Experiment No.6

Aim: To implement a basic voting mechanism within a smart contract, allowing users to vote for predefined candidates. (Voting Contract)

I. **Objectives:**

A. Introduce the concept of state variables and mappings in Solidity.

B. Demonstrate the use of functions to modify and retrieve the contract's state.

C. Highlight the importance of contract security and input validation.

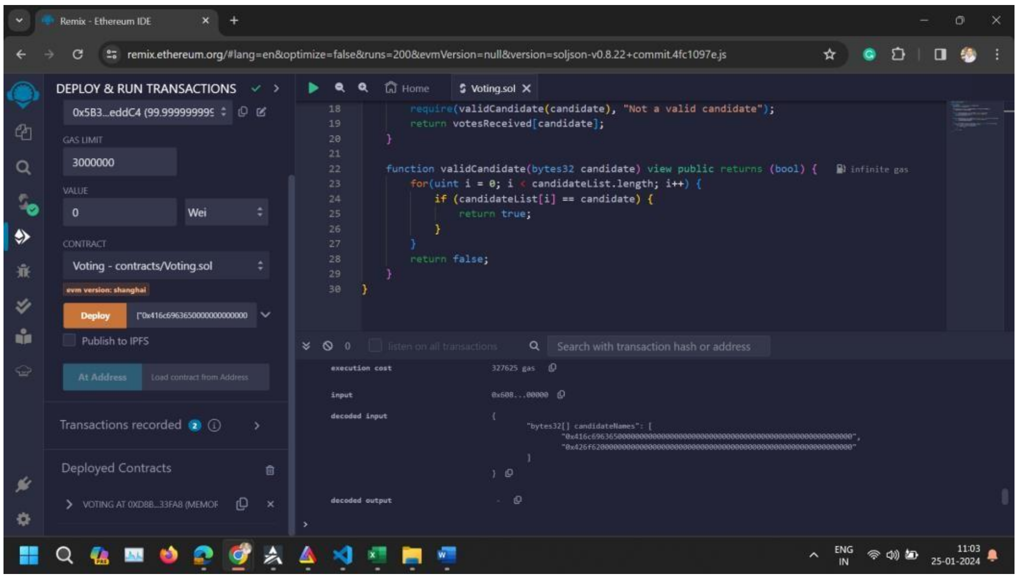
II. **Steps:**

1. **Access Remix IDE:** Navigate to Remix IDE using your web browser.
2. **Create the Contract File:**
   - In the "File Explorers" tab on the left sidebar, create a new file by clicking the "Create New File" icon. Name your file Voting.sol.
   - Paste the Voting contract code into Voting.sol:

```
// Specifies the license under which this contract is released, MIT in this case
// SPDX-License-Identifier: MIT

// Set the compiler version to be used for this contract
pragma solidity ^0.8.22;

// Declares a new contract named Voting
contract Voting {
    // A mapping to keep track of each candidate's vote count, using their name (bytes32) as the key
    mapping(bytes32 => uint256) public votesReceived;

    // An array to store the list of candidates eligible for voting
    bytes32[] public candidateList;

    // Constructor to initialize the contract with a list of candidate names
    constructor(bytes32[] memory candidateNames) {
        candidateList = candidateNames;
    }

    // Function to record a vote for a candidate
    function voteForCandidate(bytes32 candidate) public {
        // Check if the candidate is valid before allowing a vote
        require(validCandidate(candidate), "Not a valid candidate");
        // Increment the vote count for the specified candidate
        votesReceived[candidate] += 1;
    }
    // Function to retrieve the total votes a candidate has received
    function totalVotesFor(bytes32 candidate) view public returns
(uint256) {
        // Ensure the candidate is valid before fetching the vote count
        require(validCandidate(candidate), "Not a valid candidate");
        // Return the vote count for the specified candidate
        return votesReceived[candidate];
    }

    // Helper function to check if a candidate is in the list of valid
candidates
    function validCandidate(bytes32 candidate) view public returns
(bool) {
        // Iterate over the list of candidates
        for(uint i = 0; i < candidateList.length; i++) {
            // If the candidate matches one in the list, return true
            if (candidateList[i] == candidate) {
                return true;
            }
        }
        // If no match is found, return false
        return false;
    }
}
```

3. **Compile the Contract:**
    - Click on the "Solidity compiler" tab on the left sidebar.
    - Ensure the compiler version matches the version specified in your contract (^0.8.22). Remix usually auto-detects the correct version. If not, select it manually from the "Compiler" dropdown.
    - Click "Compile Voting.sol" to compile your contract.
4. **Deploy the Contract:**
    - Switch to the "Deploy & run transactions" tab.
    - Select an environment. "JavaScript VM" is suitable for testing; it simulates an Ethereum blockchain environment within Remix.
    - Before deploying, you need to provide candidate names as an array. Since Solidity expects bytes32[] for candidate names, you must format the input accordingly. For example, to input candidates "Alice" and "Bob", you would use:

```
["0x416c6963650000000000000000000000000000000000000000000000000
00000","0x426f62000000000000000000000000000000000000000000000000
0000000000"]
```


Remix - Ethereum IDE

remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.22+commit.4fc1097e.js

DEPLOY & RUN TRANSACTIONS

0x5B3...eddC4 (99.99999999

GAS LIMIT: 3000000

VALUE: 0 Wei

CONTRACT: Voting - contracts/Voting.sol

evm version: shanghai

Deploy ["0x416c696365000000000000

Publish to IPFS

At Address | Load contract from Address

Transactions recorded 2

Deployed Contracts

VOTING AT 0XD8B...33FA8 (MEMOR

```
        require(validCandidate(candidate), "Not a valid candidate");
        return votesReceived[candidate];
    }

    function validCandidate(bytes32 candidate) view public returns (bool) {    infinite gas
        for(uint i = 0; i < candidateList.length; i++) {
            if (candidateList[i] == candidate) {
                return true;
            }
        }
        return false;
    }
}
```

```
execution cost          327625 gas
input                   0x608...00000
decoded input           {
                            "bytes32[] candidateNames": [
                                "0x416c696365000000000000000000000000000000000000000000000000000000",
                                "0x426f62000000000000000000000000000000000000000000000000000000000000"
                            ]
                        }
decoded output          -
```

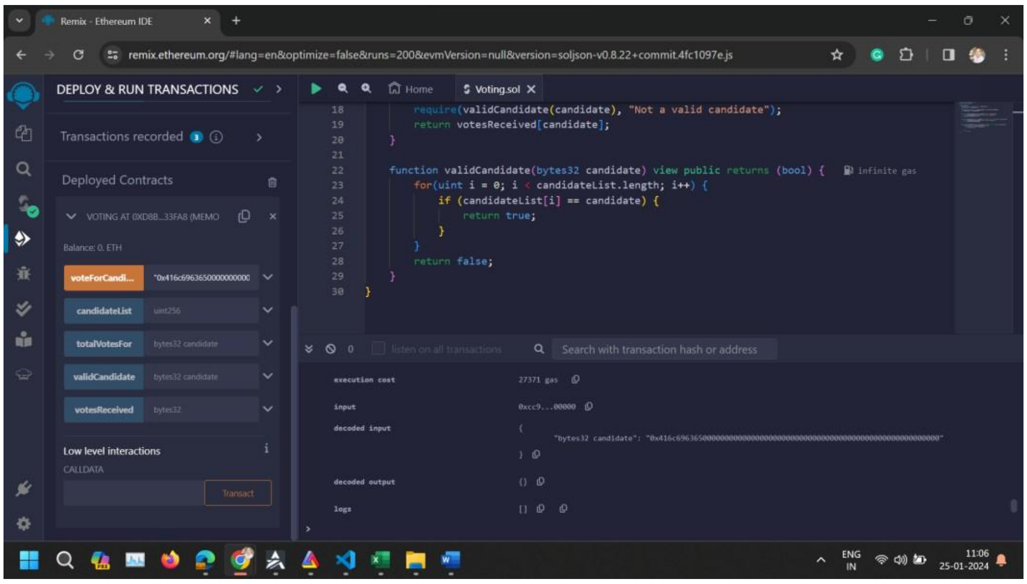
- These are hexadecimal representations of "Alice" and "Bob" in bytes32 format. You can use online tools to convert strings to bytes32 hex format.
- Paste the bytes32 array into the "Deploy" field that appears after selecting the "Voting" contract.
- Click "Deploy". Your contract is now deployed, and you'll see it under "Deployed Contracts".

5. Interact with the Contract:

- To vote for a candidate, expand your deployed contract in the "Deployed Contracts" section, find the voteForCandidate function, and input the bytes32 representation of the candidate's name. For example, to vote for Alice, you would input:

```
"0x416c6963650000000000000000000000000000000000000000000000000000
0000"
```

- Click the button to call voteForCandidate.
- To check the total votes for a candidate, use the totalVotesFor function with the candidate's bytes32 representation as the argument.
- To verify a candidate is valid, use the validCandidate function similarly.



```
        require(validCandidate(candidate), "Not a valid candidate");
        return votesReceived[candidate];
    }

    function validCandidate(bytes32 candidate) view public returns (bool) {
        for(uint i = 0; i < candidateList.length; i++) {
            if (candidateList[i] == candidate) {
                return true;
            }
        }
        return false;
    }
}
```

```
execution cost      27371 gas
input               0xcc9...00000
decoded input       {
                        "bytes32 candidate": "0x416c696365000000000000000000000000000000000000000000000000000000"
                    }
decoded output      {}
logs                []
```

Remix - Ethereum IDE
remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.22+commit.4fc1097e.js
DEPLOY & RUN TRANSACTIONS
Transactions recorded 3
Deployed Contracts
VOTING AT 0XD88...33FA8 (MEMO
Balance: 0. ETH
voteForCandi... "0x416c6963650000000000
candidateList uint256
totalVotesFor bytes32 candidate
validCandidate bytes32 candidate
votesReceived bytes32
Low level interactions
CALLDATA
Transact
Home
Voting.sol
infinite gas
listen on all transactions
Search with transaction hash or address
ENG IN
11:06
25-01-2024

III. **Conclusion:** The voting contract experiment illustrates how Ethereum can be used for decentralized decision-making processes. It demonstrates how to deploy and interact with contracts that require more complex inputs and how to manage state within a contract. By deploying this contract, you've taken a step further into Ethereum development, learning to work with mappings, arrays, and user inputs. It also underscores the critical aspects of smart contract development, such as data storage, function execution, and access control.