SCHOOL OF INFORMATION, COMPUTER AND COMMUNICATION TECHNOLOGY

SIRINDHORN INTERNATIONAL INSTITUTE OF TECHNOLOGY
THAMMASAT UNIVERSITY

Capstone Project Technical Report
Secure and Transparent Voting System using Smart Contracts and Blockchain

Group Members:

6222771337	Ruihua	Lu
6222772046	Sirichai	Srinu
6222782953	Sirapop	Lahanakew

Submitted to:

Dr. Watthanasak Jeamwatthanachai

CSS486 Blockchain Development
Semester 2 Academic Year 2022

Table of Content

Chapter 1: Introduction

This technical report serves as a detailed exploration to design and implementation of a voting system built on blockchain technology and smart contracts, developed as a capstone project. The main goal is to develop an application that takes full advantage of the transformative capabilities of blockchain technology and smart contracts. Through the development of this capstone project, our objective is to ensure the integrity and fairness in the electoral process, fostering trust and confidence among all participants.

Traditional voting systems often face challenges related to trust, transparency, and security. By harnessing the decentralized nature of blockchain technology, we aim to address these issues and provide a reliable platform for conducting elections. Through the utilization of smart contracts on the Ethereum blockchain, all votes will be recorded in an immutable and tamper-proof manner, establishing a high level of trust in the system.This report will delve into the key components of the voting smart contract, including the structuring of data, the functionality of various functions, and the overall architecture of the application. It will also discuss the challenges encountered during the development process and the solutions implemented to mitigate them.

Furthermore, this report will highlight the potential impact of the developed voting application, emphasizing the benefits it offers such as increased transparency, enhanced security, and improved accessibility. It will also outline the areas for further improvement and expansion, allowing for future enhancements and adaptations to meet evolving requirements.

Overall, this capstone project aims to showcase the potential of blockchain technology in revolutionizing the voting process, paving the way for more inclusive, secure, and trustworthy elections. Through this report, readers will gain insights into the technical details and considerations involved in the development of a blockchain-based voting smart contract, fostering a deeper understanding of its practical implications and potential applications.

Chapter 2: Design Specification

The design specification for the capstone project entails the development of a blockchain-based voting application that aims to ensure secure and transparent elections through the use of smart contracts. The application will facilitate user registration, requiring individuals to provide their personal information for identification purposes. Authentication mechanisms, such as digital signatures, will be implemented to verify the identity of registered users. The voting process will involve presenting users with a list of candidates or ballot measures, allowing them to make their selections through the application. These votes will be encrypted and stored on the blockchain, ensuring their immutability. Once the voting period concludes, the application will tally the votes and determine the winners based on the election rules. The election results will be publicly available on the blockchain, providing transparency and allowing anyone to verify the fairness of the election process.

As the smart contract will be designed for the voting application, it should include crucial functions related to vote management, authentication, and security measures. These functions enable users to interact with the contract and perform essential actions such as registration, identity authentication, and secure vote casting. Moreover, the smart contract should incorporate mechanisms that automatically calculate the vote tally and determine the winner based on predefined voting rules. The use of a blockchain network ensures the transparency and immutability of the voting results, safeguarding them against tampering or modifications.

To complement the smart contract, a user-friendly frontend interface would allow users to seamlessly register, authenticate, cast their votes, and access the election results. Acting as a link between users and the smart contract, the interface facilitates secure and anonymous interaction with the contract's functions. Integrating the frontend interface with the smart contract ensures that users can confidently participate in the voting process, ensuring the privacy and integrity of their votes.

The designed mockup for the voting application incorporates key features essential to the voting process. The registration page should allow users to provide their personal details, enabling them to create an account and actively participate in the voting process. Once registered, users can authenticate their identity using a secure method such as a digital signature or login credentials through the login page.

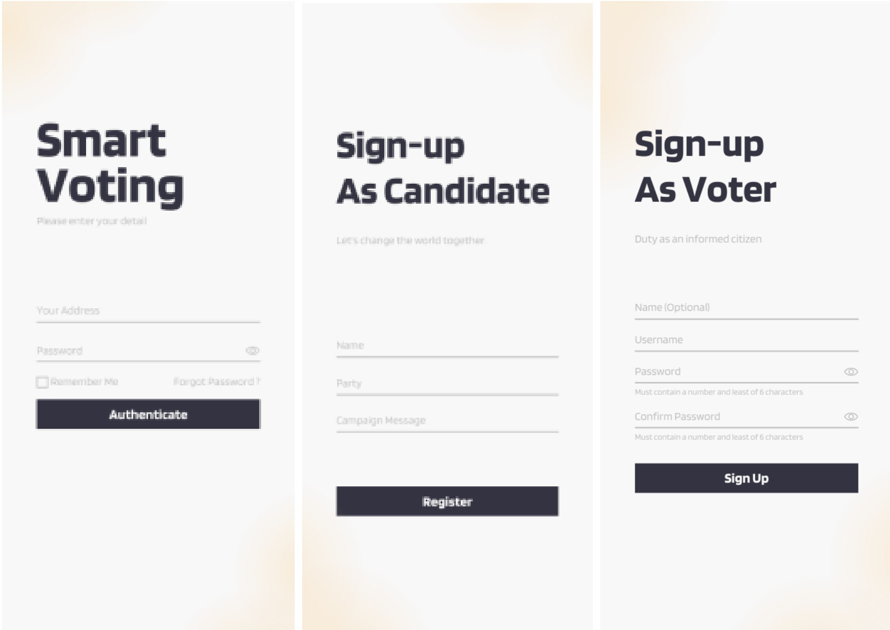


Figure 1, 2, and 3: Illustrating the authentication page and registration pages for both candidates and voters.

The mockup should also feature a user-friendly voting interface where users can view a comprehensive list of candidates or ballot measures, including candidate names, party affiliations, and campaign messages. Users can easily make their selections and securely submit their votes through the interface.

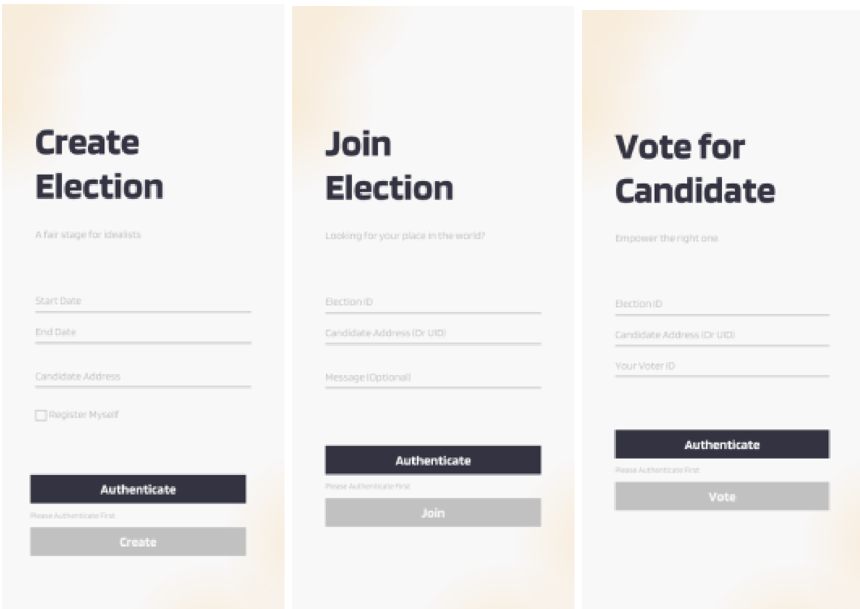


Figure4, 5, and 6: Illustrating the process of creating, and joining elections, and voting for a candidate.

To ensure transparency, the mockup includes a results page that displays the total vote count for each candidate or measure, facilitating the declaration of winners based on the predefined voting rules. Overall, the mockup design prioritizes a seamless and intuitive user experience, allowing users to navigate the voting process effortlessly, cast their votes securely, and access the election results efficiently.

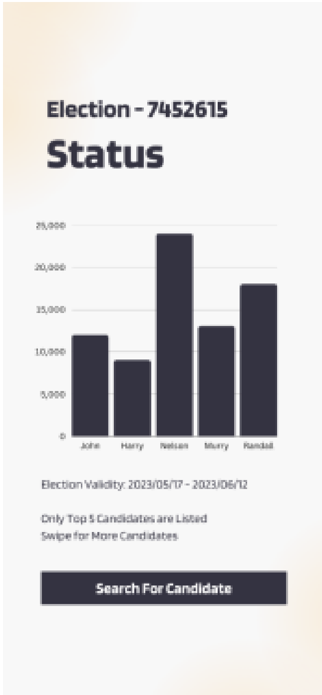


Figure 7: Illustrating the result page of balloting in an election.

Chapter 3: Methodology

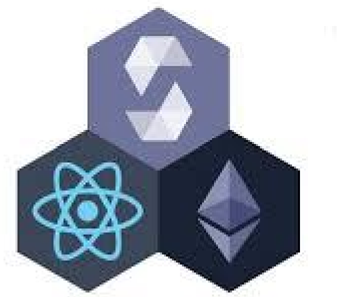


Figure 8: Technology Stack

The capstone project leveraged a comprehensive technology stack comprising React.js, Solidity, Metamask, Hardhat, and ethers.js. This stack synergistically facilitated the development of a decentralized voting system that prioritized user-friendliness while harnessing the potential of blockchain technology and smart contracts. The integration of these technologies ensured transparency, security, and immutability throughout the entire voting process.

To set up the development environment for the capstone project, several components were integrated, including React, Metamask, Ganache, and Ethereum. Each of these elements played a crucial role in facilitating the creation and evaluation of a decentralized application.

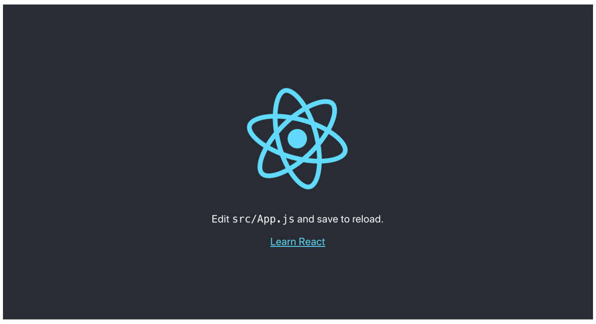


Figure 9: create-react-app

Initially, React.js, a widely used JavaScript library for building user interfaces, was employed as the foundational framework for frontend development. The setup process involved the installation of Node.js and npm (Node Package Manager), followed by the creation of a new React project using the "create-react-app" command. This automatically generated the necessary dependencies and project structure.

While smart contract exemplifies a simple voting system on the Ethereum blockchain using Solidity. It consists of the following components:

1. Structs:
   - Voter: Represents a voter with a username and password.
   - Candidate: Represents a candidate with a name, party affiliation, and campaign message.
   - Vote: Contains the address of a candidate and the corresponding vote count.

```
pragma solidity ^0.8.0;
// WARNING: For using the ElectionID as parameter, use 0x as prefix. Such as the
first election is usually named '0x1'.
contract VotingSystem {
   event ErrorOccurred(string errorMessage);

   struct Voter {
       string username;
       string password;
   }
   struct Candidate {
       string name;
       string party;
       string campaignMessage;
   }
   struct Vote {
       address candidateAddr;
       uint256 voteCount;
   }

   struct Election {
       uint256 startDate;
       uint256 endDate;
       address[] candidateAddresses;
       address[] voterAddresses;
       mapping(address => uint256) voteCount;
       mapping(address => bool) hasVoted;
   }
```

2. ElectionType enum:
   - Defines different types of elections: General, Primary, Runoff, and Referendum.

```
   enum ElectionType {
       General,
       Primary,
```

```
    Runoff,
    Referendum
}
```

3. State Variables:

- `voters`: Maps voter addresses to Voter structs.
- `candidates`: Maps candidate addresses to Candidate structs.
- `elections`: Maps election IDs (bytes32) to Election structs.
- `candidateAddresses`: Stores all candidate addresses.
- `electionCounter`: A counter to track the number of elections.

```
mapping(address => Voter) voters;
mapping(address => Candidate) candidates;
mapping(bytes32 => Election) elections;

address[] public candidateAddresses;
bytes32 public electionCounter;
```

4. Events:

- `ErrorOccurred`: Emits an event when an error occurs during voting.

5. Functions:

- `register`: Allows a user to register as a voter by providing a username and password.
- `authenticate`: Verifies the provided username and password against the stored voter's credentials (currently unused).
- `registerCandidate`: Allows a user to register as a candidate by providing their name, party affiliation, and campaign message.
- `getCandidate`: Retrieves candidate information based on the candidate's address.
- `hasVoted`: Checks whether a voter has already voted in a specific election.
- `vote`: Enables a voter to cast their vote for a particular candidate in an election.
- `isValidCandidate`: A helper function to validate if a candidate is valid within a given election.
- `getResultsFromElection`: Retrieves the results (candidates and their vote counts) for a specific election.
- `getResults`: Returns all candidates (currently unused).

- `getCandidateAddress`: Returns the candidate address based on the index in the candidateAddresses array.
- `getNumCandidates`: Returns the number of registered candidates.
- `getCandidateByName`: Retrieves the candidate address based on the candidate's name.
- `createElection`: Creates a new election with the specified start and end dates, and an optional list of candidate addresses.
- `addCandidateToElection`: Adds a candidate address to an existing election.
- `getCandidateAddressesForElection`: Returns the addresses of all candidates participating in a specific election.
- `findCandidateIndexv3`: A helper function to find the index of a candidate address in an array.
- `incrementBytes32`: A helper function to increment a bytes32 value.

```
   function register(string memory _username, string memory _password) public { // 
Register as voter
       voters[msg.sender] = Voter(_username, _password);
   }

   function authenticate( // Check Username and Password, not utilized atm
       string memory _username,
       string memory _password
   ) public view returns (bool) {
       Voter memory voter = voters[msg.sender];
       return (keccak256(abi.encodePacked(_username)) ==
           keccak256(abi.encodePacked(voter.username)) &&
           keccak256(abi.encodePacked(_password)) ==
           keccak256(abi.encodePacked(voter.password)));
   }

   function registerCandidate( // Register as candidate
       string memory _name,
       string memory _party,
       string memory _campaignMessage
   ) public {
       address candidateAddr = msg.sender;
       candidates[msg.sender] = Candidate(_name, _party, _campaignMessage);
       candidateAddresses.push(candidateAddr);
   }

   function getCandidate( // Get candidate from address
```

```
        address _candidateAddr
    ) public view returns (Candidate memory) {
        return candidates[_candidateAddr];
    }

    function hasVoted( // Check whether the voter is voted in the election specified
by electionID
        bytes32 _electionId,
        address _voterAddress
    ) public view returns (bool) {
        return elections[_electionId].hasVoted[_voterAddress];
    }

    function vote( // Vote by given ElectionID and both addresses, Checks whether
candidate is in the election and whether voter is voted. If so, emit errors.
        bytes32 _electionId,
        address _voterAddress,
        address _candidateAddress
    ) public {
        if (!isValidCandidate(_electionId, _candidateAddress)) {
            emit ErrorOccurred("Invalid candidate");
        }

        Election storage election = elections[_electionId];

        if (election.hasVoted[_voterAddress]) {
            emit ErrorOccurred("User Voted");
        }

        election.hasVoted[_voterAddress] = true;

        election.voteCount[_candidateAddress]++;
    }

    function isValidCandidate( // Helper function used in vote()
        bytes32 _electionId,
        address _candidateAddr
    ) internal view returns (bool) {
        Election storage election = elections[_electionId];
        for (uint256 i = 0; i < election.candidateAddresses.length; i++) {
            if (election.candidateAddresses[i] == _candidateAddr) {
                return true;
            }
        }
        return false;
```

```
    }

    function getResultsFromElection( // Provide electionID, returns 2 arrays with
first being all candidates(informations), second being their votes
        bytes32 _electionID
    ) public view returns (Candidate[] memory, uint256[] memory) {
        Election storage election = elections[_electionID];
        uint256 numCandidates = election.candidateAddresses.length;
        Candidate[] memory result = new Candidate[](numCandidates);
        uint256[] memory voteCounts = new uint256[](numCandidates);

        for (uint256 i = 0; i < numCandidates; i++) {
            address candidateAddr = election.candidateAddresses[i];
            result[i] = candidates[candidateAddr];
            voteCounts[i] = election.voteCount[candidateAddr];
        }

        return (result, voteCounts);
    }

    function getResults() public view returns (Candidate[] memory) { // Return all
candidates, (Not used)
        uint256 numCandidates = getNumCandidates();
        Candidate[] memory result = new Candidate[](numCandidates);
        for (uint256 i = 0; i < numCandidates; i++) {
            result[i] = getCandidate(getCandidateAddress(i));
        }
        return result;
    }

    function getCandidateAddress(uint256 _index) public view returns (address) { //
Given an index in candidate array, return the address (Not used)
        return candidateAddresses[_index];
    }

    function getNumCandidates() public view returns (uint256) {
        uint256 count = 0;
        for (uint256 i = 0; i < candidateAddresses.length; i++) {
            count++;
        }
        return count;
    }

    function getCandidateByName( // Given a name of candidate return the address
        string memory _name
```

```
    ) public view returns (address) {
        for (uint256 i = 0; i < candidateAddresses.length; i++) {
            Candidate memory candidate = candidates[candidateAddresses[i]];
            if (
                keccak256(abi.encodePacked(candidate.name)) ==
                keccak256(abi.encodePacked(_name))
            ) {
                return candidateAddresses[i];
            }
        }
        revert("Candidate not found");
    }
    function createElection( // the _candidateAddresses can be explicitly defined by
passing addresses, or just []
        uint256 _startDate,
        uint256 _endDate,
        address[] memory _candidateAddresses
    ) public {
        bytes32 electionId = electionCounter;
        electionCounter = incrementBytes32(electionCounter);
        Election storage newElection = elections[electionId];
        newElection.startDate = _startDate;
        newElection.endDate = _endDate;
        newElection.candidateAddresses = _candidateAddresses;
    }

    function addCandidateToElection(
        bytes32 _electionID,
        address _candidateAddr
    ) public {
        elections[_electionID].candidateAddresses.push(_candidateAddr);
    }

    function getCandidateAddressesForElection( // Used to check all the current
candidates in an election
        bytes32 _electionID
    ) public view returns (address[] memory) {
        return elections[_electionID].candidateAddresses;
    }

    function findCandidateIndexv3( // helper function for checking the candidate is
registered in an election or not
        address[] memory _candidateAddresses,
        address _candidateAddress
    ) public pure returns (uint256) {
```

```
        for (uint256 i = 0; i < _candidateAddresses.length; i++) {
            if (_candidateAddresses[i] == _candidateAddress) {
                return i;
            }
        }
        return 2 ** 256 - 1; // Fancier -1 but okay
    }

    function incrementBytes32(bytes32 _value) private pure returns (bytes32) { //
Helper function
        return bytes32(uint256(_value) + 1);
    }
}
```

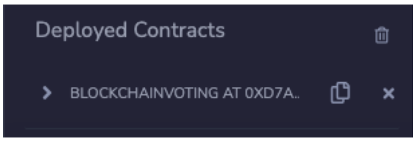
Metamask, a browser extension wallet, was then incorporated to enable seamless interaction with the Ethereum blockchain. By installing the Metamask extension on a compatible browser, such as Google Chrome or Mozilla Firefox, a new Ethereum wallet was established. This wallet granted access to Ethereum network accounts, allowing for the execution of transactions and interactions with smart contracts during the development phase.

Ganache, a local Ethereum blockchain development tool, was utilized to create a simulated blockchain environment for testing and development purposes. By downloading and installing Ganache, a local blockchain network with predefined accounts and test Ether was set up. This provided a controlled environment for deploying and testing smart contracts without the need for a live Ethereum network.

Finally, the Ethereum blockchain itself played a fundamental role in the development process. It served as the underlying technology for the decentralized application, providing the necessary infrastructure for secure and transparent transactions and smart contract execution. By leveraging the capabilities of React, Metamask, Ganache, and Ethereum, the capstone project was able to establish a robust development environment conducive to the creation of a decentralized application.

Chapter 4: Functionality Test and Verification

The process of utilizing the functionality of a smart contract involves several steps. First, the smart contract is deployed onto a blockchain network, such as Ethereum, using the Hardhat development framework. This includes compiling the contract code, creating a migration script, and deploying it to a test network or a local development blockchain like Ganache.



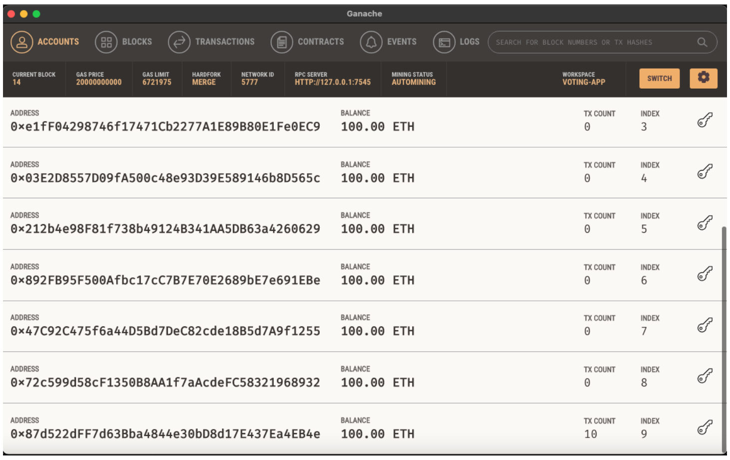
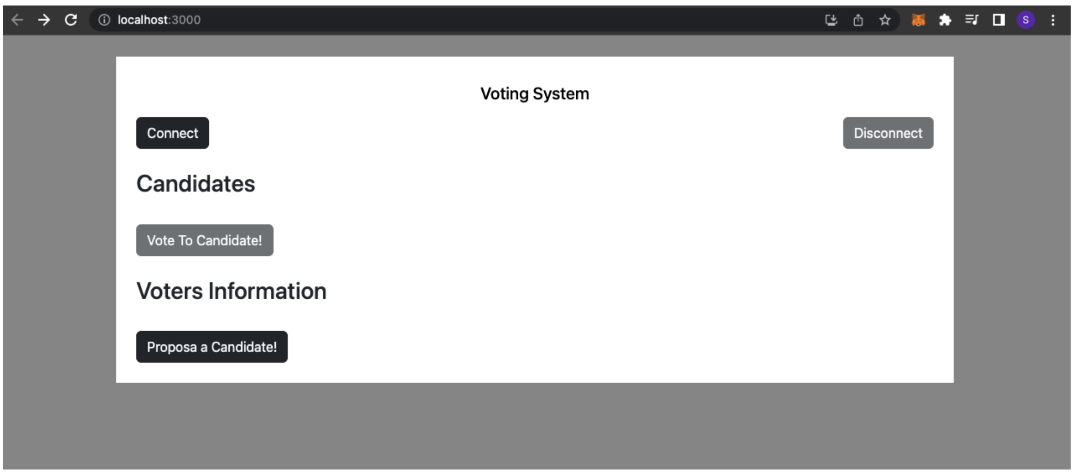
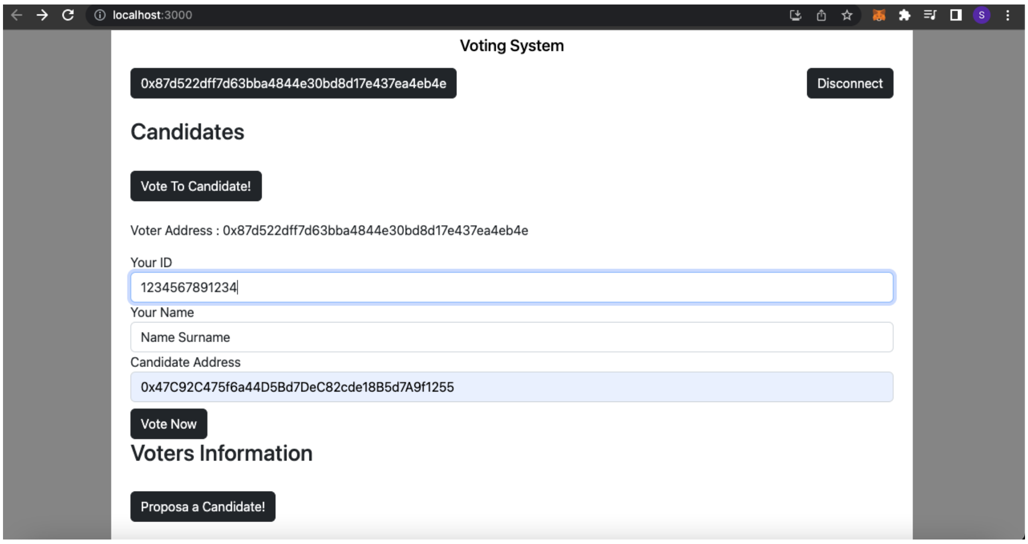


Figure 10, and 11: Solidity smart contract deployment, and Ganache local blockchain network

Next, a web application or client-side interface is set up to interact with the smart contract. This is achieved by utilizing frameworks like React.js or Angular, along with libraries such as ethers.js or web3.js, which provide APIs for interacting with the contract.





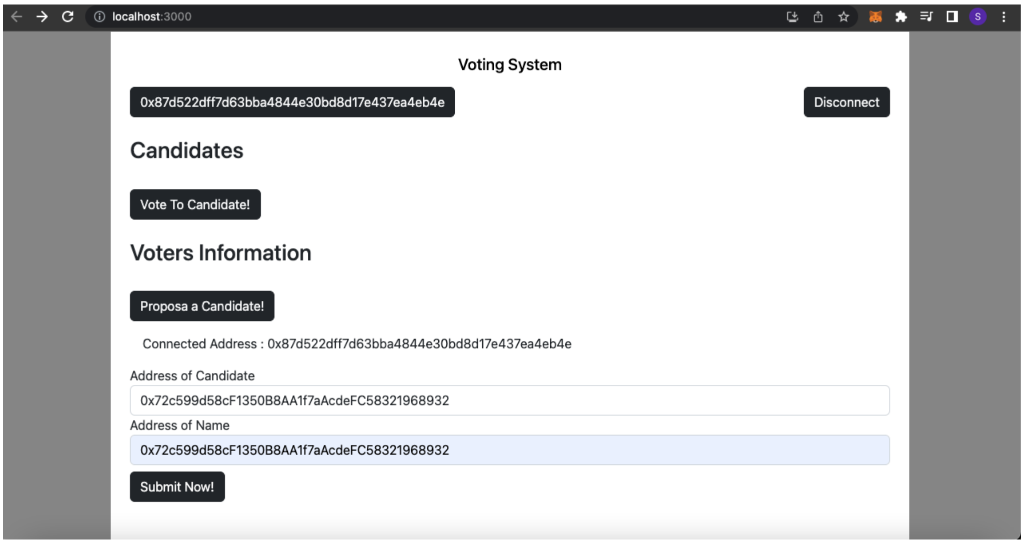


Figure 11, 12, and 13 : The user interface designed to evaluate the fundamental capabilities of the smart contract.

Instances of the deployed smart contract are then created within the web application by specifying the contract address and ABI (Application Binary Interface), which represents the contract's functions and data structures in a JSON format.

To establish a connection to the blockchain network, tools like MetaMask are employed. MetaMask acts as a bridge between the browser and the Ethereum network, allowing for the management of Ethereum accounts and enabling secure transaction signing.

For user authentication purposes, a mechanism is implemented within the web application to ensure that only authorized users can access specific contract functionalities. This can involve integrating authentication providers or implementing custom login systems.

Once authenticated, users can invoke smart contract functions by calling the relevant methods within the contract instances. These function calls enable users to retrieve data from the contract or modify its state.

When invoking functions that modify the contract's state, such as voting or updating candidate information, users are required to sign the transactions using their private keys. This signing process is facilitated by the web3 provider, such as MetaMask, which prompts users to confirm the transaction and signs it with their private key. The signed transaction is then submitted to the blockchain network.

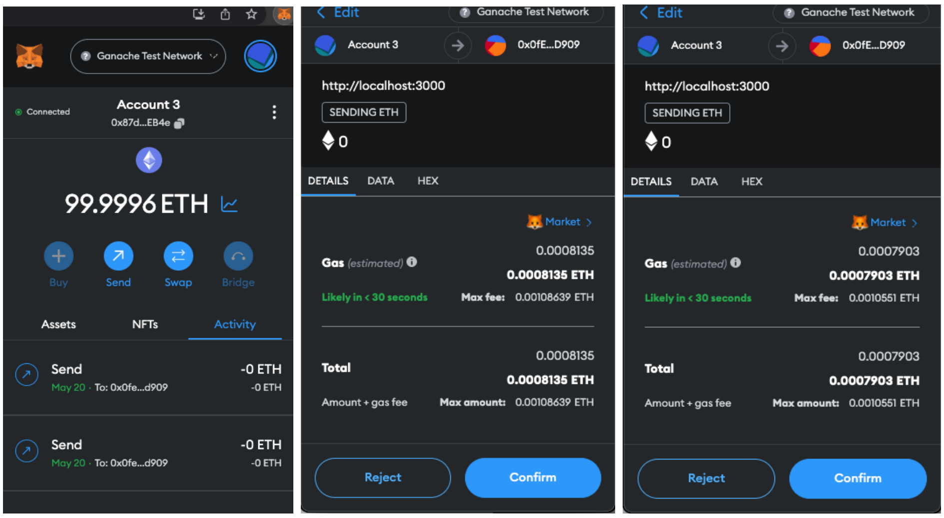
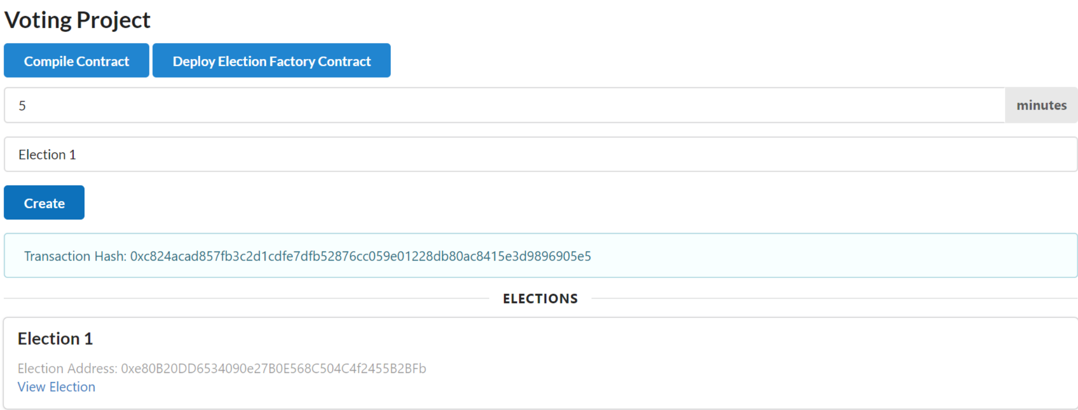


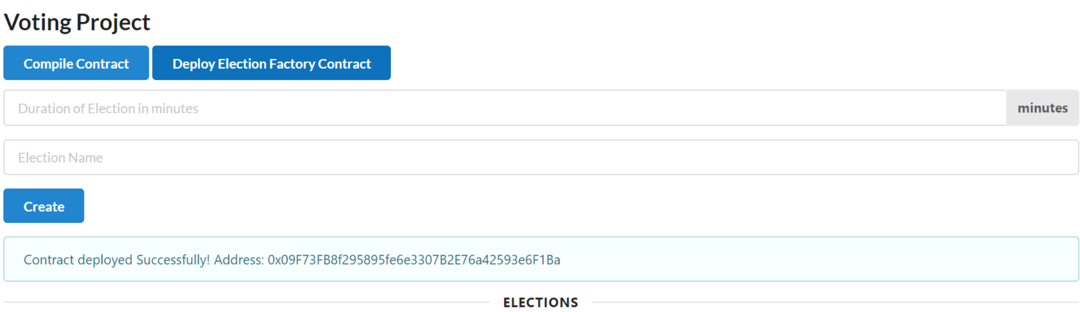
Figure 14, 15, and 16: Illustrating MetaMask as an intermediary connecting the browser and the Ethereum network for smart contract invocation.

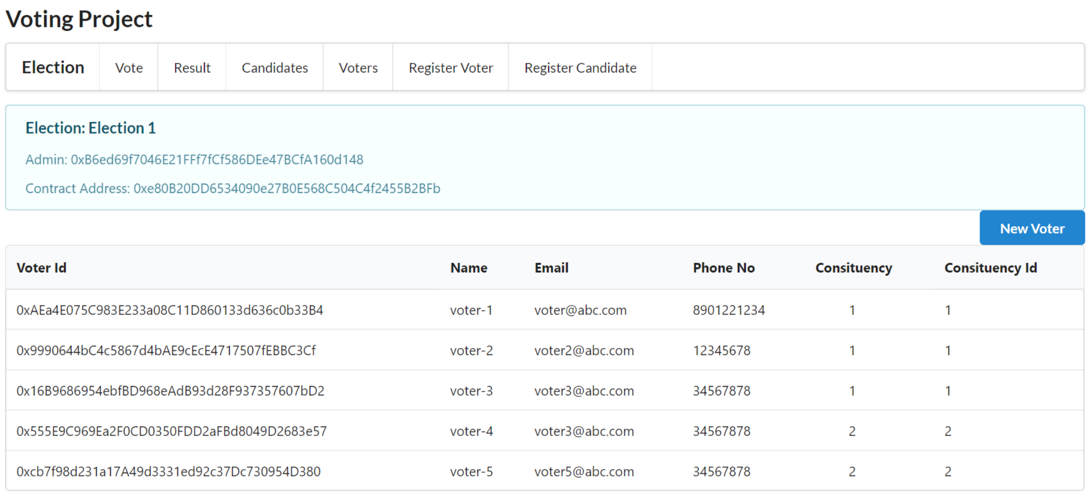
After submitting a transaction, the web application can monitor events emitted by the smart contract to track the progress and completion of the transaction. These events provide information such as the transaction hash, block confirmation, or any additional data emitted by the contract during transaction execution.

Figure: Illustrating the results of smart contract invocation on the Ganache local blockchain network.

Following is the process of initiating a new election involving providing a name and specifying the duration, registering candidates and voters, and closing the election. The workflow commences with compiling and deploying the smart contract.







Voting Project

Election | Vote | Result | Candidates | Voters | Register Voter | Register Candidate

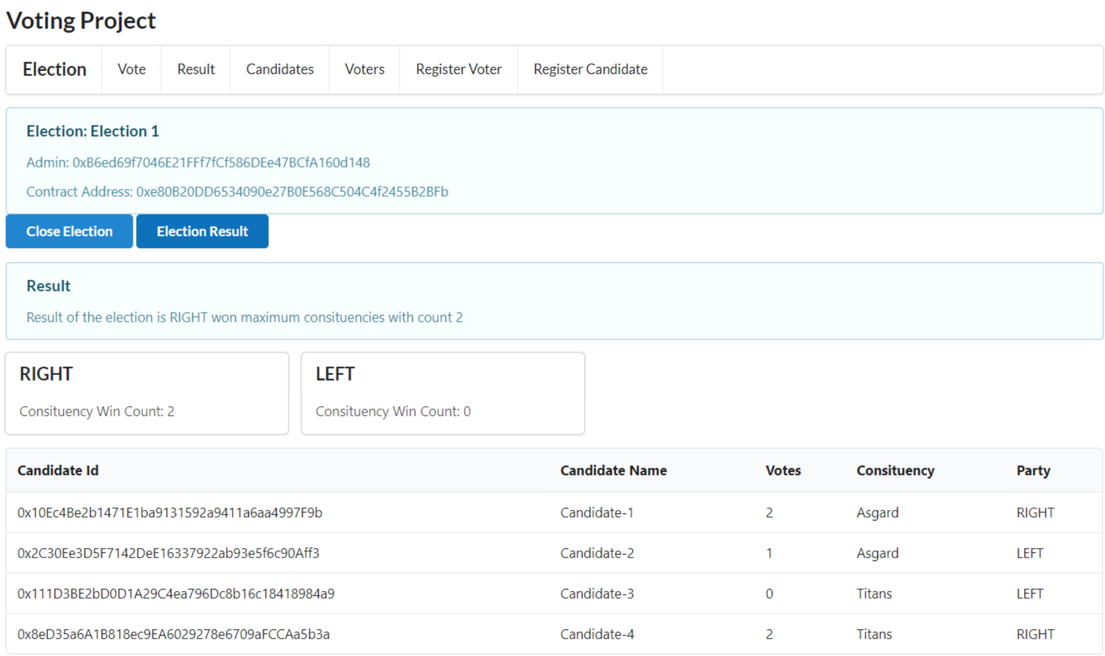
Election: Election 1

Admin: 0xB6ed69f7046E21FFf7fCf586DEe47BCfA160d148

Contract Address: 0xe80B20DD6534090e27B0E568C504C4f2455B2BFb

New Voter

Voter Id	Name	Email	Phone No	Consituency	Consituency Id
0xAEa4E075C983E233a08C11D860133d636c0b33B4	voter-1	voter@abc.com	8901221234	1	1
0x9990644bC4c5867d4bAE9cEcE4717507fEBBC3Cf	voter-2	voter2@abc.com	12345678	1	1
0x16B9686954ebfBD968eAdB93d28F937357607bD2	voter-3	voter3@abc.com	34567878	1	1
0x555E9C969Ea2F0CD0350FDD2aFBd8049D2683e57	voter-4	voter3@abc.com	34567878	2	2
0xcb7f98d231a17A49d3331ed92c37Dc730954D380	voter-5	voter5@abc.com	34567878	2	2



Voting Project

Election | Vote | Result | Candidates | Voters | Register Voter | Register Candidate

Election: Election 1

Admin: 0xB6ed69f7046E21FFf7fCf586DEe47BCfA160d148

Contract Address: 0xe80B20DD6534090e27B0E568C504C4f2455B2BFb

Close Election | Election Result

Result

Result of the election is RIGHT won maximum consituencies with count 2

RIGHT

Consituency Win Count: 2

LEFT

Consituency Win Count: 0

Candidate Id	Candidate Name	Votes	Consituency	Party
0x10Ec4Be2b1471E1ba9131592a9411a6aa4997F9b	Candidate-1	2	Asgard	RIGHT
0x2C30Ee3D5F7142DeE16337922ab93e5f6c90Aff3	Candidate-2	1	Asgard	LEFT
0x111D3BE2bD0D1A29C4ea796Dc8b16c18418984a9	Candidate-3	0	Titans	LEFT
0x8eD35a6A1B818ec9EA6029278e6709aFCCAa5b3a	Candidate-4	2	Titans	RIGHT

Subsequently, constituencies are established by specifying their unique ID and name. Additional candidates and voters can be registered, and their details can be accessed as required. During the voting phase, votes are cast by selecting a voter, resulting in the display of the corresponding candidates for the chosen constituency. The election concludes with the closure of the voting process, and the final election result is obtainable.

The demonstration successfully presents a comprehensive overview of the necessary steps involved in organizing an election, effectively highlighting the system's functionalities and the seamless flow of information throughout the entire process.

Chapter 5: Conclusion

The functionality testing and verification conducted in this technical report have provided evidence of the successful primary utilization of the smart contract in a blockchain-based voting application. By deploying the smart contract onto a blockchain network and integrating it with a web application interface, users can interact with the contract's functions in a reliable manner. The authentication mechanism ensures that only authorized users can access specific features, enhancing the overall security of the system. Through the invocation of smart contract functions, users can retrieve data, modify the contract's state, and actively participate in the voting process. The integration of tools like MetaMask streamlines the transaction signing process, ensuring that user interactions with the smart contract are authenticated and remain unaltered. The functionality testing phase has affirmed the effective functioning of the smart contract and the associated web application, providing a trustworthy and transparent platform for conducting secure elections.

To summarize, this technical report has presented the design and implementation of a blockchain-based voting application using smart contracts. The report has covered system architecture, design and functionality of the smart contract, integration of a user-friendly web application interface, and the testing and verification process. By leveraging the blockchain network, the voting application ensures transparency and immutability throughout the voting process, safeguarding the integrity of election results. The application's authentication mechanisms and security measures protect user information and maintain confidentiality. The successful completion of the functionality testing phase confirms the system's ability to facilitate secure and transparent elections through the use of smart contracts. This project has contributed to the foundation for future enhancements and real-world deployment of the voting application.

References

Jeamwatthanachai, W. (2023). *Write Smart Contracts for Political Voting and Polling Platform*. In CSS486 Blockchain Development (Chapter 10).