Smart Contracts Exercise 02: Decentralized Voting System

1 Introduction

In this exercise, you will implement a smart contract of a decentralized voting system on the blockchain. The goal of this exercise is to familiarize yourself with the basics of the Solidity language.

Project Setup

You have two options for working with this exercise. Using docker container or local installation. Choose the one that best fits your preferences.

1.1 Using Docker with VS Code

This option uses Docker to create a development environment with all the necessary tools and dependencies pre-installed.

Prerequisites:

- **Docker** - A platform for developing, shipping, and running applications in containers.
- **Visual Studio Code** - A lightweight but powerful source code editor.
- **Dev Containers** - An extension to VS Code that lets you use a Docker container as a full-featured development environment.

Setting Up the Project:

1. Visit the following GitLab repository and clone it to your local machine.
2. Open the repository folder in VS Code.
3. When prompted, click "Reopen in Container" or use the command palette (F1) and run `Dev Containers: Reopen in Container`.

1.2 Local Setup

If you prefer working directly on your machine without Docker, you can set up the development environment locally.

Prerequisites

- **Node.js** - https://nodejs.org/en/ - An open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser.
- **NPM**: Node Package Manager, which comes with Node.js.

Open your terminal and run the following commands to verify the installations:

```
$ node -v
$ npm -v
```

Both commands should return the installed version numbers of Node.js and NPM respectively. Node.js provides the runtime environment required to execute JavaScript-based tools like Hardhat, while NPM is used to manage the packages and dependencies needed for development.

Setting Up the Project

1. Visit the following GitLab repository and clone it to your local machine.
2. Open a terminal and navigate to the project directory.
3. Install the project dependencies by running `npm install`.

2 Task Specification: Voting Contract

Your implementation will be in the file `contracts/Voting.sol`. In this file, there are #TODO comments where you should implement the required functionality. To fulfill this task, you need to pass all the provided tests. You can run the tests with the following command:

```
$ npx hardhat test
```

There is also a deployment script in the `scripts` folder. You can deploy the contract to the local Hardhat network with the following command:

```
$ npx hardhat run scripts/deploy.js
```

2.1 Overview

The **Voting** contract is a simple implementation of a voting system using Solidity. It allows the contract owner to add candidates, and any address to vote exactly once for a candidate. The contract includes the following functionalities:

- The contract owner can add candidates.
- Any address can vote exactly once for a candidate.
- The contract tracks the number of votes each candidate has received.

- The contract tracks whether an address has already voted.
- A function to get the total number of candidates.
- A function to retrieve a candidate's name and vote count by index.
- A function to get the index of the winning candidate.

2.2 Solidity Crash Course

The **Voting** contract is designed to facilitate a decentralized voting system. Below are some solidity code snippets that you might find useful for implementing the contract and explanations of the key components.

State Variables

State variables are used to store data permanently on the blockchain. They represent the contract's state and can be accessed and modified by the contract's functions.

```
// Address of the contract owner
address public owner;

// Dynamic array to store all candidates
Candidate[] public candidates;

// Mapping to track whether an address has already voted
mapping(address => bool) public hasVoted;
```

Structs

Structs are custom data types that allow you to group related data together. They are useful for organizing complex data structures within the contract.

```
/**
* @dev Struct to represent a candidate.
* @param name The name of the candidate.
* @param voteCount The number of votes the candidate has received.
*/
struct Candidate {
    string name;
    uint voteCount;
}
```

Constructor

The constructor is a special function that runs once during the contract's deployment and cannot be invoked later.

```
constructor() {
  // The deployer of the contract is the owner
  owner = msg.sender;
}
```

Events

Events are used to log information on the blockchain that can be accessed by off-chain applications. They are essential for tracking contract activities and facilitating interactions with the user interface.

```
 /**
  * @dev Event emitted when a vote is cast.
  * @param voter The address of the voter.
  * @param candidateIndex The index of the candidate voted for.
  */
event Voted(address indexed voter, uint indexed candidateIndex);

/**
  * @dev Event emitted when a new candidate is added.
  * @param name The name of the candidate to be added.
  */
event CandidateAdded(string name);
```

Errors

Errors allow developers to provide more information to the caller about why a condition or operation failed. Errors are used together with the revert statement or require function. On failure, they abort and revert **all** changes made by the transaction.

```
 /// Only the owner can call this function.
 error NotOwner();
 /// The candidate name cannot be empty.
 error EmptyCandidateName();

 // revert if condition is not met
 require(msg.sender == owner, NotOwner());

 // revert statement
 revert EmptyCandidateName();
```

Modifiers

Modifiers are used to change the behavior of functions in a declarative way. They can enforce rules or conditions before executing a function's code.

```
 // Modifier to restrict access to the contract owner
 modifier onlyOwner() {
     require(msg.sender == owner, NotOwner());
     _; // Continue executing the function
 }

 function addCandidate(string memory name) public onlyOwner {
   // Only the contract owner can call this function
 }
```

Functions

Functions define the behavior of the contract. They can read and modify the contract's state, perform computations, and interact with other contracts or external systems.

Useful Code Snippets

Here are some useful code snippets you might need:

```
// Sender of the transaction
address sender = msg.sender;

// Amount sent with the transaction
uint amount = msg.value;

// Enforcing conditions
require(condition, CustomError());

// Casting arbitrary data to uint
uint number = uint(data);

// Empty address
address emptyAddress = address(0);

// Emit an event
emit EventName(parameters);
```

3 More Solidity Concepts

This extra section covers additional Solidity concepts that are useful for developing smart contracts, including more detailed function examples, visibility, and advanced data types. It is not needed to complete the exercise.

Function Types and Visibility

Functions in Solidity can have different visibility modifiers that determine how and where they can be called from:

```
// Public functions can be called internally or via messages
function publicFunction() public returns (uint) {
  return 1;
}
// Private functions can only be called from within this contract
function privateFunction() private returns (uint) {
  return 2;
}
// Internal functions can be called internally or by derived contracts
function internalFunction() internal returns (uint) {
  return 3;
}
// External functions can only be called from other contracts
function externalFunction() external returns (uint) {
  return 4;
}
// View functions promise not to modify the state
function viewFunction() public view returns (uint) {
  return someStateVariable;
}
// Pure functions promise not to modify or read from the state
function pureFunction(uint a, uint b) public pure returns (uint) {
  return a + b;
```

```
}
```

Function Modifiers with Parameters

Modifiers can also accept parameters, making them more flexible:

```
// Modifier with parameters
modifier onlyRole(bytes32 role) {
  require(hasRole(role, msg.sender), "Caller does not have the required role");
  _;
}
// Using the modifier with different roles
function adminFunction() public onlyRole(ADMIN_ROLE) {
// Only addresses with admin role can call this
}
function moderatorFunction() public onlyRole(MODERATOR_ROLE) {
// Only addresses with moderator role can call this
}
```

Advanced Data Structures

Solidity supports several advanced data structures that help organize complex data:

```
// Nested mappings for complex relationships
mapping(address => mapping(uint => bool)) public userPermissions;
// Set a permission
function setPermission(address user, uint permissionId, bool value) public {
  userPermissions[user][permissionId] = value;
}
// Arrays with push and pop operations
uint[] public values;
function addValue(uint value) public {
  values.push(value);
}
function removeLastValue() public {
  values.pop();
}
// Enums for named constants
enum Status { Pending, Active, Inactive, Completed }
Status public currentStatus;
function setStatus(Status newStatus) public {
  currentStatus = newStatus;
}
```

Memory Management in Solidity

```
// Storage - persisted between function calls (expensive)
uint[] public storageArray;
// Memory - temporary during function execution (cheaper)
function processArray(uint[] memory memoryArray) public {
  // This array exists only during function execution
  uint[] memory tempArray = new uint;

  for(uint i = 0; i < memoryArray.length; i++) {
    tempArray[i] = memoryArray[i] * 2;
  }
```

```
  // Store results in contract storage if needed
  for(uint i = 0; i < tempArray.length; i++) {
    storageArray.push(tempArray[i]);
  }
}
// Calldata - read-only, non-modifiable location (most gas efficient)
function readOnlyProcess(uint[] calldata calldataArray) external {
  // Can read from but not modify calldataArray
  for(uint i = 0; i < calldataArray.length; i++) {
    // Process without modifying
  }
}
```

Error Handling

Solidity provides several mechanisms for error handling:

```
// Using require for input validation
function transfer(address to, uint amount) public {
  require(to != address(0), "Cannot transfer to zero address");
  require(amount > 0, "Amount must be greater than zero");
  require(balances[msg.sender] >= amount, "Insufficient balance");

  balances[msg.sender] -= amount;
  balances[to] += amount;
}

// Using assert for internal consistency checks
function internalOperation(uint a, uint b) private {
  uint result = a + b;
  // Should never happen if our code is correct
  assert(result >= a && result >= b); // Check for overflow
}

// Custom errors (more gas efficient than string messages)
error InsufficientBalance(address user, uint available, uint required);

function withdraw(uint amount) public {
  if (balances[msg.sender] < amount) {
    revert InsufficientBalance(msg.sender, balances[msg.sender], amount);
  }

  balances[msg.sender] -= amount;
  payable(msg.sender).transfer(amount);
}
```

Gas Optimization Techniques

Gas optimization is crucial for cost-effective smart contracts:

```
// Use uint256 instead of smaller sizes (usually)
uint256 public largeNumber; // Often cheaper than uint8, uint16, etc.

// Pack variables that are used together
struct PackedData {
  uint128 firstValue;  // These two uint128 variables
```

```
  uint128 secondValue; // will fit in a single storage slot
}

// Cache array length in loops
function processItems() public {
  uint length = items.length; // Read once
  for(uint i = 0; i < length; i++) {
    // Process items[i]
  }
}

// Use calldata for read-only function parameters
function readOnlyOperation(string calldata text) external pure returns (uint) {
  return bytes(text).length;
}
```

Events and Logging

Events are crucial for offchain services and DApp frontends:

```
// Simple event with basic data
event Transfer(address indexed from, address indexed to, uint amount);
// Event with additional data
event VoteCast(
  address indexed voter,
  uint indexed candidateId,
  uint timestamp,
  string comments
);
function castVote(uint candidateId, string memory comments) public {
  // Process vote...

  // Emit event for off-chain applications to track
  emit VoteCast(msg.sender, candidateId, block.timestamp, comments);
}
// Events for multi-step processes
event ProcessStarted(uint indexed processId, address initiator);
event ProcessStep(uint indexed processId, uint step, string description);
event ProcessCompleted(uint indexed processId, bool success);
function runProcess(uint processId) public {
  emit ProcessStarted(processId, msg.sender);

  // Step 1
  emit ProcessStep(processId, 1, "Validation");
  // ... processing ...

  // Step 2
  emit ProcessStep(processId, 2, "Calculation");
  // ... processing ...

  emit ProcessCompleted(processId, true);
}
```

Inheritance and Contract Interaction

Solidity supports contract inheritance and interfaces:

```
// Base contract
contract Ownable {
  address public owner;

  constructor() {
    owner = msg.sender;
  }

  modifier onlyOwner() {
    require(msg.sender == owner, "Not owner");
    _;
  }

  function transferOwnership(address newOwner) public onlyOwner {
    require(newOwner != address(0), "New owner cannot be zero address");
    owner = newOwner;
  }
}
// Derived contract
contract VotingEnhanced is Ownable {
  // Inherits all functions and modifiers from Ownable

  function addCandidate(string memory name) public onlyOwner {
    // Only the owner can add candidates
  }
}
// Interface definition
interface IVoting {
  function vote(uint candidateId) external;
  function getCandidateCount() external view returns (uint);
  function getCandidate(uint index) external view returns (string memory name, uint
    voteCount);
}
// Using another contract
contract VotingClient {
  IVoting public votingContract;

  constructor(address votingAddress) {
    votingContract = IVoting(votingAddress);
  }

  function voteForCandidate(uint candidateId) public {
    votingContract.vote(candidateId);
  }

  function getNumberOfCandidates() public view returns (uint) {
    return votingContract.getCandidateCount();
  }
}
```

Using Modifiers Effectively

Modifiers can be combined and chained for complex access control:

```
// Multiple modifiers
contract VotingWithRoles {
  address public admin;
  mapping(address => bool) public moderators;
```

```
  bool public votingOpen;

  constructor() {
    admin = msg.sender;
  }

  modifier onlyAdmin() {
    require(msg.sender == admin, "Not admin");
    _;
  }

  modifier onlyModerator() {
    require(moderators[msg.sender], "Not moderator");
    _;
  }

  modifier whenVotingOpen() {
    require(votingOpen, "Voting not open");
    _;
  }

  // Function with multiple modifiers
  function emergencyCloseVoting() public onlyAdmin {
    votingOpen = false;
  }

  // Another example with multiple modifiers
  function addCandidate(string memory name) public onlyAdmin whenVotingOpen {
    // Only the admin can add candidates when voting is open
  }

  // Moderators can only count votes when voting is closed
  function countVotes() public onlyModerator {
    require(!votingOpen, "Voting still open");
    // Count votes logic
  }
}
```

To see some more advanced smart contract examples, visit the Solidity by Example section of the Solidity documentation.