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.

Functionality Provided by Solidity

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);
```

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