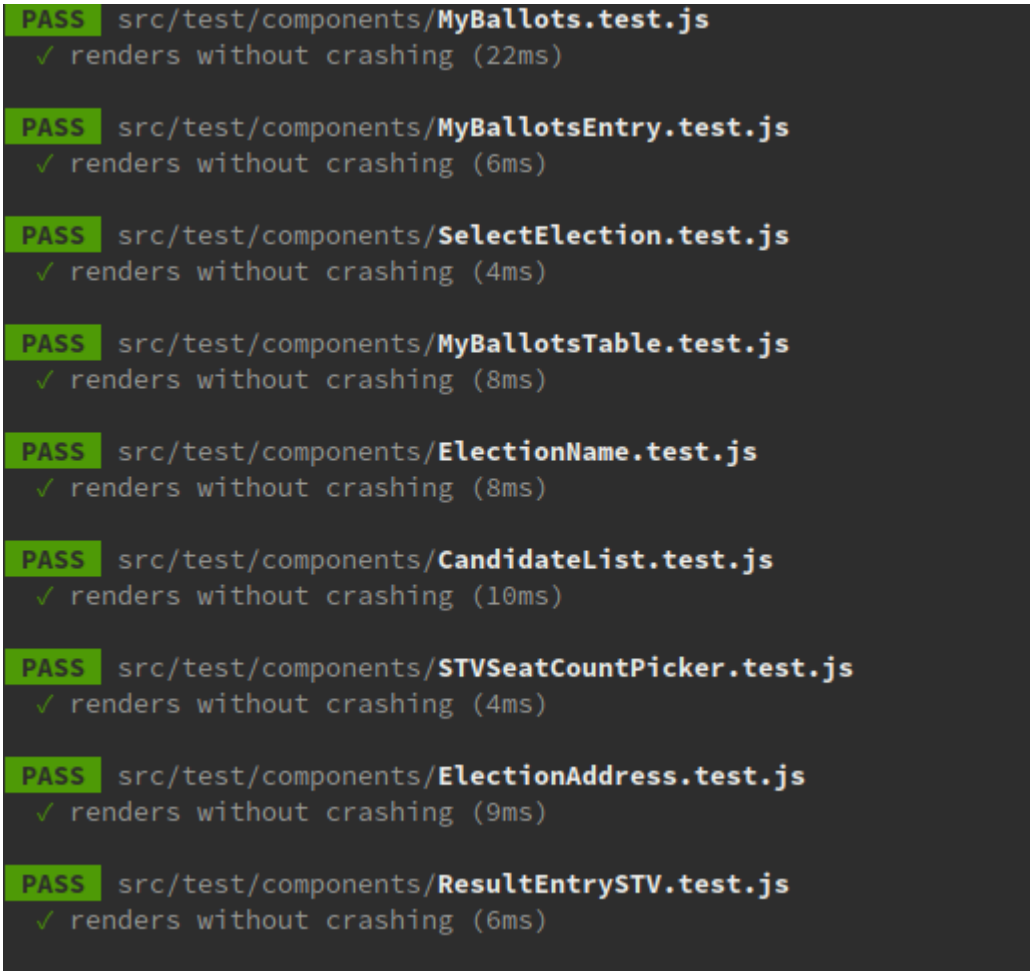
```
(base) [bart@localhost Ethocracy]$ truffle test
Using network 'test'.


Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.



  Contract: Contracts
    ✓ Should deploy the ElectionBuilder contract
    ✓ deployElection should create new Election contract (371ms)
    ✓ deployElection should increment electionCount (39ms)
    ✓ deployElection should store election address
    ✓ deployElection should store used names (40ms)
    ✓ deployElection should increment user election count
    ✓ deployElection should add user address to electionCreators
    ✓ checkDuplicatNames should return false when election name is taken (38ms)
    ✓ checkDuplicatNames should return true when election name is not taken (40ms)
    ✓ addUserBallot should increment the userBallotCount (108ms)
    ✓ addUserBallot should add ballot information to the userBallots (57ms)
    ✓ getUserBallots should return all ballots (66ms)
    ✓ getUserBallots returns correct ballot data (161ms)
    ✓ Election status starts as inProgress (45ms)
    ✓ Election gets deployed without the result key (45ms)
    ✓ Validate user should accept a valid id  (46ms)
    ✓ Validate user should reject a invalid ids  (47ms)


  17 passing (1s)
```

Contract unit tests



Tally algorithms


PASS src/test/components/SelectCandidate.test.js
✓ renders without crashing (26ms)
PASS src/test/components/SubmitElection.test.js
✓ renders without crashing (6ms)
PASS src/test/components/ElectionAddressList.test.js
✓ renders without crashing (3ms)
PASS src/test/components/Home.test.js
✓ renders without crashing (32ms)
PASS src/test/components/NavigationBar.test.js
✓ renders without crashing (23ms)
PASS src/test/components/ElectionType.test.js
✓ renders without crashing (13ms)
PASS src/App.test.js
✓ renders without crashing (31ms)
PASS src/test/components/ValidVoterSubmit.test.js
✓ renders without crashing (6ms)
PASS src/test/components/App.test.js
✓ renders without crashing (27ms)
PASS src/test/components/ResultsTable.test.js
✓ renders without crashing (11ms)
PASS src/test/components/Candidate.test.js
✓ renders without crashing (11ms)
PASS src/test/components/AddCandidate.test.js
✓ renders without crashing (9ms)



React component render tests

Section 6: Future Work

Support for Additional Voting Systems

While the application provides support for the majority of the democratic world, there are several niche, mixed voting systems.

Japan, Lithuania and Taiwan all run mixed elections utilising both implemented systems running in parallel.

France uses a unique system which has special requirements for candidates to be elected, and unique phases which the counting process goes through until these special requirements are met.

Future work could be done on the application to add support for these more unique electoral systems.

Support for Referendums

The application could be adapted to include support for national referendums, with modifications to the ballot and support for unique voting requirements referendums bring.

Live Deployment

The application currently runs on local test blockchain. The smart contracts could be deployed to the main Ethereum network however this would require a monetary investment in the form of ETH, Ethereum's local cryptocurrency.

Full Decentralisation

The application could potentially be made fully decentralised by hosting it on Interplanetary File System (IPFS) which would allow the app to be hosted in a decentralized manner similar to BitTorrent and by replacing the node key manager server with a swarm decentralised data storage. These technologies unfortunately have the downside of being in their early experimental stages of development which is why we have decided against using them in the current version of our application, however they have the potential to be a great addition to our project in the future.