TASK - 1

Problem Statement:

Create a smart contract which accepts a key value pair as an input (id and string), deploy the smart contract on Polygon using amoy testnet with test matic tokens.

Write a script using web3.js library to interact with blockchain directly rather than using browser wallets (i.e. metamask and etc.)

Steps:

- Setup amoy testnet in your wallet by adding the below given custom network details:

  Network Name: Polygon Amoy Testnet
  New RPC URL: https://rpc-amoy.polygon.technology/
  Chain ID: 80002
  Currency Symbol: MATIC
  Block Explorer URL: https://amoy.polygonscan.com/

- Create a smart contract using the Remix ide and deploy in the injected test environment (Polygon amoy testnet).
- Make sure to copy the abi and deployed contract address.
- Create a basic interface as a part of UI using html with input fields such as ID, String and buttons (Submit and Fetch Data).
- Install web3.js library to interact with blockchain.
- Write a script to directly initiate transactions using web3.js library methods and bypass the browser wallet-based transactions.

Steps to Create the Smart Contract:

1. **Open Remix IDE**: Navigate to Remix IDE in your web browser.
2. **Create a New File**: In the Remix file explorer, click on the "+" icon to create a new file. Name the file Task1.sol.
3. **Write the Smart Contract Code**: Copy and paste the following code into the newly created file:

```
pragma solidity ^0.8.0;

contract Task1 {
    struct Data {
        uint256 id;
        string name;
    }

    mapping(uint256 => Data) public data;

    function addData(uint256 _id, string memory _name) public {
        data[_id] = Data(_id, _name);
    }

    function getData(uint256 _id) public view returns (string memory) {
        Data memory rdata = data[_id];
        return rdata.name;
    }
}
```

   This contract defines a Data structure to hold the ID and string pair and uses a mapping to store these pairs.

4. **Compile the Smart Contract**: Click on the "Solidity Compiler" tab in the left sidebar and select the appropriate compiler version (ensure it matches the pragma version specified in your contract). Click on the "Compile Task1.sol" button. Ensure there are no errors in the compilation process.

5. **Deploy the Contract**: Switch to the "Deploy & Run Transactions" tab. Ensure that the "Environment" is set to "Injected Web3" to interact with the Polygon Amoy Testnet. Select the account you wish to use for deployment.
6. **Deploy**: Click on the "Deploy" button. Confirm the transaction in your wallet when prompted. Your smart contract will be deployed on the Amoy testnet.
7. **Copy ABI and Contract Address**: After deployment, the contract address will be displayed in the Remix interface. Make sure to copy this address along with the ABI, as you will need them to interact with your contract using web3.js.

By following these steps, you will have successfully created and deployed a smart contract on the Polygon Amoy Testnet. The contract allows you to add and retrieve data using an ID, demonstrating the mapping functionality in Solidity.

Understanding the Smart Contract Code

The smart contract code provided in the problem statement implements a simple yet effective way to store and retrieve key-value pairs using Solidity, the programming language for Ethereum-based smart contracts. Below, we will conduct a line-by-line breakdown of the code to understand its structure, data types, and functionalities.

```
pragma solidity ^0.8.0;
```

The first two lines specify the license under which this contract is released and the version of Solidity being used. The pragma directive ensures that the contract is compiled with the specified version (0.8.0 or higher), which includes several features and security improvements.

```
contract Task1 {
```

This line begins the definition of the smart contract named Task1. A contract in Solidity is similar to a class in object-oriented programming and serves as the blueprint for creating instances on the blockchain.

```
    struct Data {
        uint256 id;
        string name;
    }
```

Here, a struct called Data is defined to hold two properties: an unsigned integer (uint256) for the ID and a string for the name. Structs are useful for grouping related data together.

```
    mapping(uint256 => Data) public data;
```

A mapping is declared here, which acts like a hash table or dictionary. It maps a uint256 (the ID) to the Data struct. The public visibility modifier allows external contracts and users to access this mapping, automatically generating a getter function for it.

```
    function addData(uint256 _id, string memory _name) public {
        data[_id] = Data(_id, _name);
    }
```

This function, addData, takes two parameters: _id and _name. It allows users to add a new entry to the mapping. The memory keyword specifies that the string is stored in memory rather than on the blockchain, which is necessary for function parameters. When called, it creates a new Data instance and stores it in the data mapping under the provided _id.

```
    function getData(uint256 _id) public view returns (string memory) {
        Data memory rdata = data[_id];
        return rdata.name;
    }
}
```

The getData function retrieves the name associated with a given ID. It is marked as view, indicating that it does not modify the state of the blockchain. The function fetches the Data struct from the mapping and returns the name field. The retrieved Data is stored in a variable rdata declared in memory.

Overall, this smart contract exemplifies the fundamental concepts of Solidity, including structs, mappings, and function definitions, while enabling the efficient storage and retrieval of data on the blockchain.

Building the User Interface

To create a user-friendly interface for interacting with the smart contract, you will need to utilize HTML along with some basic JavaScript to facilitate the connection to the blockchain via the web3.js library. Below is a step-by-step guide to building this interface, which will include input fields for the ID and string data, as well as buttons to submit new data and fetch existing data from the smart contract.

Step 1: Set Up HTML Structure

Begin by creating a basic HTML file. Inside the <body> tag, you will include input fields for the user to enter the ID and the corresponding string name. Here is an example of the HTML structure:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Smart Contract Interface</title>
  <script src="https://cdn.jsdelivr.net/npm/web3/dist/web3.min.js"></script>
</head>
<body>
  <h1>Smart Contract Interaction</h1>
  <div>
    <label for="id">ID:</label>
    <input type="number" id="id" required>
  </div>
  <div>
    <label for="name">Name:</label>
    <input type="text" id="name" required>
  </div>
  <button id="submitButton">Submit Data</button>
  <button id="fetchButton">Fetch Data</button>
  <div id="result"></div>
</body>
</html>
```

Step 2: Integrate Web3.js

Next, you will need to write JavaScript to handle the interactions with the smart contract. First, initialize web3 and connect to your deployed contract using its ABI and address. You can include the following script at the bottom of your HTML file:

```
<script>
  const web3 = new Web3('https://rpc-amoy.polygon.technology/');
  const contractABI = [/* ABI from Remix IDE */];
  const contractAddress = 'YOUR_CONTRACT_ADDRESS';

  const contract = new web3.eth.Contract(contractABI, contractAddress);

  document.getElementById('submitButton').onclick = async function() {
    const id = document.getElementById('id').value;
    const name = document.getElementById('name').value;
    const accounts = await web3.eth.getAccounts();

    await contract.methods.addData(id, name).send({ from: accounts[0] });
    alert('Data submitted successfully!');
  };

  document.getElementById('fetchButton').onclick = async function() {
    const id = document.getElementById('id').value;
    const result = await contract.methods.getData(id).call();
    document.getElementById('result').innerText = 'Fetched Name: ' + result;
  };
</script>
```

Explanation of JavaScript Code

1. **Web3 Initialization**: The script begins by creating an instance of web3 connected to the Amoy testnet RPC URL.
2. **Contract Interaction**: It sets up the contract interface using the ABI and contract address.
3. **Submit Data Function**: The submitButton event listener retrieves the ID and name from the input fields, fetches the user's account, and sends a transaction to the smart contract to add the data.
4. **Fetch Data Function**: The fetchButton listener calls the smart contract's getData method using the provided ID and displays the fetched name on the web page.

By integrating this HTML and JavaScript code, you will have created a basic interface that allows users to submit and fetch data from the smart contract deployed on the Polygon Amoy Testnet, effectively bridging the gap between users and blockchain technology.

Installing and Using Web3.js

To interact with the Ethereum blockchain, particularly for projects involving smart contracts like the one being developed on the Polygon Amoy Testnet, the web3.js library serves as a crucial tool. This library provides a JavaScript API that allows developers to communicate with Ethereum nodes, facilitating functions such as sending transactions, querying data, and managing smart contracts.

Installation of Web3.js

To get started with web3.js, you need to install it in your project. This can be done easily using npm (Node Package Manager). If you haven't set up a Node.js environment yet, make sure to download and install Node.js from the official website. Once Node.js is installed, open your terminal or command prompt and run the following command:

```
npm install web3
```

This command fetches the web3.js library from the npm repository and adds it to your project. After installation, you can include it in your JavaScript files using:

```
const Web3 = require('web3');
```

If you prefer to use a CDN for quick prototyping, you can include web3.js directly in your HTML file with the following script tag:

```
<script src="https://cdn.jsdelivr.net/npm/web3/dist/web3.min.js"></script>
```

Significance of Web3.js

The primary significance of web3.js lies in its ability to connect your application to the blockchain without relying on traditional wallet interactions, such as MetaMask. While MetaMask provides a user-friendly interface for managing accounts and transactions, web3.js enables developers to bypass these browser wallet dependencies, allowing for a more programmatic approach to blockchain interactions.

Using web3.js, you can initiate transactions directly from your scripts, manage private keys, and execute smart contract functions without needing a browser wallet interface. This capability is particularly valuable for automated systems or backend applications where user interaction with a wallet is impractical.

Bypassing Traditional Wallet Interactions

With web3.js, developers can authenticate and sign transactions programmatically. This is accomplished by using the private key associated with an Ethereum account. For instance, when you want to add data to your smart contract, you would create and sign a transaction object using the private key, allowing you to execute the transaction directly on the blockchain.

Here's a snippet illustrating how to send a transaction:

```
const privateKey = 'YOUR_PRIVATE_KEY';
const account = 'YOUR_ACCOUNT_ADDRESS';
const data = contract.methods.addData(id, name).encodeABI();
const gasPrice = await web3.eth.getGasPrice();
const nonce = await web3.eth.getTransactionCount(account);

const txnObject = {
   to: contractAddress,
   gas: estimatedGasAmount,
   gasPrice: gasPrice,
   nonce: nonce,
   data: data,
};

const signedTxn = await web3.eth.accounts.signTransaction(txnObject, privateKey
);
await web3.eth.sendSignedTransaction(signedTxn.rawTransaction);
```

This example highlights how web3.js facilitates a direct and efficient connection to the blockchain, allowing for seamless interactions with smart contracts without the overhead of a browser wallet.

Writing the Interaction Script

To effectively interact with the deployed smart contract using web3.js, a specific script is crafted to facilitate the sending and fetching of data. This script is essential for ensuring that the smart contract functions as intended without relying on traditional browser wallet methods. Below, we will break down the various components of the script, focusing on key variables and their roles in the interaction process.

Key Variables in the Script

1. **Web3 Instance**: The script begins by creating a new instance of web3, which provides the necessary methods to communicate with the Ethereum blockchain. For example:

   ```
   const web3 = new Web3('https://rpc-amoy.polygon.technology/');
   ```

2. **ABI and Contract Address**: The ABI (Application Binary Interface) is a crucial component that describes the methods and structures of the smart contract. The contract address is the unique identifier for the deployed contract on the blockchain:

   ```
   const contractABI = [/* ABI from Remix IDE */];
   const contractAddress = 'YOUR_CONTRACT_ADDRESS';
   const contract = new web3.eth.Contract(contractABI, contractAddress);
   ```

3. **Private Key**: This variable holds the private key of the wallet that will sign transactions. It is vital for authenticity and must be kept secure:

   ```
   const privateKey = 'YOUR_PRIVATE_KEY';
   ```

4. **Wallet Address**: The wallet address associated with the private key is used to initiate transactions:

   ```
   const walletAddress = 'YOUR_ACCOUNT_ADDRESS';
   ```

5. **Transaction Object**: This object encapsulates all the details needed to execute a transaction, including the recipient address, gas limit, gas price, nonce, and the data payload (method call):

```
const txnObject = {
   to: contractAddress,
   gas: estimatedGasAmount,
   gasPrice: gasPrice,
   nonce: nonce,
   data: data,
};
```

Gas Estimation and Signing Transactions

Gas estimation is a critical step in the transaction process, as it determines the amount of gas required to execute a function on the blockchain. This is done using the web3.eth.getGasPrice() method to fetch the current gas price and web3.eth.getTransactionCount(walletAddress) to obtain the nonce, which is essential for preventing transaction replay attacks.

The signing of the transaction is performed using the private key. The web3.eth.accounts.signTransaction(txnObject, privateKey) method signs the transaction, ensuring that it is legitimate and can be processed by the network.

Sending Transactions

Finally, the signed transaction is sent to the blockchain using:

```
await web3.eth.sendSignedTransaction(signedTxn.rawTransaction);
```

This method broadcasts the transaction, allowing the smart contract to execute the specified function, such as adding data to the mapping.

Retrieving Data

To fetch data from the smart contract, the script utilizes a simple call method:

```
const result = await contract.methods.getData(parameters).call();
```

This line calls the getData function of the smart contract, returning the corresponding name associated with the given ID without altering the blockchain state.

Through this structured approach, the interaction script effectively manages communication with the smart contract, providing a robust method to handle blockchain transactions programmatically.

Testing the Smart Contract

Once the smart contract is deployed and the user interface is built, it is imperative to test the functionalities to ensure everything operates as expected. This section outlines the steps to test the deployed smart contract using the developed interface and interaction script, along with common troubleshooting tips.

Steps to Test the Smart Contract Functionality

1. **Open the User Interface**: Launch the HTML file containing your user interface in a web browser. Ensure that you have a connected wallet that is set to the Polygon Amoy Testnet.
2. **Input Data for Testing**: In the user interface, input a unique ID and a corresponding string name into the provided fields. These inputs will be used to test the addData function of the smart contract.
3. **Submit Data**: Click on the "Submit Data" button. This action should trigger the JavaScript function to send a transaction to the smart contract, storing the ID and string in the mapping. Monitor the console for any errors or confirmations regarding the transaction.
4. **Verify Data Submission**: After submission, utilize the "Fetch Data" button. Enter the same ID you used for submission and click the button. The interface should display the string that you previously submitted. If the correct data is returned, the smart contract is functioning properly.
5. **Repeat Tests**: To ensure robustness, repeat the submission process with different IDs and string values. Check for consistency and correctness in the data retrieval process.

Common Issues and Troubleshooting

- **Transaction Fails**: If submitting data fails, check the browser console for error messages. Common reasons include insufficient gas or incorrect contract address. Ensure that your wallet is funded with test MATIC tokens and that you are using the correct contract address and ABI.

- **Data Not Retrieved**: If the fetched data does not match the submitted data, verify that the ID used to fetch the data is correct. Additionally, check the smart contract's state by reviewing the blockchain explorer for the contract to ensure that the data was indeed added.
- **Network Connectivity Issues**: If the user interface fails to connect to the Amoy Testnet, ensure that you have correctly configured the network settings in your wallet and that the RPC URL is accurate. Sometimes, network congestion may also cause delays in transaction confirmations.
- **JavaScript Errors**: If there are errors in the JavaScript console, inspect your code for syntax errors or misconfigured variables. Ensure that the web3.js library is correctly included, and check that the contract methods are being called with the correct parameters.

By systematically following these testing steps and troubleshooting tips, you can ensure that the smart contract deployed on the Polygon Amoy Testnet operates smoothly and effectively, providing a reliable interface for users to interact with the blockchain.