Technical Manual

Project: Votegrity

Author 1: Thomas Kelly
Author 2: Rishabdev Sidhu

Supervisor: Geoffrey Hamilton

Completion Date: 20/04/2024

Table of Contents

1. Abstract p03
2. Motivation p03
3. Research p04
4. Design p06
    4.1. System Architecure p06
    4.2. Class Diagram p07
    4.3. Use Case Diagram p08
    4.4. Sitemap p09
    4.5. Component Diagram p09
    4.6. Data Flow Diagram p10
    4.7. ER Diagram p11
    4.8. High level Design p12
5. Implementation and Sample Code p13
    5.1. Backend-blockchain integration p13
    5.2. Backend-database integration p13
    5.3. Blockchain smart contract p14
    5.4. Gmail API p14
    5.5. Frontend page validation p15
    5.6. Token authentication p15
    5.7. Admin creation p16
6. Problems Solved p17
    6.1. Key Security p17
    6.2. Vote encryption p17
    6.3. Vote decryption p18
    6.4. Rate limiting p18
    6.5. Server security p18
    6.6. Password encryption p19
7. Results p20
8. Future Work p20

1. Abstract

Votegrity is an innovative blockchain-based voting system designed to enhance the transparency, security, and accessibility of election processes in a world where technology, transparency, and confidence in democratic processes are all fast advancing. Leveraging the decentralized and tamper-resistant nature of blockchain technology, the system ensures the voting integrity of each vote cast while providing a secure and transparent platform for users, employs smart contracts to automate and validate the voting process, eliminating the risk of fraud and manipulation. Votegrity prioritizes user privacy and trust, employing cryptographic techniques to safeguard sensitive information and accommodates various authentication and verification methods, ensuring a seamless and accessible voting experience for all users.

2. Motivation

The motivation behind the creation of Votegrity, a groundbreaking blockchain voting system, is deeply rooted in the recognition of the challenges and vulnerabilities that exist in traditional voting systems. We came to this idea when looking at innovative ways of implementing the emerging technology of blockchain in a way that general consumers can avail of without having a technical background and understanding of the process behind using blockchain. We came to realisation that many voting processes are tampered with or fixed, we believed that the use of blockchain for a voting system is the next step forward in how technology can be implemented to modernise the way we vote. The motivation further lies in the pursuit of instilling trust and confidence in the electoral process, empowering individuals to actively participate in shaping their democratic environments.

3. Research

Research plays a pivotal role designing systems, serving as the cornerstone for informed decision-making and successful project outcomes. Before beginning the process of implementing the system, we acknowledged the importance of prior research to gain more knowledge of the system we were about to design and make important decisions as early as possible. Throughout the development of this project, there was quite a bit of research involved as we were working with areas we haven't, or not as often, touched on before including new programming languages, frameworks, cryptographic methods and blockchain technology.

We had never designed a full-stack system using javascript so we spent some time looking at documentation and full stack JS projects on github to understand how different components work together to familiarise ourselves with JavaScript, Node-js and Vue-js. We decided to use MySQL instead of Mongoose db for our database which required a way to synchronize with MySQL. We found sequelize.js along with Atlas. Atlas is a MySQL database adapter for Sequelize which provides an interface for easily connecting, querying, and managing MySQL databases using a node.js server.

We also had to learn Solidity which is a unique language used todeploy smart contracts onto the Ethereum blockchain network. The styling was similar to C which we were familiar with but there were lots of new features, built-in classes and restrictions we had never seen before throughout our course so this took longer than expected. Along with hardhat that is used to deploy smart contracts onto an Ethereum blockchain network. Most of the learning involved watching youtube tutorials as well as experimenting with how both Solidity and Hardhat work together to submit a value onto hardhat's testing node. We then required a way to interact with the deployed smart contract for which we found Web3.js which is a collection of libraries that allow the server to interact with a local or remote Ethereum node. Each user is assigned an ethereum wallet upon creating their account.

For user account security we required tokenisation, fortunately there was a node.js library called jsonwebtoken and thanks to very well made documentation, figuring how it works and implementing this did not take too long. Each user also required an ethereum wallet which they can use to submit their vote onto the blockchain. The tokenisation worked perfectly for the account it was assigned to and so we decided to use the jsonwebtoken library

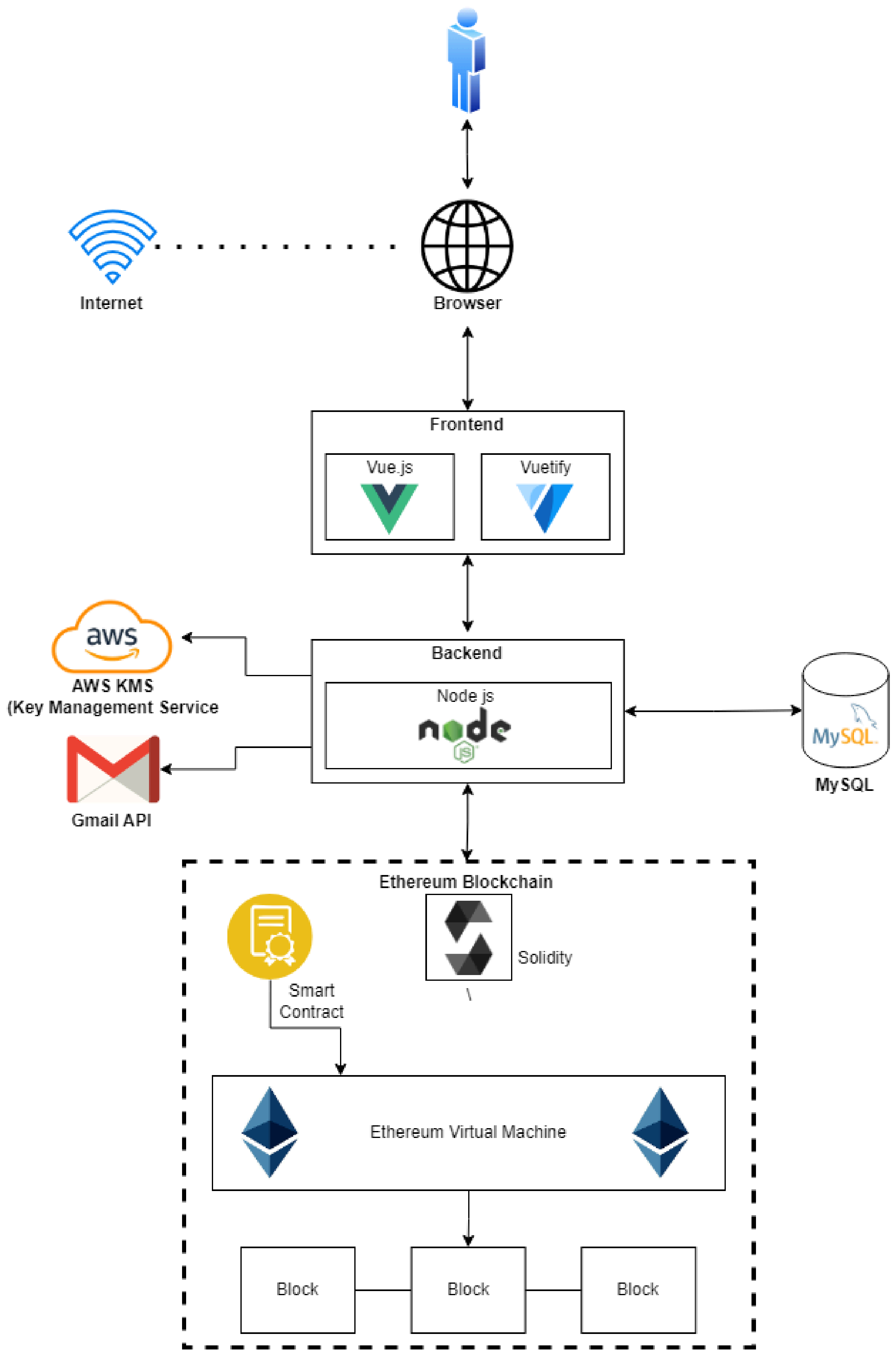
During our project proposal, our supervisor recommended we use homomorphic encryption to encrypt and tally the votes while they are encrypted. This was implemented using paillier keys to encrypt the vote using the public key, encrypt the next vote and tally them up using the public key again. Once all the votes have been tallied up, they are decrypted using the private key to get the final result. Unfortunately the documentation for the paillier library was very sparse so we had little information to work with using homomorphic encryption. This caused many problems as homomorphic multiplication does not multiply two encrypted big integers together but one encrypted big integer and one regular big integer, it took a while to debug this issue and realised why the decrypted sum was wrong.

One issue we encountered during our project was the exposure of the public and private keys, the key pairs could not be stored on a file or in the database in the unprecedented event of data leak so a secure method was required to store they keys even when the database or application code is leaked. After countless hours of brainstorming and research we decided to use the AWS Key Management Services which securely store the keys and will not be exposed. Each time a key is required it can be downloaded then decrypted to use.

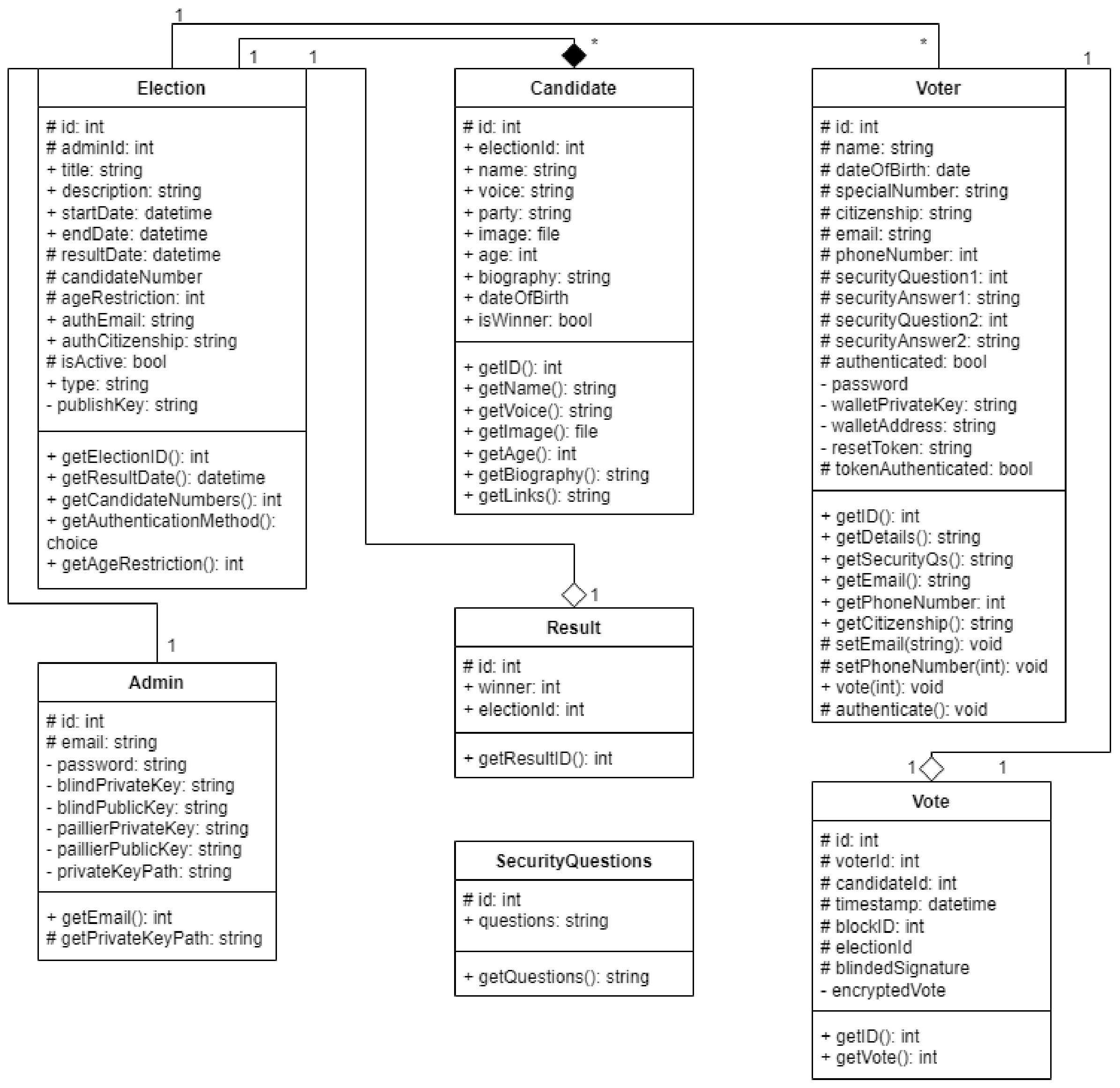
We required automated testing for our application to speed up the development process. We researched multiple testing frameworks for node.js and vue.js. The most popular ones were Mocha, Jest and Jasmine. After multiple attempts we found that Mocha was not suited to our application as it was more suited for Tyepscript so we went ahead to Jest. Jest worked amazingly well so there was no requirement to test out the other testing frameworks.

4. Design

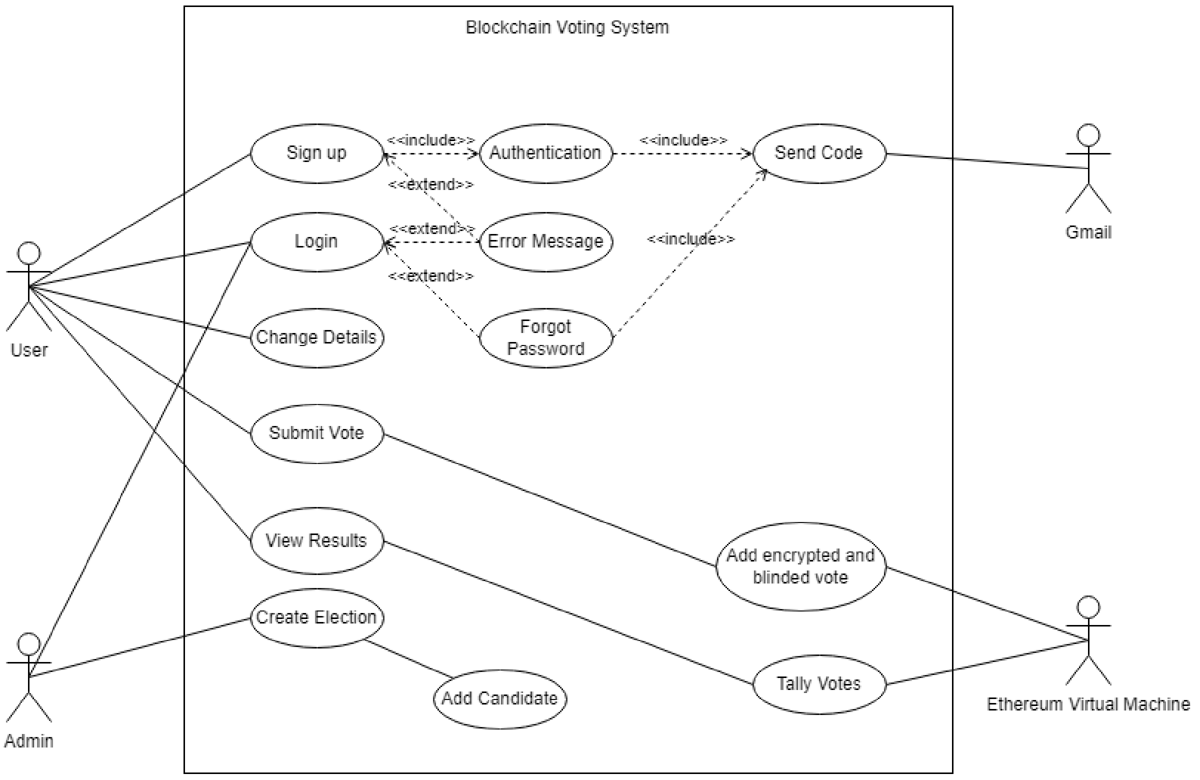
4.1 System Architecture

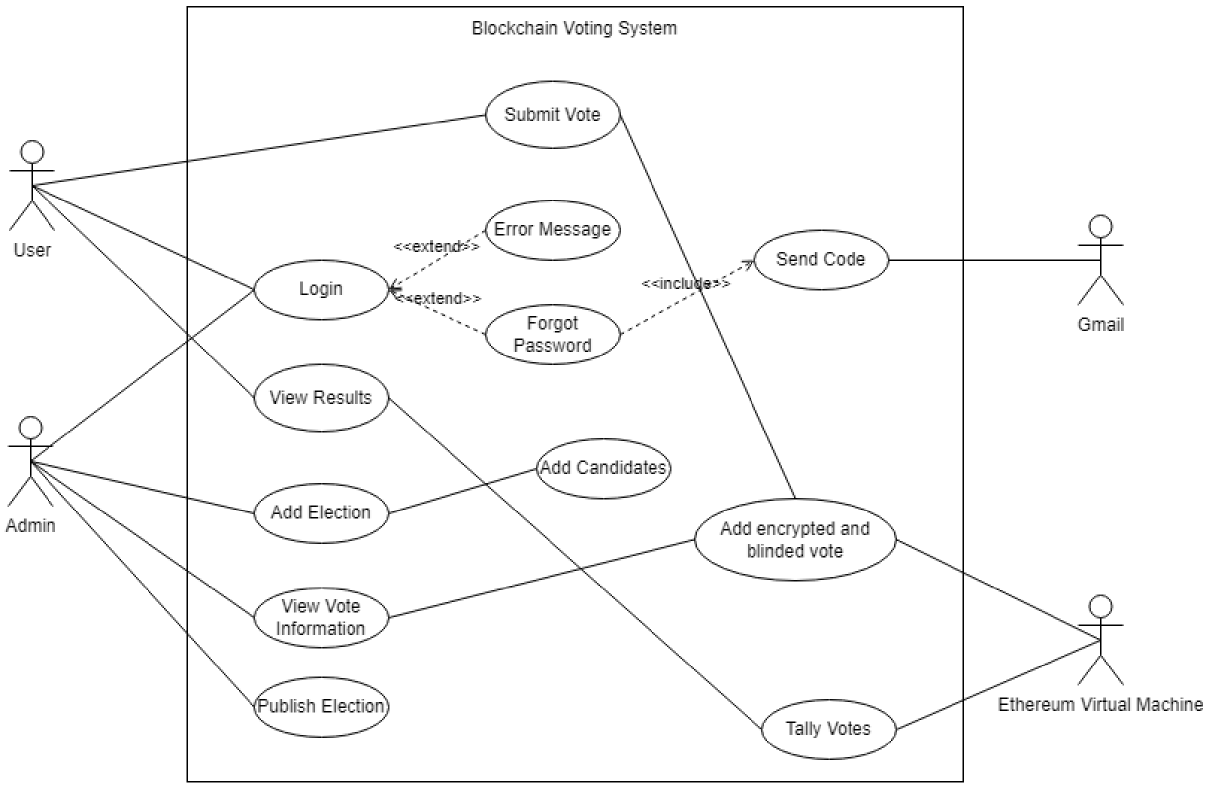


4.2 Class Diagram

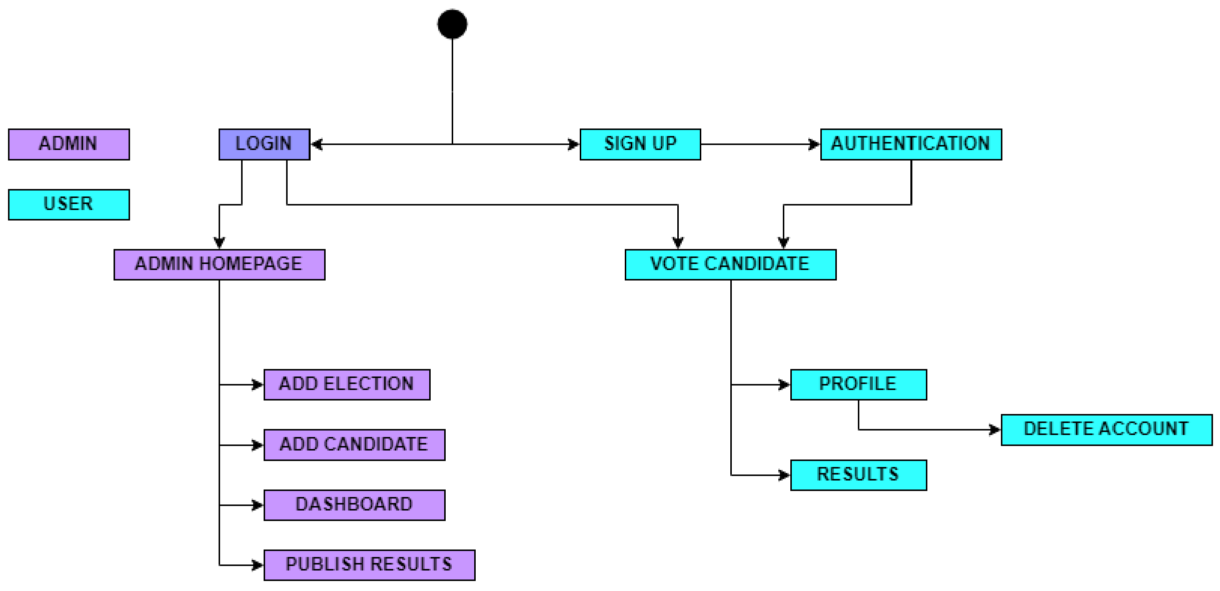


4.3 Use Case Diagram

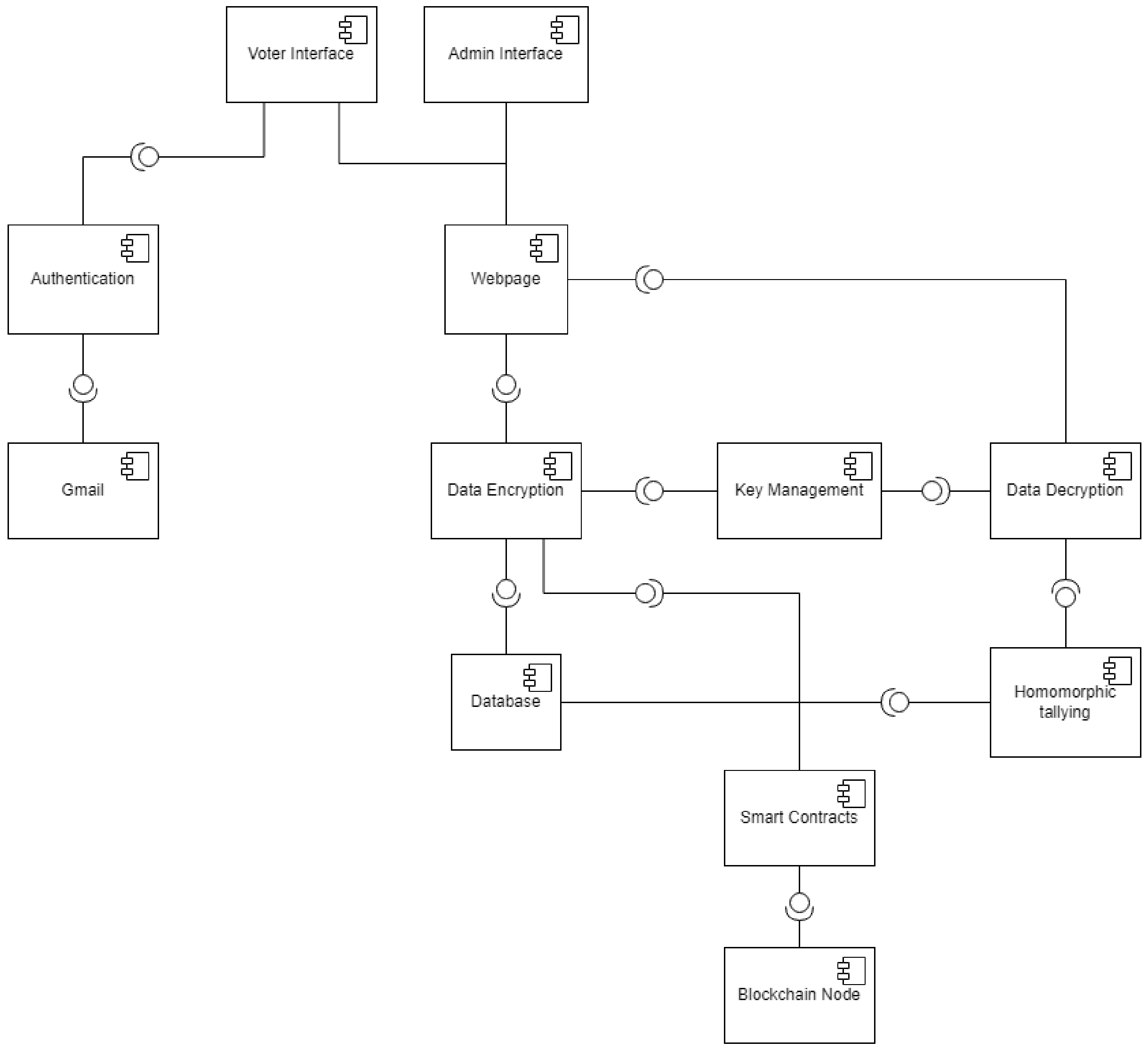




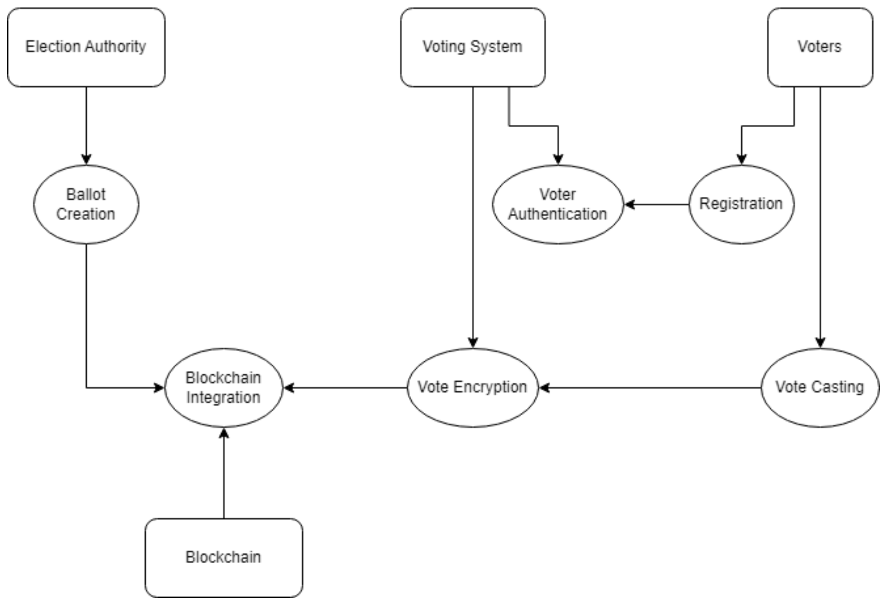
4.4 Sitemap

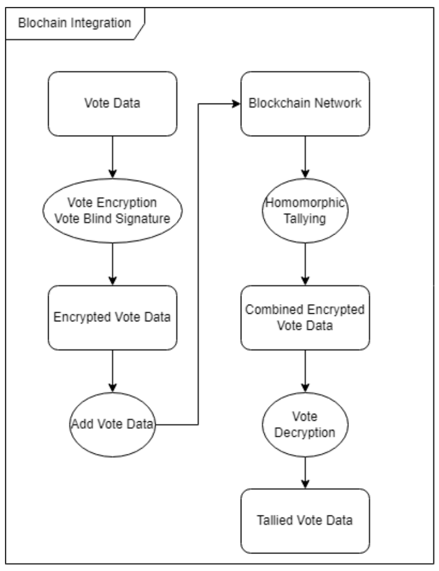


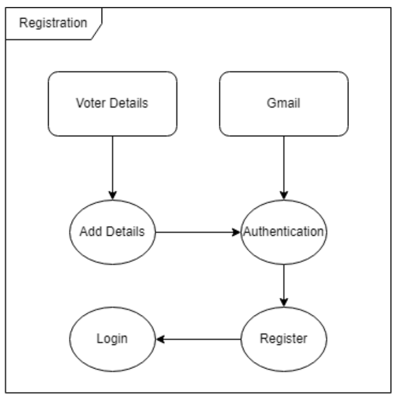
4.5 Component Diagram

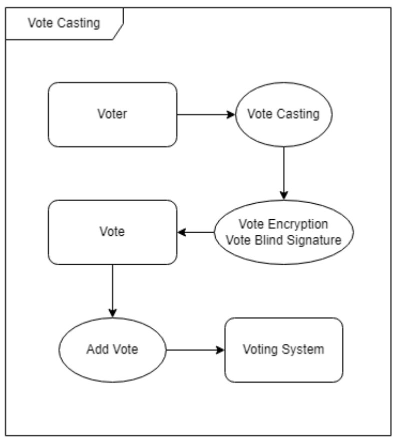


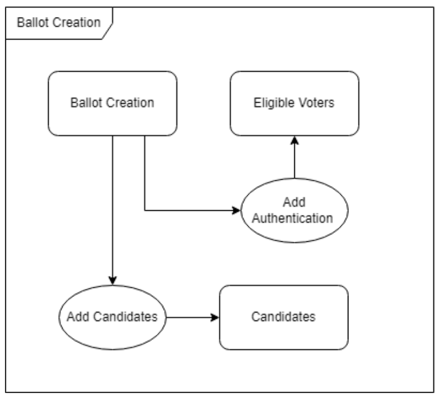
4.6 Data Flow Diagram



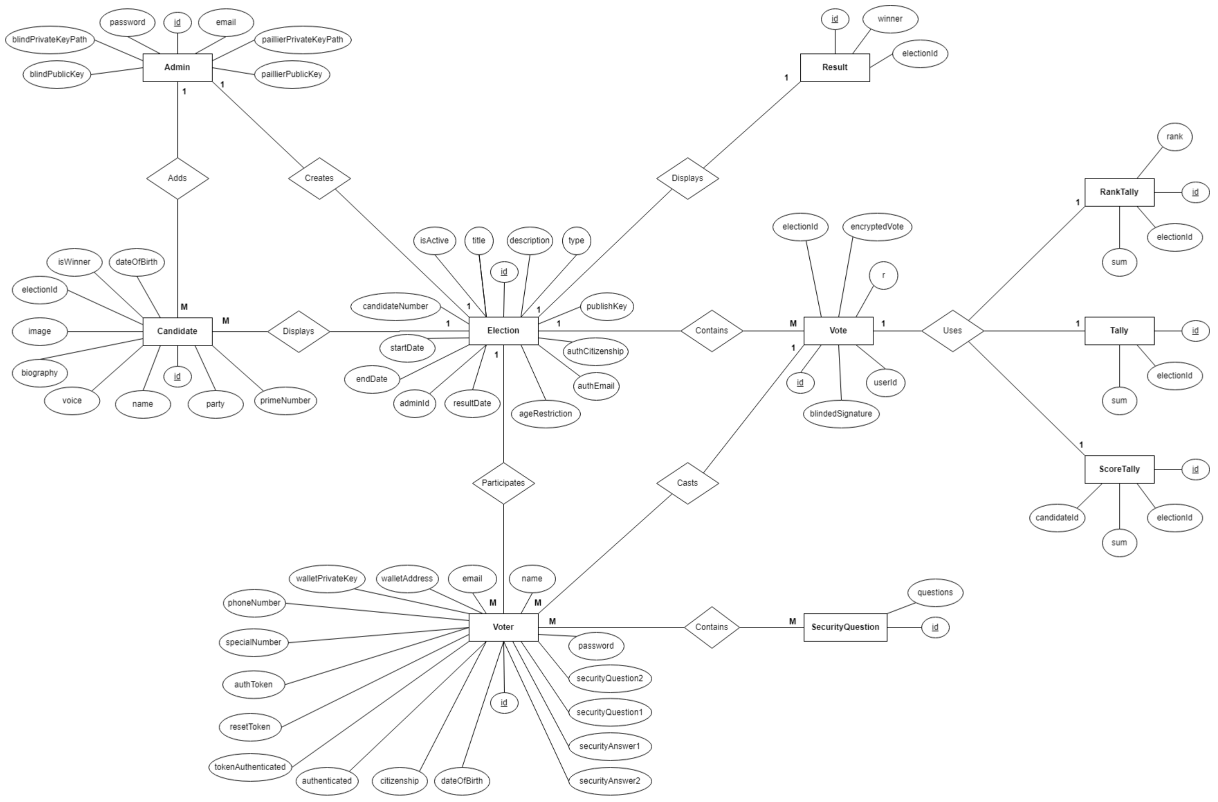




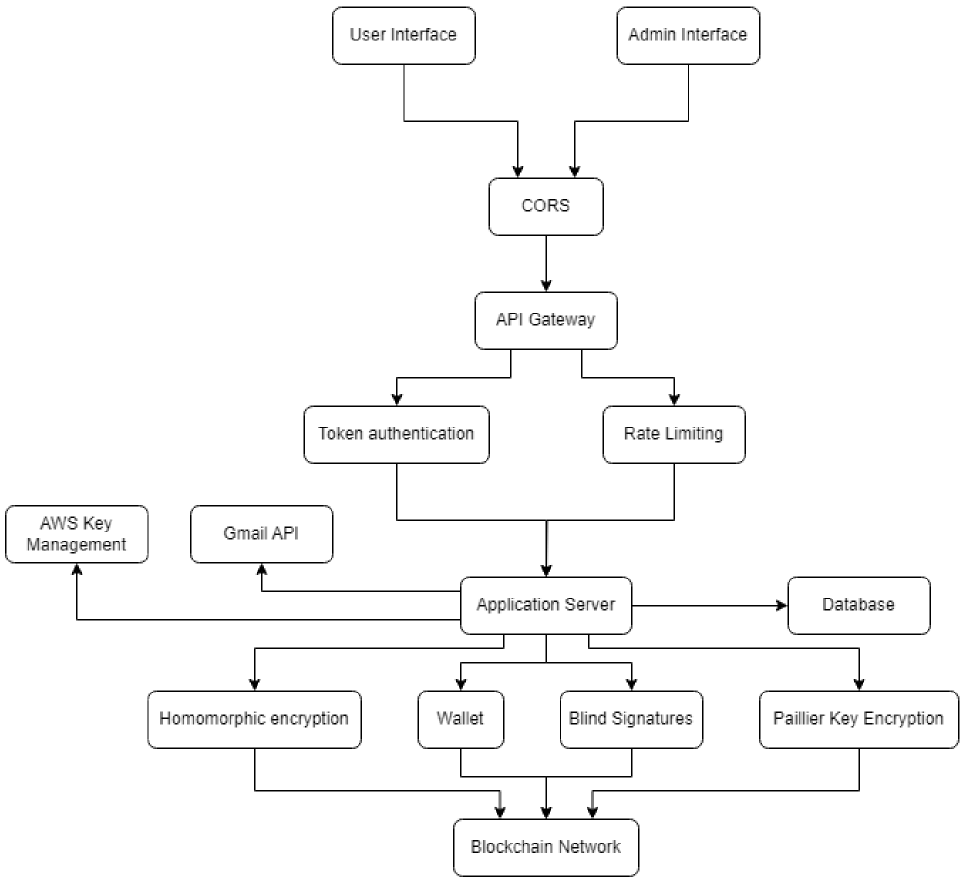




4.7 ER Diagram



4.8 High level Design

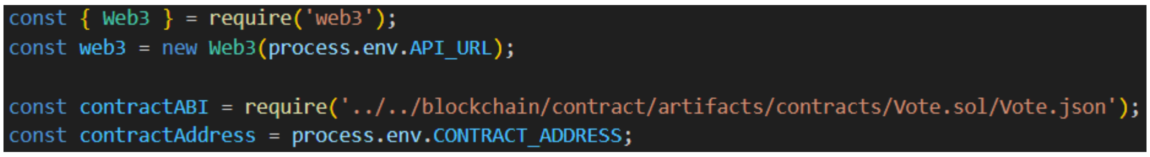

User Interface
Admin Interface
CORS
API Gateway
Token authentication
Rate Limiting
AWS Key Management
Gmail API
Application Server
Database
Homomorphic encryption
Wallet
Blind Signatures
Paillier Key Encryption
Blockchain Network

5. Implementation and Sample Code

Language/Framework	Version	Usage
Node js	v19.9.0	Backend server
Vue js	v3.3.8	Frontend server
MySQL	v8.0.35.0	Database
Solidity	v0.8.0	Smart contract
Hardhat	v2.22.2	Deploy Smart contract
Ethereum	v1.12.2	Blockchain platform

5.1 Backend-blockchain integration

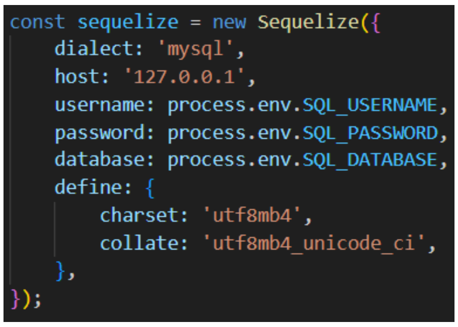
The Solidity smart contract is deployed onto the ethereum network using hardhat, Web3 is implemented to communicate with the deployed smart contract and send or call to the functions in the smart contract on the deployed address



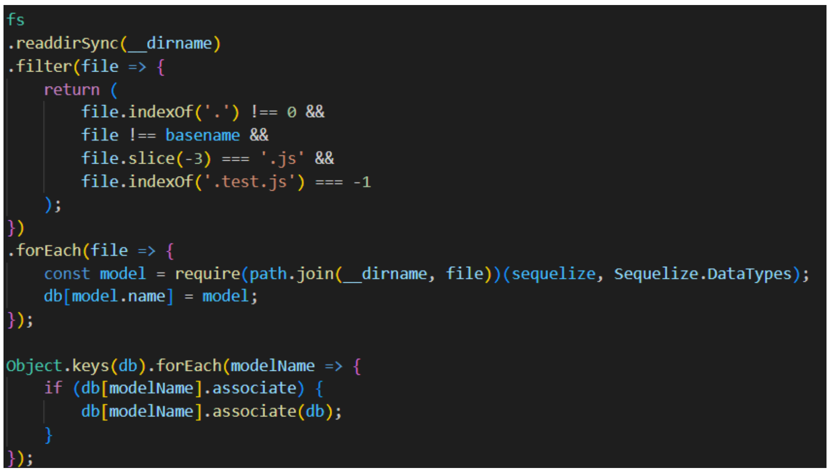
```
const { Web3 } = require('web3');
const web3 = new Web3(process.env.API_URL);

const contractABI = require('../../blockchain/contract/artifacts/contracts/Vote.sol/Vote.json');
const contractAddress = process.env.CONTRACT_ADDRESS;
```

5.1 Backend-database integration

Sequelize and Atlas libraries are implemented to synchronize with the MySQL databaseand perform MySQL operations using javascript to create, update, delete or retrieve data from the MySQL database



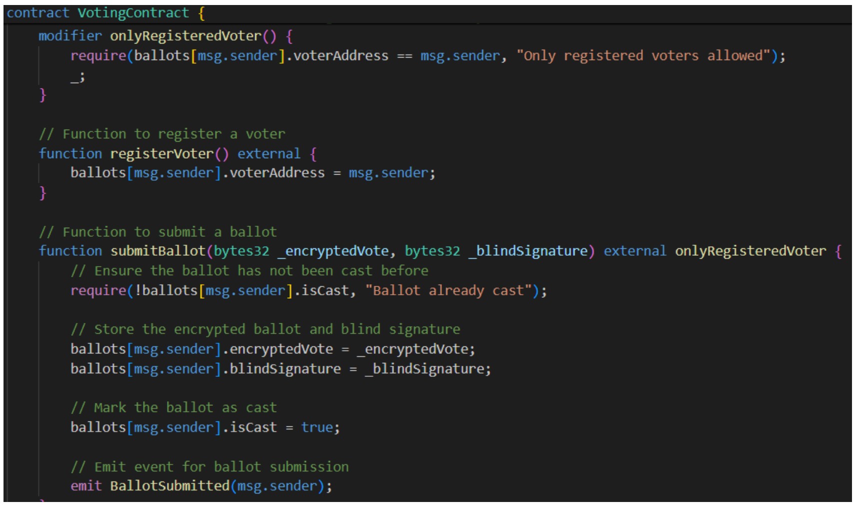
```
const sequelize = new Sequelize({
    dialect: 'mysql',
    host: '127.0.0.1',
    username: process.env.SQL_USERNAME,
    password: process.env.SQL_PASSWORD,
    database: process.env.SQL_DATABASE,
    define: {
        charset: 'utf8mb4',
        collate: 'utf8mb4_unicode_ci',
    },
});
```



```
fs
.readdirSync(__dirname)
.filter(file => {
    return (
        file.indexOf('.') !== 0 &&
        file !== basename &&
        file.slice(-3) === '.js' &&
        file.indexOf('.test.js') === -1
    );
})
.forEach(file => {
    const model = require(path.join(__dirname, file))(sequelize, Sequelize.DataTypes);
    db[model.name] = model;
});

Object.keys(db).forEach(modelName => {
    if (db[modelName].associate) {
        db[modelName].associate(db);
    }
});
```

Blockchain smart contract

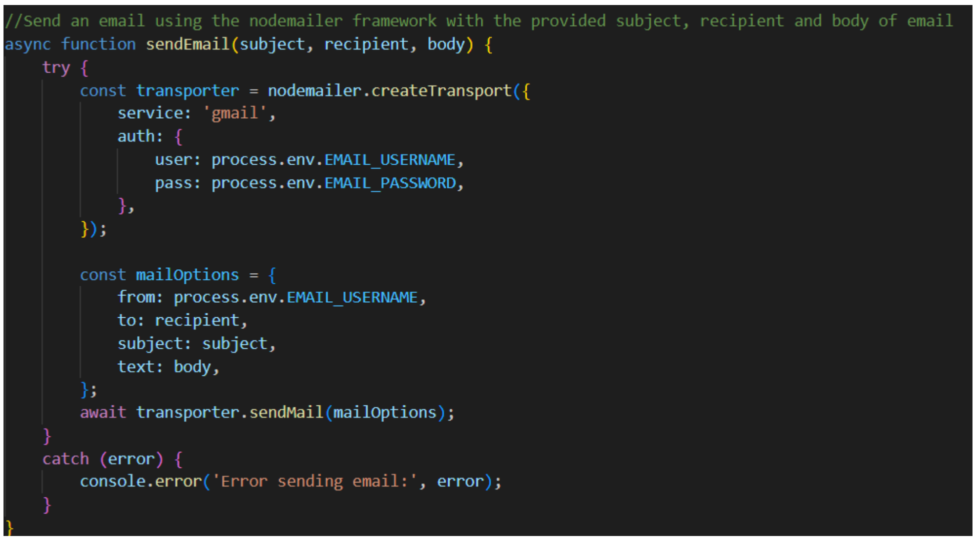
The smart contract is created using Soldity and compiled using hardhat. The smart contract requires a contract to be created with a constructor to assign variables along with functions to perform computations or retrieve data



```
contract VotingContract {
    modifier onlyRegisteredVoter() {
        require(ballots[msg.sender].voterAddress == msg.sender, "Only registered voters allowed");
        _;
    }

    // Function to register a voter
    function registerVoter() external {
        ballots[msg.sender].voterAddress = msg.sender;
    }

    // Function to submit a ballot
    function submitBallot(bytes32 _encryptedVote, bytes32 _blindSignature) external onlyRegisteredVoter {
        // Ensure the ballot has not been cast before
        require(!ballots[msg.sender].isCast, "Ballot already cast");

        // Store the encrypted ballot and blind signature
        ballots[msg.sender].encryptedVote = _encryptedVote;
        ballots[msg.sender].blindSignature = _blindSignature;

        // Mark the ballot as cast
        ballots[msg.sender].isCast = true;

        // Emit event for ballot submission
        emit BallotSubmitted(msg.sender);
```

Gmail API

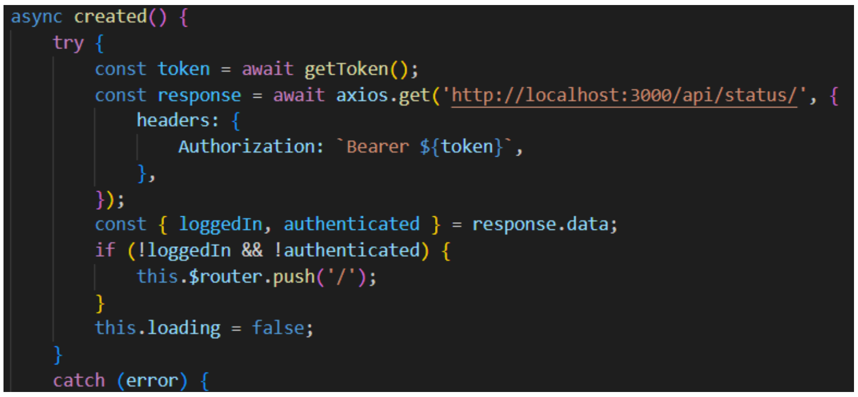
To send emails a gmail account was created for the system, A 16 digit code is used to communicate with the gmail services and send an email to the provided recipient along with the subject and text to send



```
//Send an email using the nodemailer framework with the provided subject, recipient and body of email
async function sendEmail(subject, recipient, body) {
    try {
        const transporter = nodemailer.createTransport({
            service: 'gmail',
            auth: {
                user: process.env.EMAIL_USERNAME,
                pass: process.env.EMAIL_PASSWORD,
            },
        });

        const mailOptions = {
            from: process.env.EMAIL_USERNAME,
            to: recipient,
            subject: subject,
            text: body,
        };
        await transporter.sendMail(mailOptions);
    }
    catch (error) {
        console.error('Error sending email:', error);
    }
}
```

Frontend page validation

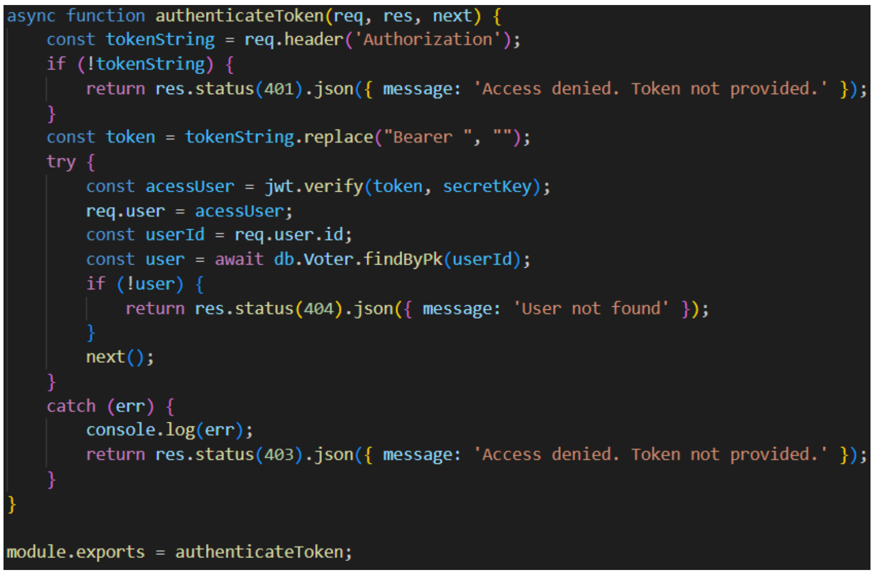
The client side retrieves the status of the user from the backend to determine if they are logged in, logged out, not authenticated or an admin so they can be redirected if they are on the incorrect page



```
async created() {
    try {
        const token = await getToken();
        const response = await axios.get('http://localhost:3000/api/status/', {
            headers: {
                Authorization: `Bearer ${token}`,
            },
        });
        const { loggedIn, authenticated } = response.data;
        if (!loggedIn && !authenticated) {
            this.$router.push('/');
        }
        this.loading = false;
    }
    catch (error) {
```

Token authentication

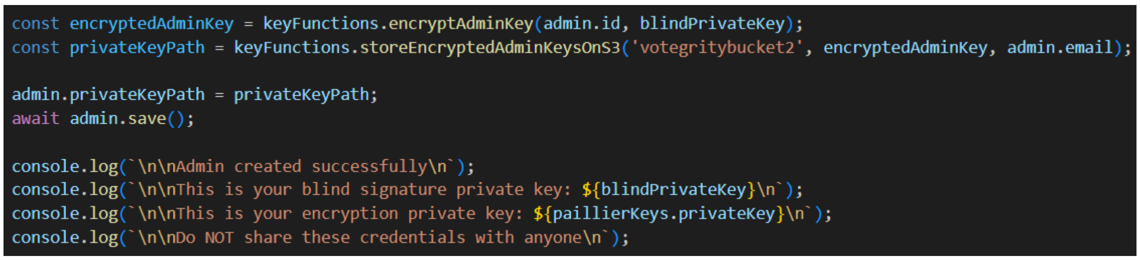
To prevent the user from having to login everytime they visit a new page, tokens are implemented to allow user's to stay logged in from the client side, preserve their password by using an irreversible token and provide a method of ensuring the user's authenticity when making a request to the server



```
async function authenticateToken(req, res, next) {
    const tokenString = req.header('Authorization');
    if (!tokenString) {
        return res.status(401).json({ message: 'Access denied. Token not provided.' });
    }
    const token = tokenString.replace("Bearer ", "");
    try {
        const acessUser = jwt.verify(token, secretKey);
        req.user = acessUser;
        const userId = req.user.id;
        const user = await db.Voter.findByPk(userId);
        if (!user) {
            return res.status(404).json({ message: 'User not found' });
        }
        next();
    }
    catch (err) {
        console.log(err);
        return res.status(403).json({ message: 'Access denied. Token not provided.' });
    }
}

module.exports = authenticateToken;
```

Admin creation

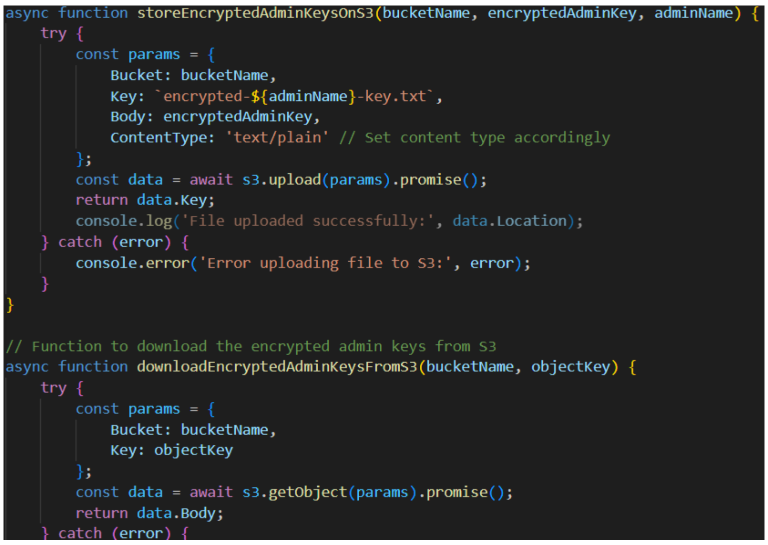
For security purposes, the admin can only be created on the backend server instead of the client side to prevent anyone from making an admin account as only those with access to the system can do so. The admin simply enters in their details and their account is created



```
const encryptedAdminKey = keyFunctions.encryptAdminKey(admin.id, blindPrivateKey);
const privateKeyPath = keyFunctions.storeEncryptedAdminKeysOnS3('votegritybucket2', encryptedAdminKey, admin.email);

admin.privateKeyPath = privateKeyPath;
await admin.save();

console.log(`\n\nAdmin created successfully\n`);
console.log(`\n\nThis is your blind signature private key: ${blindPrivateKey}\n`);
console.log(`\n\nThis is your encryption private key: ${paillierKeys.privateKey}\n`);
console.log(`\n\nDo NOT share these credentials with anyone\n`);
```

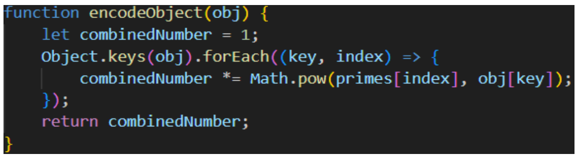
6. Problems Solved

Key Security:
Problem:
The paillier and blind signature keys used for encryption and decryption were exposed in the application and the database so a method or location was required to store the keys securely and access them at any given time
Solution:
The utilization of AWS Key Management Service (KMS) was selected as the best method to securely store the paillier and blind signature keys after encryption.



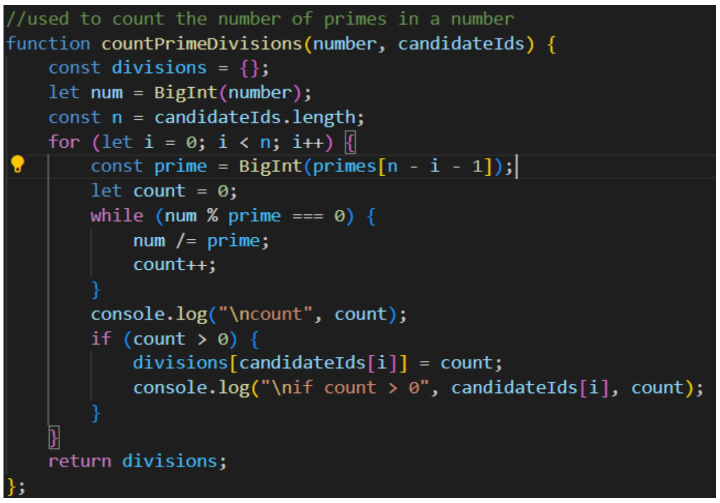
```
async function storeEncryptedAdminKeysOnS3(bucketName, encryptedAdminKey, adminName) {
    try {
        const params = {
            Bucket: bucketName,
            Key: `encrypted-${adminName}-key.txt`,
            Body: encryptedAdminKey,
            ContentType: 'text/plain' // Set content type accordingly
        };
        const data = await s3.upload(params).promise();
        return data.Key;
        console.log('File uploaded successfully:', data.Location);
    } catch (error) {
        console.error('Error uploading file to S3:', error);
    }
}

// Function to download the encrypted admin keys from S3
async function downloadEncryptedAdminKeysFromS3(bucketName, objectKey) {
    try {
        const params = {
            Bucket: bucketName,
            Key: objectKey
        };
        const data = await s3.getObject(params).promise();
        return data.Body;
    } catch (error) {
```

Vote encryption:
Problem:
Ranked voting is produced as an object in the form {candidateId_1: rank_1, … candidateId_n: rank_n}, the object must be turned into one integer and be able to perform homomorphic tallying on it.
Solution:
The votes are encrypted and tallied using paillier public keys facilitated by prime numbers for unique assignment for each candidate’s id

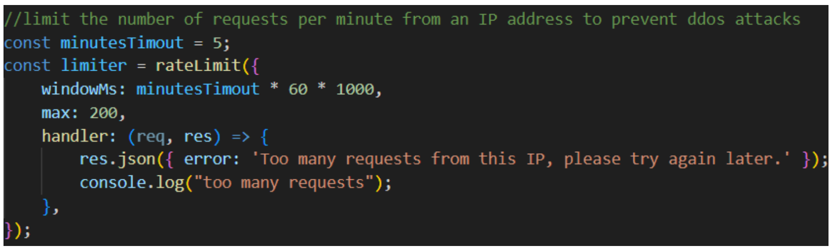


```
function encodeObject(obj) {
    let combinedNumber = 1;
    Object.keys(obj).forEach((key, index) => {
        combinedNumber *= Math.pow(primes[index], obj[key]);
    });
    return combinedNumber;
}
```

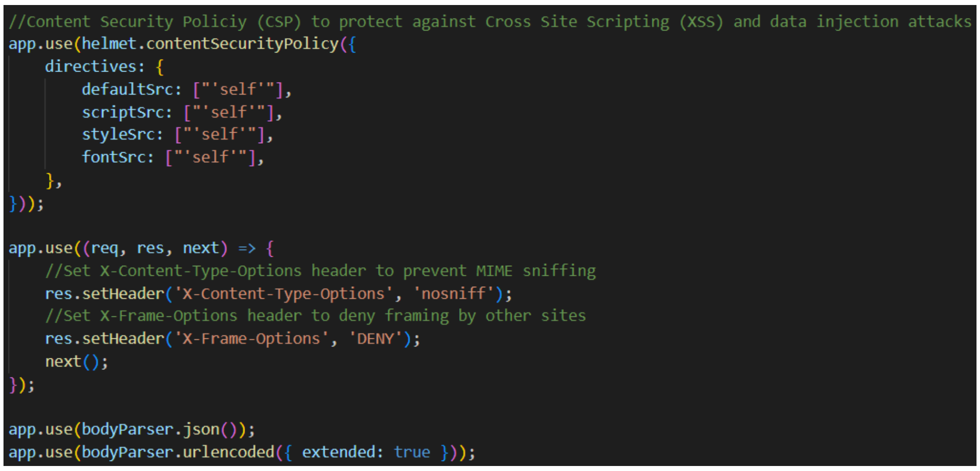
Vote decryption:
Problem:
Once the votes have been encrypted and tallied, one big number is produced as a result and the rankings must be decrypted into their respective candidate Id's and total tallied votes
Solution:
The votes are decrypted using paillier private keys and converted into their original form using prime factorisation to determine how many votes were received for each candidate with their assigned id to prime number


```
//used to count the number of primes in a number
function countPrimeDivisions(number, candidateIds) {
    const divisions = {};
    let num = BigInt(number);
    const n = candidateIds.length;
    for (let i = 0; i < n; i++) {
        const prime = BigInt(primes[n - i - 1]);
        let count = 0;
        while (num % prime === 0) {
            num /= prime;
            count++;
        }
        console.log("\ncount", count);
        if (count > 0) {
            divisions[candidateIds[i]] = count;
            console.log("\nif count > 0", candidateIds[i], count);
        }
    }
    return divisions;
};
```

Rate limiting:
Problem:
Flooding attacks and DDoS attacks can cause disruption to the voting process and denial of service to actual users
Solution:
Rate limiting has been implemented to mitigate DDoS attacks and stop individuals from attempting multiple incorrect login attempts, only a certain number of requests per minute can be made from a specific IP address

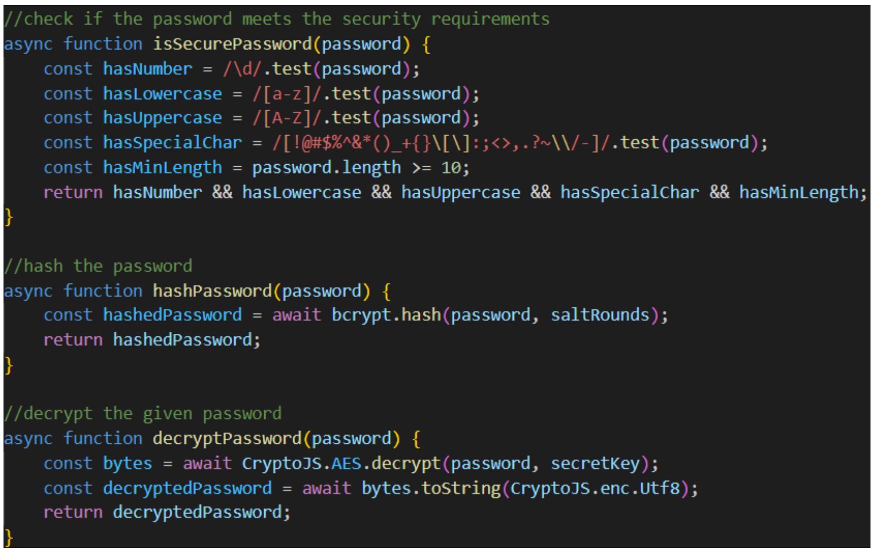

```
//limit the number of requests per minute from an IP address to prevent ddos attacks
const minutesTimout = 5;
const limiter = rateLimit({
    windowMs: minutesTimout * 60 * 1000,
    max: 200,
    handler: (req, res) => {
        res.json({ error: 'Too many requests from this IP, please try again later.' });
        console.log("too many requests");
    },
});
```

Server security:
Problem:
Using ZAP for system testing there were many vulnerabilities detected in the system
Solution:
Various measures have been implemented to safeguard the server from vulnerability threats including CORS, CSP, X-Content, X-Frame protections and a few others pointed out by ZAP



```
//Content Security Policiy (CSP) to protect against Cross Site Scripting (XSS) and data injection attacks
app.use(helmet.contentSecurityPolicy({
    directives: {
        defaultSrc: ["'self'"],
        scriptSrc: ["'self'"],
        styleSrc: ["'self'"],
        fontSrc: ["'self'"],
    },
}));

app.use((req, res, next) => {
    //Set X-Content-Type-Options header to prevent MIME sniffing
    res.setHeader('X-Content-Type-Options', 'nosniff');
    //Set X-Frame-Options header to deny framing by other sites
    res.setHeader('X-Frame-Options', 'DENY');
    next();
});

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```

Password encryption:
Problem:
The passwords are exposed when submitted to login and sent via the API to the server where a man in the middle can steal the user's password.
Solution:
Symmetric encryption was employed to securely transmit the password from the frontend to the backend via API by encrypting it using a secret key, and subsequently, the password is then hashed into the database with added salt rounds.



```
//check if the password meets the security requirements
async function isSecurePassword(password) {
    const hasNumber = /\d/.test(password);
    const hasLowercase = /[a-z]/.test(password);
    const hasUppercase = /[A-Z]/.test(password);
    const hasSpecialChar = /[!@#$%^&*()_+{}\[\]:;<>,.?~\\/-]/.test(password);
    const hasMinLength = password.length >= 10;
    return hasNumber && hasLowercase && hasUppercase && hasSpecialChar && hasMinLength;
}

//hash the password
async function hashPassword(password) {
    const hashedPassword = await bcrypt.hash(password, saltRounds);
    return hashedPassword;
}

//decrypt the given password
async function decryptPassword(password) {
    const bytes = await CryptoJS.AES.decrypt(password, secretKey);
    const decryptedPassword = await bytes.toString(CryptoJS.enc.Utf8);
    return decryptedPassword;
}
```

7. Results

Following thorough system testing, the system successfully encrypts and submits the user’s vote to the blockchain via the smart contract, ensuring their security and integrity throughout the process. Subsequently, upon retrieval, the encrypted votes are decrypted and tallied accurately using homomorphic encryption, demonstrating the robustness and reliability of the decryption mechanism. Additionally, rigorous security testing was conducted to assess and validate the system's quality, reliability and safety against potential vulnerabilities accidental or incidental, further enhancing confidence in its overall quality. Testing documentation has been provided that includes, unit testing, integration testing, system testing, security testing and user acceptance testing. The integration of single-transferrable voting into the system proved challenging within the allotted time frame, primarily due to the complexity involved in tallying encrypted votes whilst keeping the next preference of each vote given a candidate has exceeded their quota for votes or a candidate is to be eliminated thus the need to transfer the excess votes.

8. Future Work

There is most certainly a wide use for a web-application like this in the current market. In the future there is consideration to add more voting processes, diversifying the system's applicability to various electoral scenarios. For added verification and authentication, the system could incorporate AI-driven document verification for users' passports or IDs, enhancing security and trust in the voter authentication process. Furthermore, extending Votegrity's accessibility by building a dedicated mobile application will cater to the evolving preferences of users who seek a seamless and convenient voting experience on their mobile devices without having to go to the web page. In the future more time and research will be dedicated to making Single-Transferable Voting possible using homomorphically added data, due to the limitations of time constraints this voting process could not be added to the system but with enough dedicated time and dedication it will certainly be possible to incorporate Single-Transferable Voting into a secure and encrypted blockchain voting system. Recognizing the broad potential for an application of this nature in the current market, future work will focus on refining and optimizing Votegrity to meet the growing demand for secure, transparent, and technologically advanced voting solutions.