**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**



**LAB REPORT on**

# Blockchain Technology

***Submitted by***

## Rajdeep Bandyopadhaya (1BM22IC045)

***in partial fulfillment for the award of the degree of***

**BACHELOR OF ENGINEERING**

***in***

## COMPUTER SCIENCE AND ENGINEERING(IoT and Cybersecurity including Blockchain)



**B.M.S. COLLEGE OF ENGINEERING**

**(Autonomous Institution under VTU)**

## BENGALURU-560019 March-2025 to June-2025

**B. M. S. College of Engineering,**

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

### Department of Computer Science and Engineering(IoT and Cybersecurity including Blockchain)



#### CERTIFICATE

This is to certify that the Lab work entitled “Blockchain Technology” carried out by **Rajdeep Bandyopadhaya (1BM22IC0145),** who is a bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering(IoT and Cybersecurity including Blockchain)** of the Visvesvaraya Technological University, Belgaum during the year 2025. The Lab report has been approved as it satisfies the academic requirements in respect of a **Blockchain Technology (23IC6PCBCT)** work prescribed for the said degree.

Krupa K S **Dr. Prasad G R**

Assistant Professor Professor and Head

Department of CSE(ICB) Department of CSE(ICB)

BMSCE, Bengaluru BMSCE, Bengaluru

`

## Index Sheet

|  |  |  |
| --- | --- | --- |
| **Sl.**  **No.** | **Experiment Title** | **Page No.** |
| 1 | Implement a Solidity contract that uses arrays and control structures (if-else, loops). | 1-7 |
| 2 | Write a Solidity contract using different types of functions (view, pure, fallback) and function modifiers. | 8-13 |
| 3 | Install MetaMask, connect to an Ethereum testnet, and interact with a deployed smart contract. | 14-17 |
| 4 | Set up Truffle and Ganache, compile and deploy a Solidity contract on a local blockchain. | 17-19 |
| 5 | Implement a basic peer-to-peer blockchain using socket. | 20-23 |
| 6 | Create and deploy an ERC20 token on Ethereum. Implement a blockchain with a mining process in Python, simulating the Proof-of-Work consensus algorithm | 23-26 |
| 7 | Implement a Merkle tree and basic blockchain focusing on state alteration and block creation. | 26-32 |
| 8 | Implement a blockchain with a mining process in Python, simulating the Proof-of-Work consensus algorithm | 33-36 |

**LAB 2: Control Flow Statements in Solidity**

### BlockChain Lab Report – 01 (Date: 19-03-2025)

#### Objective

The objective of Lab 2 is to understand and implement various control flow statements in Solidity such as if-else , while loops , do-while loops , for loops , break , and continue statements. Students will implement these control structures to solve real-world problems and test their code using Solidity functions.

**Pre-requisites:**

Basic understanding of Solidity programming language. Familiarity with control flow statements (if-else, loops).

A working development environment such as Remix IDE or Hardhat to write and test the Solidity contract.

**Experiment Tasks:**

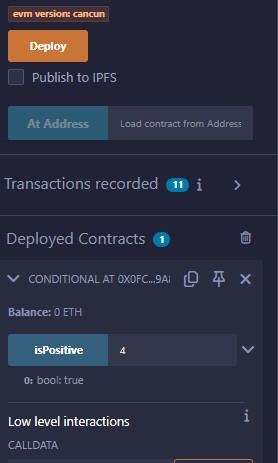
In this lab, we learned:

1. To write Solidity functions using different control flow statements to solve problems.
2. To Test functions to ensure they work correctly by implementing test cases using assertions.
3. Experiment with loops and conditions to see how the logic flows and results are computed.

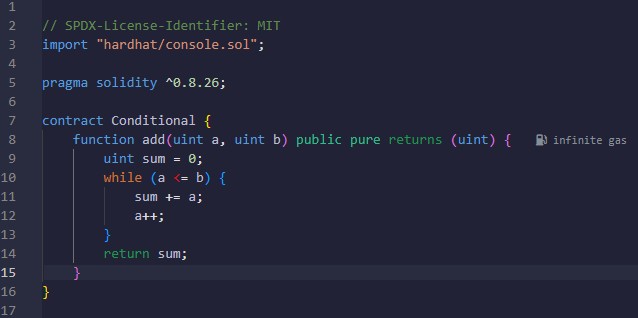
#### 1: isPositive function



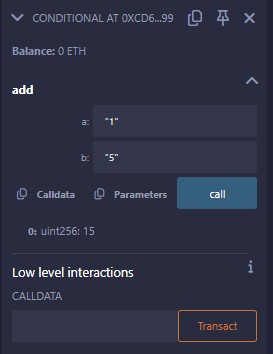
#### OUTPUT



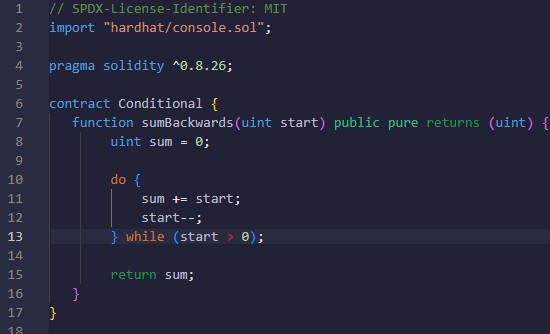
#### Task 2: Implement the recursiveSum function using a while loop



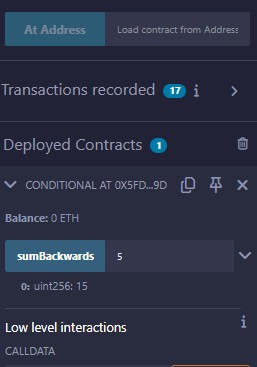
#### OUTPUT



**3: sumBackwards function using a do-while loop**



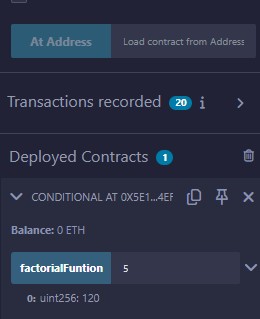
#### OUTPUT



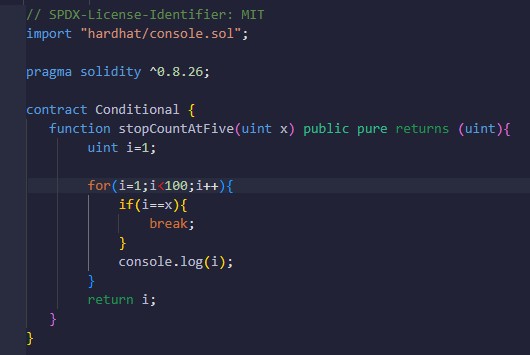
#### Task 4: Implement the factorial Function using a for loop



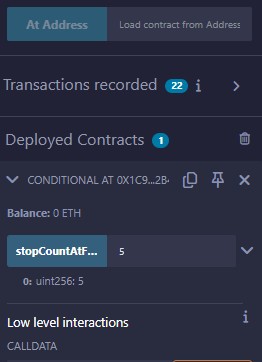
#### OUTPUT



#### 5: stopCountAtFive function using the break statement



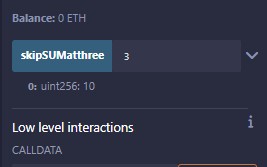
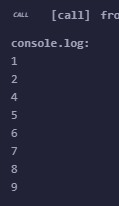
#### OUTPUT



#### Task 6: Implement the skipSumAtThree function using the continue statement



#### OUTPUT



#### Blockchain Lab Report – 03 (Date: 03-04-2025)

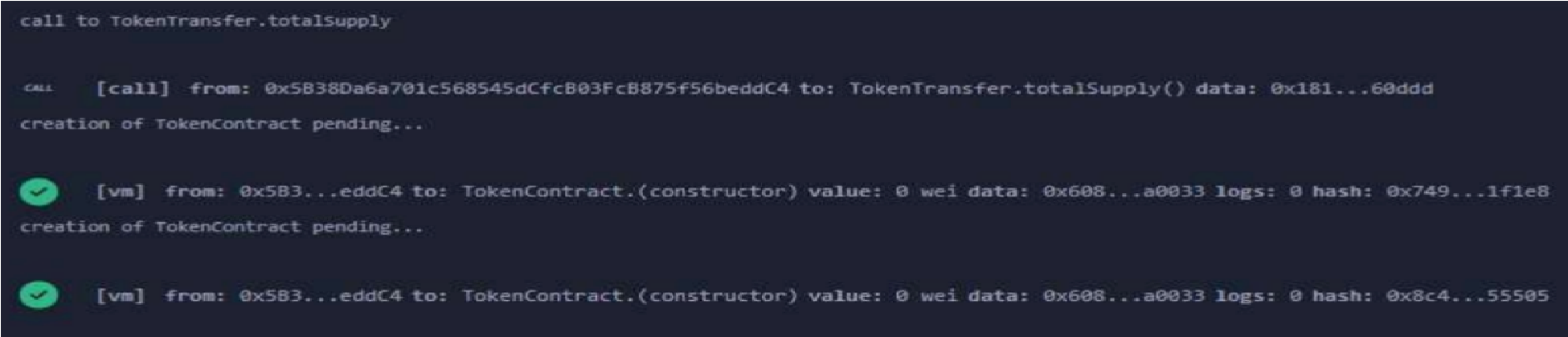
**Token Management Token System in Solidity**

**Introduction:**

In this lab, a secure token management system was implemented, allowing users to deposit tokens into their accounts while restricting token transfers to authorized administrators. The contract integrated essential Solidity function types, including **view**, **pure**, **fallback**, and **payable** functions, along with modifiers for access control and validation. Events were emitted to track transactions and key contract actions.

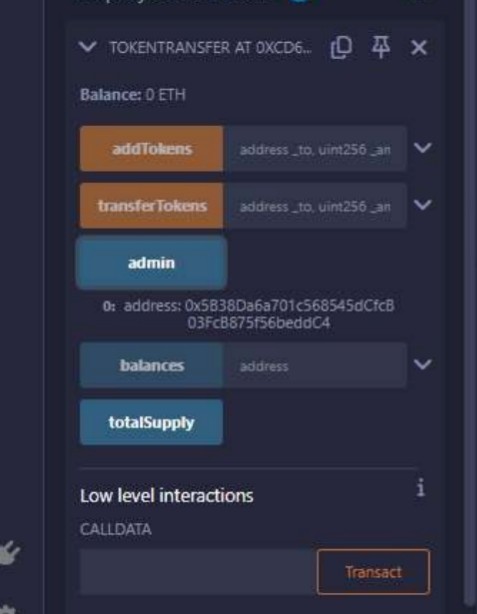
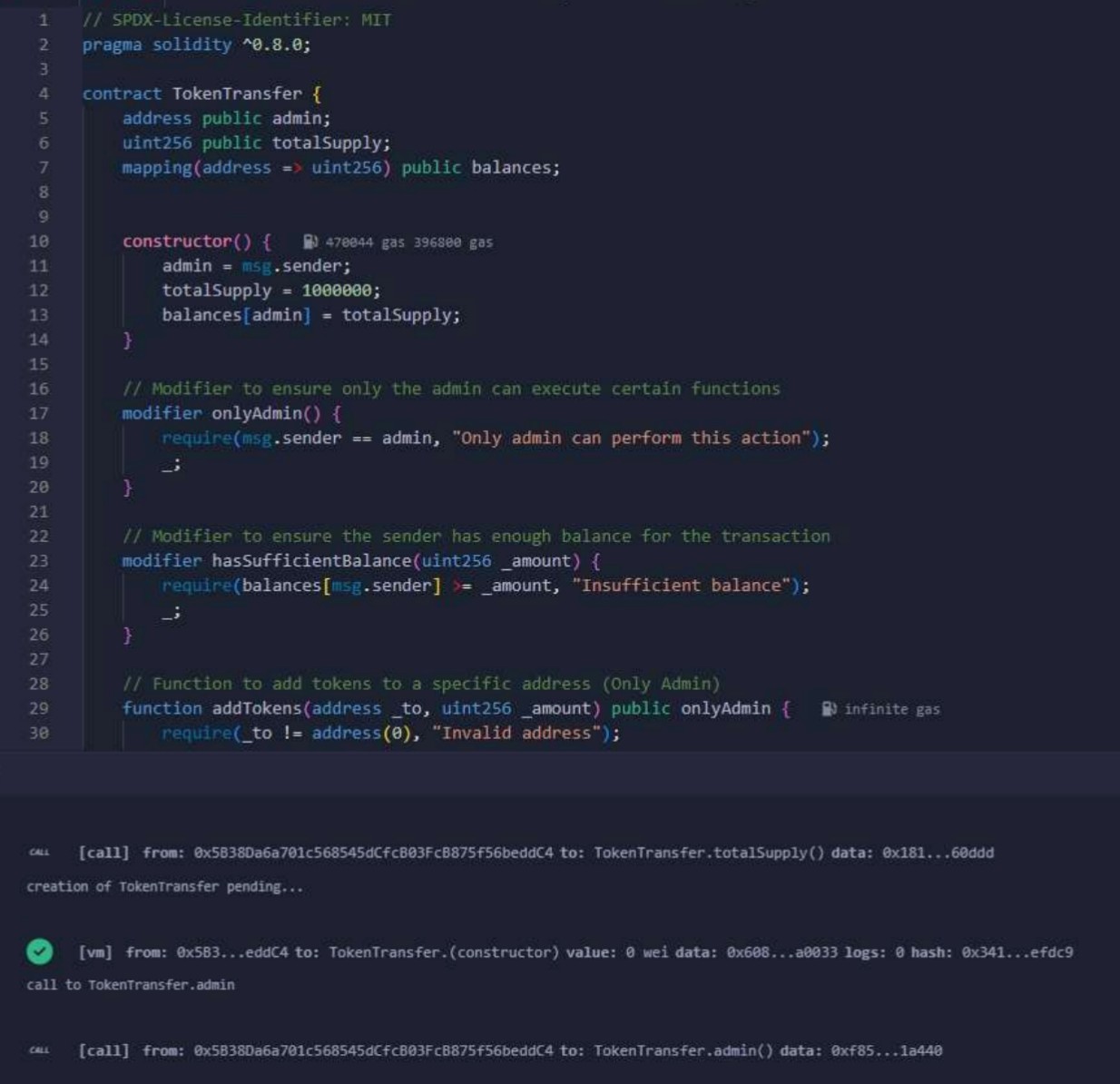
**For Error handling**





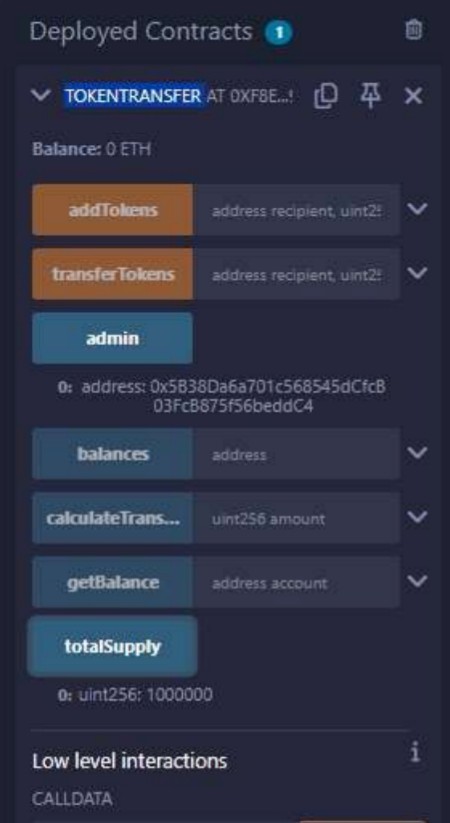
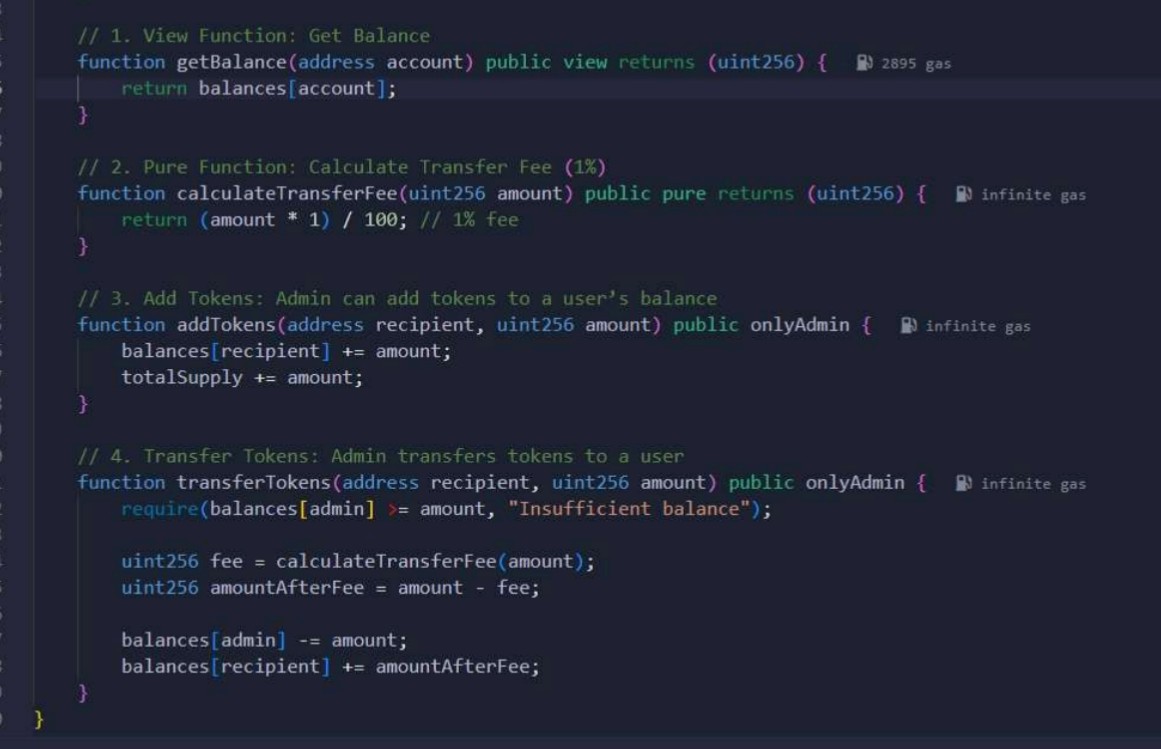
* **Require**
* **Revert**
* **Assert**

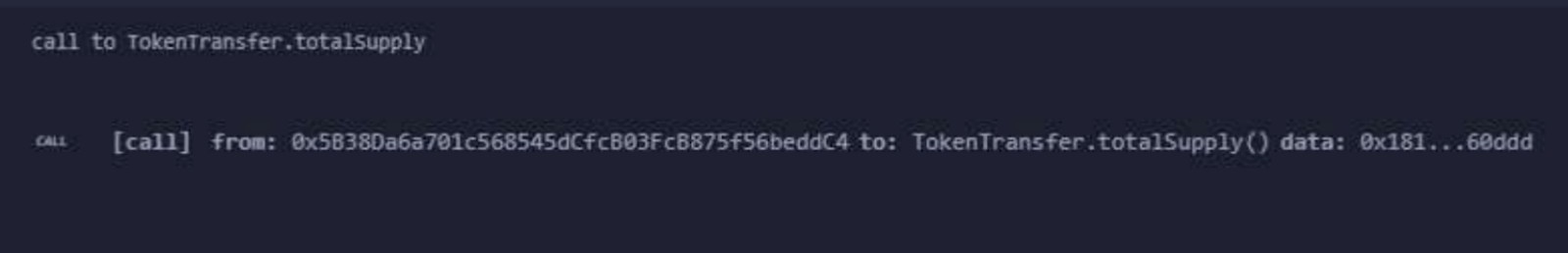
##### Task 1: Contract Setup



* **State Variables:** Declare admin (stores admin address), totalSupply (total tokens), and balances (tracks token holdings).
* **Constructor:** Sets the deployer as admin, initializes totalSupply to 1,000,000, and assigns all tokens to the admin.

##### Task 2: Implement Functions





* **View Function (getBalance):** Returns the balance of a given address from the balances mapping without modifying the blockchain state.

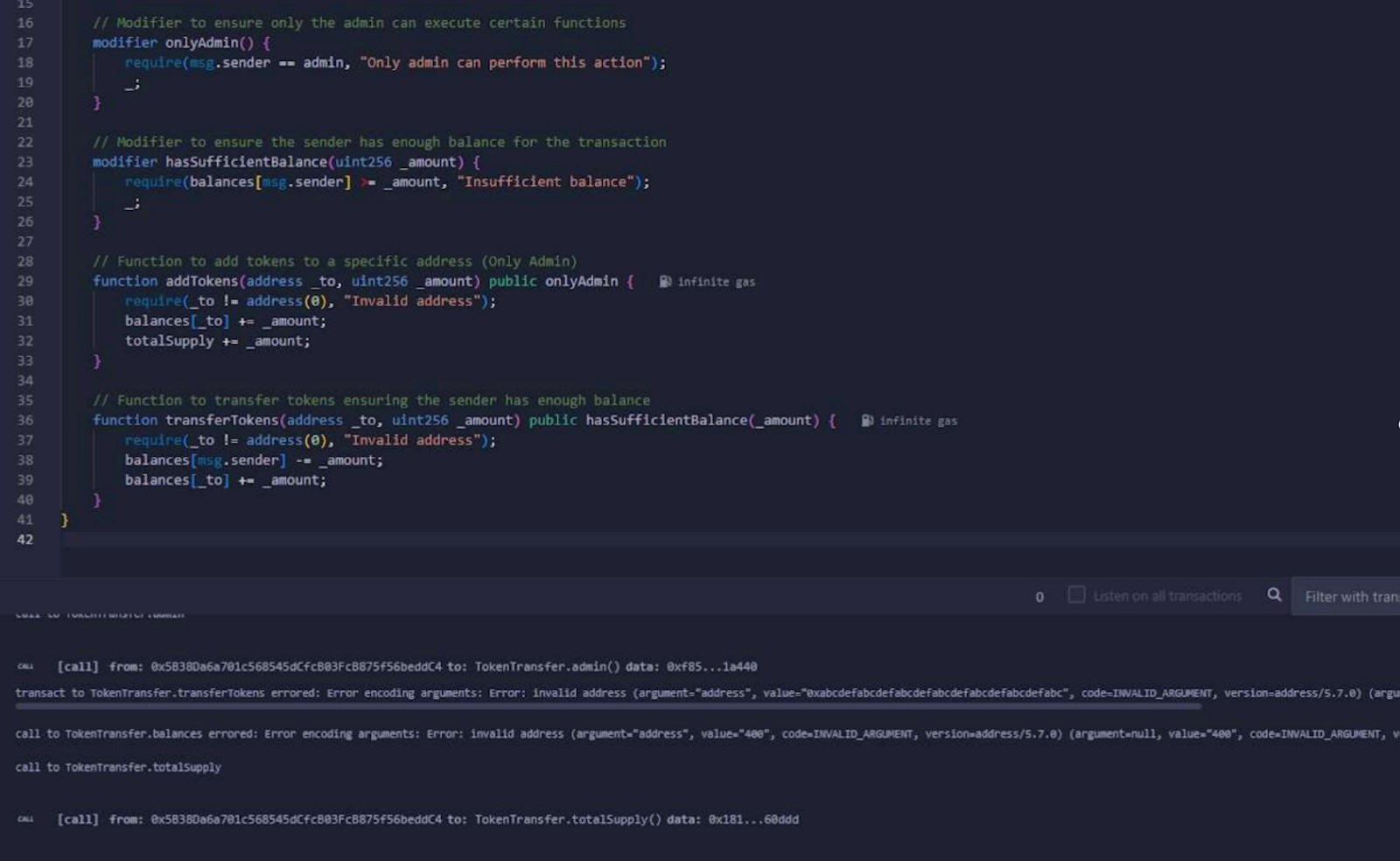
* **Pure Function (calculateTransferFee):** Computes a 1% transfer fee without reading or modifying the blockchain state.

* **Add Tokens (addTokens):** Allows only the admin to add tokens to a user’s balance using an access control modifier.

* **Transfer Tokens (transferTokens):** Lets the admin transfer tokens to a recipient, ensuring the sender has enough balance before proceeding.

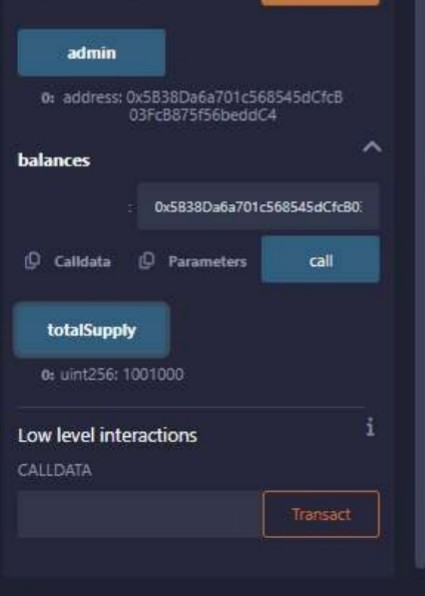
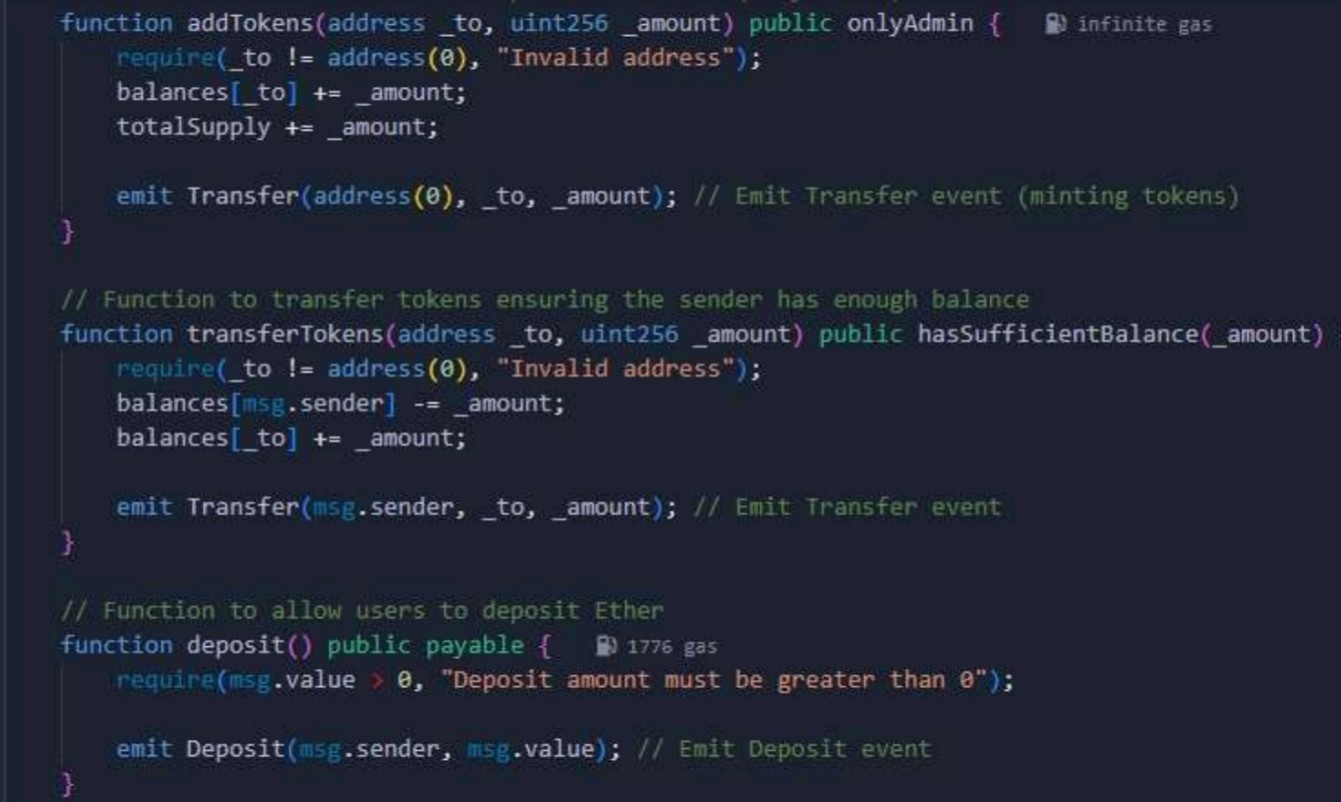
##### Task 3: Implement Modifiers

* **Admin Access Modifier (onlyAdmin):** Restricts access to admin-only functions like addTokens and transferTokens.
* **Balance Validation Modifier (hasSufficientBalance):** Ensures the sender has enough tokens before allowing a transfer, reverting if insufficient.



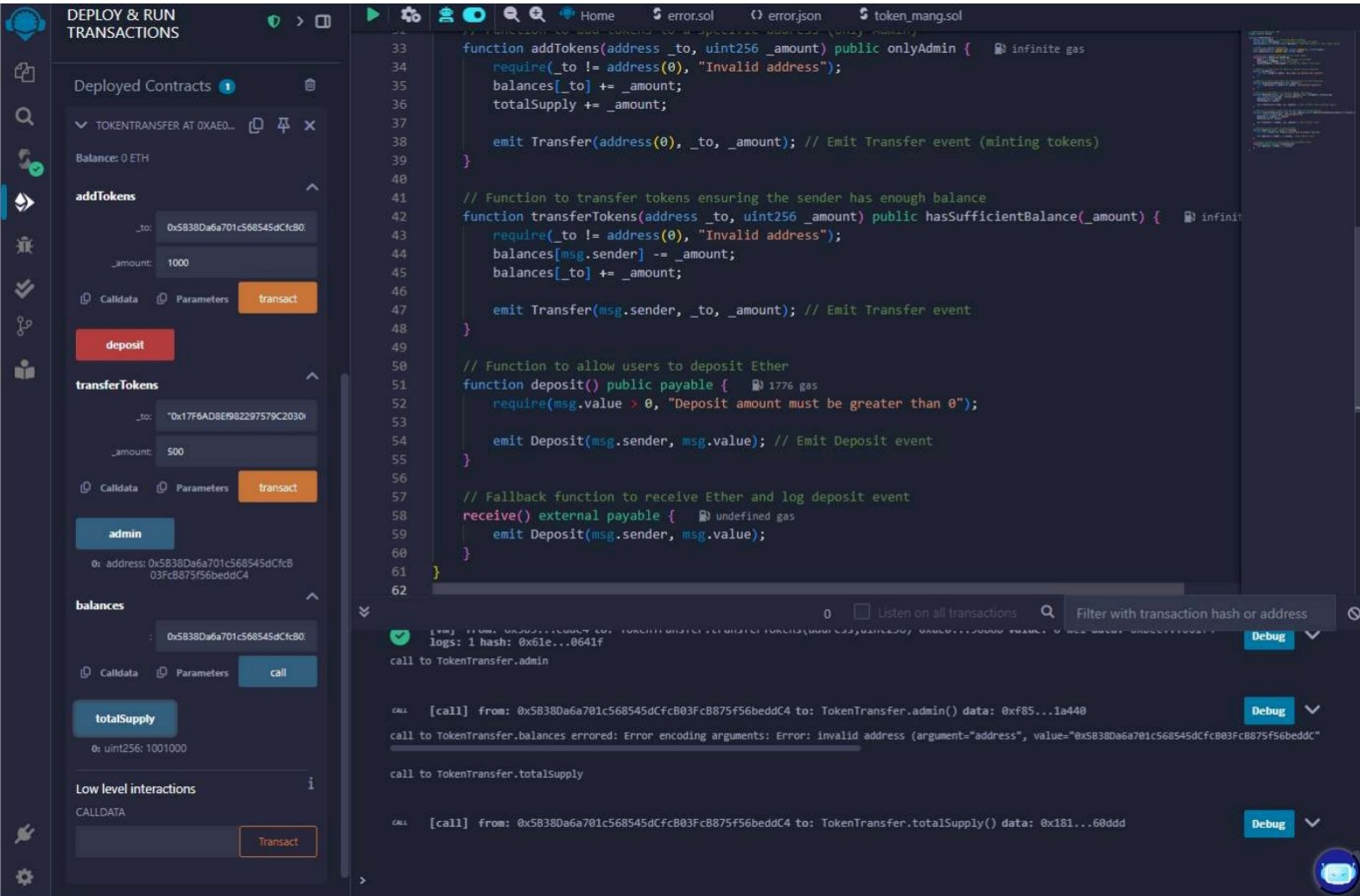
##### Task 4: Implement Events

* **Define Events:** Declare Transfer (logs token transfers) and Deposit (logs Ether deposits) events.
* **Emit Events:** Trigger Transfer in addTokens and transferTokens, and emit Deposit and Transfer when Ether is deposited.

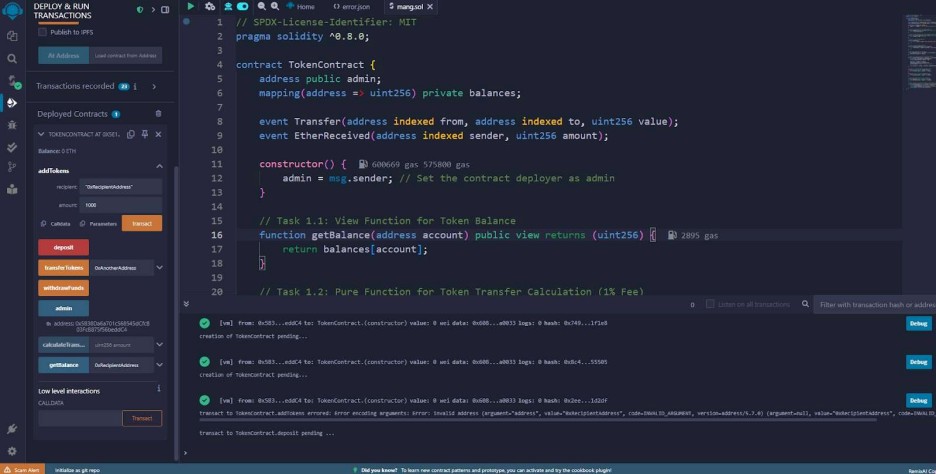


##### Task 5: Fallback and Payable Functions

* **Fallback Function:** Transfers received Ether to the admin and logs the transaction using the Transfer event.
* **Deposit Function:** Allows users to deposit Ether into the contract and emits a Deposit event upon deposit.

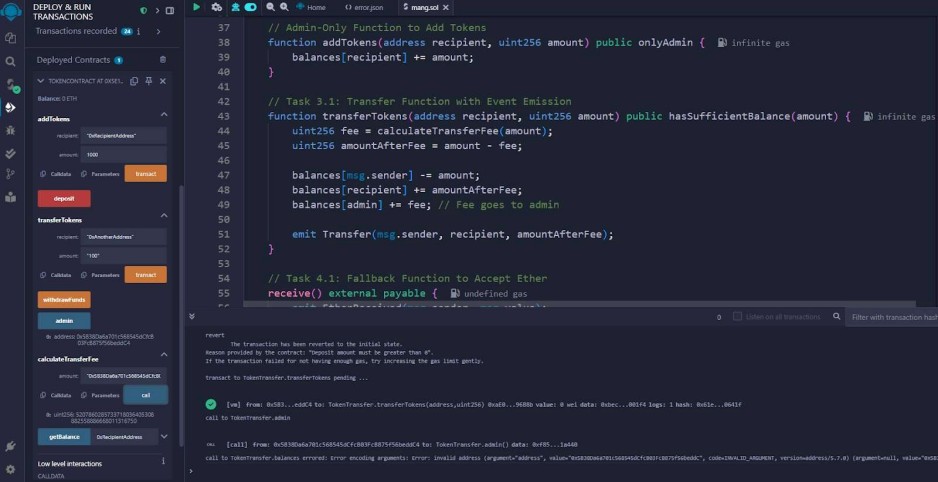
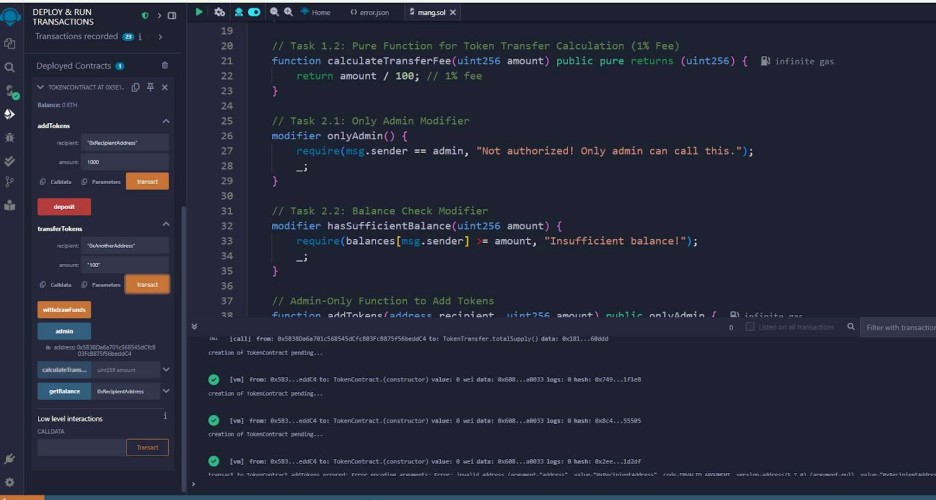


##### Task 6: Testing the Contract

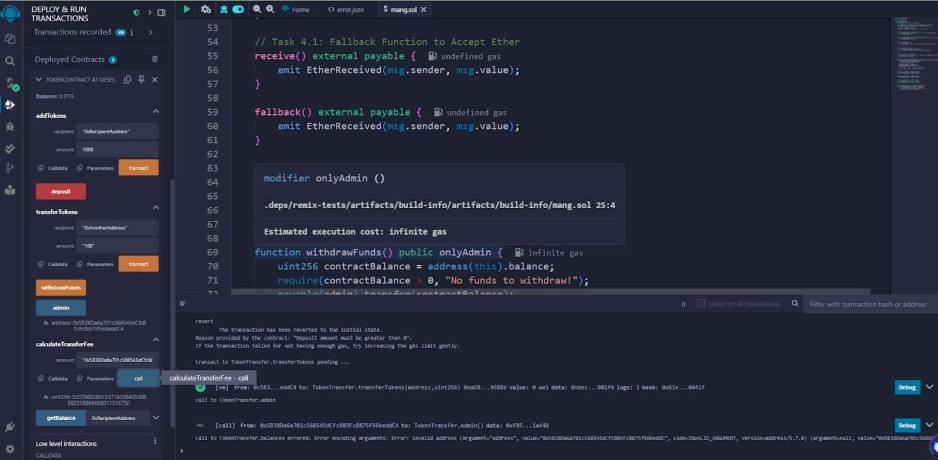


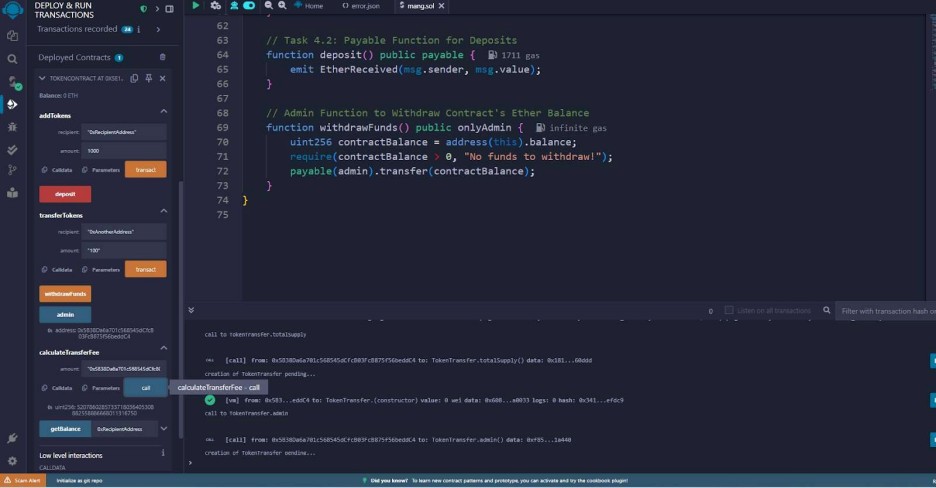
1. **Deploy the Contract:**

● Deploy TokenTransfer using Remix, then verify that the admin is correctly set and totalSupply is initialized to 1,000,000 tokens.



1. **Test the Functions:**





#### Blockchain Lab Report – 06 & 07 (Date: 17-04-2025)

**Managing Complex Data with Structs and Mappings in Solidity and Deploy a smart**

**contract on the Ethereum testnet using MetaMask**

**Introduction:**

In this lab, we implemented a smart contract using Solidity to manage complex data, specifically for managing records like student or employee data. The main goal of this lab was to:

* Utilize **structs** to store and organize multiple attributes of data.
* Implement **mappings** to allow for efficient retrieval of data using keys (e.g., an ID).
* Create functions that allow for adding, updating, and retrieving the data stored in the contract.
* Apply **modifiers** for validation and control of data management.
* Deploy the contract on the **Ethereum testnet** using MetaMask.

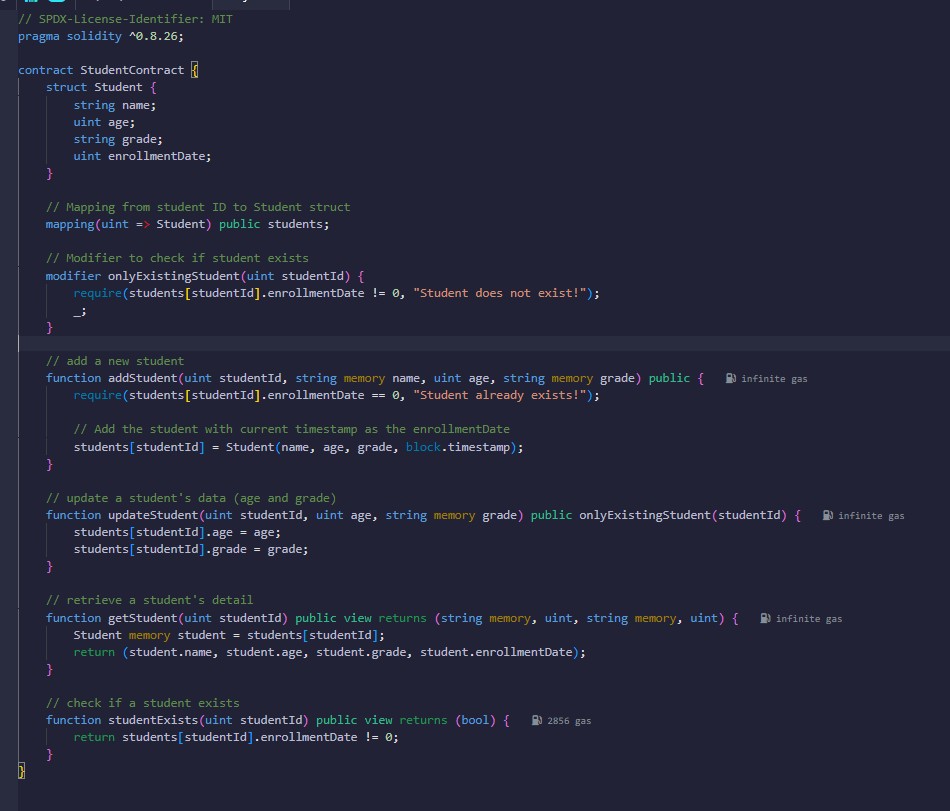
**Problem Statement:**

To manage data such as student records, which typically consist of multiple attributes (e.g., name, age, address, etc.), we needed an efficient way to store and manipulate these records. Directly using arrays for this purpose would be inefficient for large datasets. Hence, using **structs** for data organization and **mappings** for fast lookups is more suitable.

Real-World Scenario given:

Imagine you are building a Student Management System for a school, where each student has a unique ID, name, age, and grades. This contract will store and manage these student records using structs and mappings.

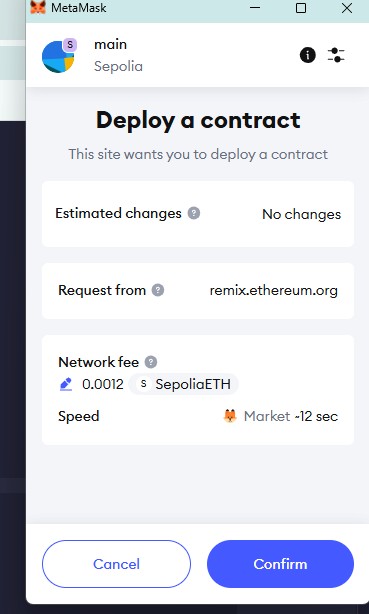
CODE



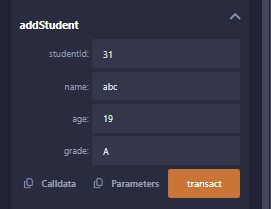
Functions:

* addStudent: Add students with a unique ID, name, age, and grade.
* getStudent: Retrieve student details by their ID.
* updateStudent: Update a student's age and grade. ● studentExists: Check if a student exists.

* While deploying we will get popup of metamask like below

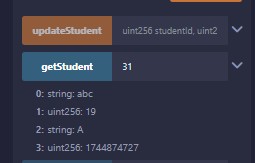


* Click confirm and then
* Add new student in remix IDE



* In metamask we will get this here new user is created

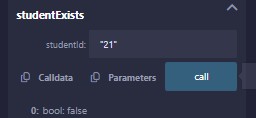
To see that go get student



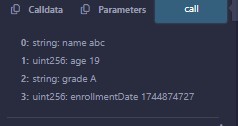
* To check student exist



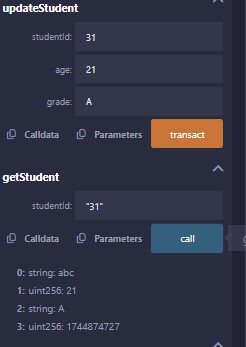
* This student exist



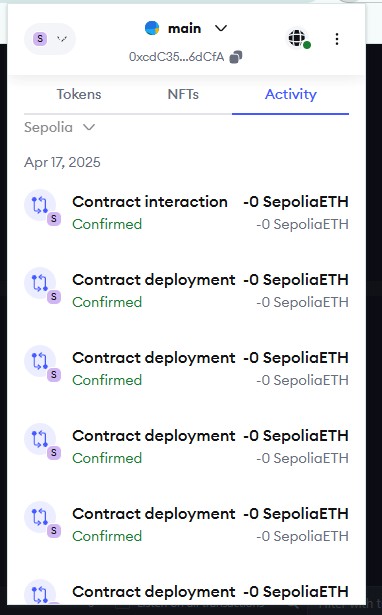
* This don't exist so its giving false output



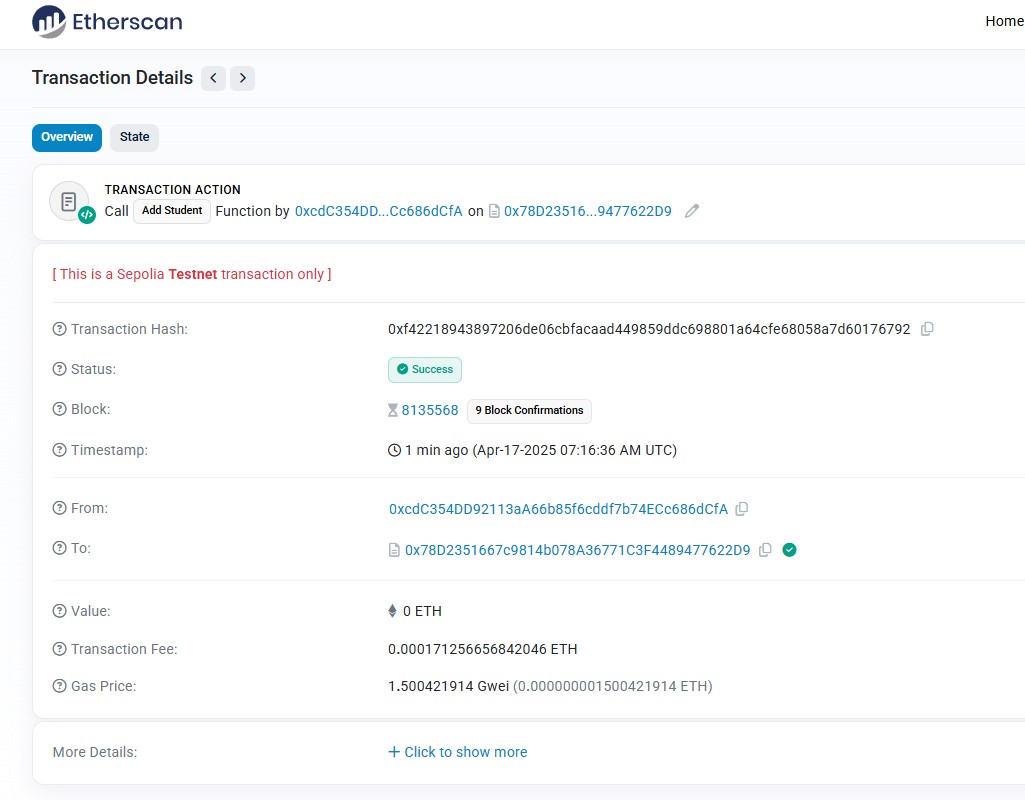
After updating age we get this



* While adding new user we need to confirm in metamask also
* Here we are using SepoliaETH as test token it will charge some amount of gas fee per transaction



In etherscan we can see transaction detail as below



##### Conclusion

In this lab, we successfully implemented a **Student Management Contract** using Solidity and tested it on the **Sepolia testnet** using **SepoliaETH** as the testing token. MetaMask was used to interact with the Ethereum network, providing a secure way to deploy and interact with the contract. The contract efficiently handled the storage of student data using **structs** and **mappings**, with functions to add, update, and retrieve student information. Through proper validation with **modifiers**, the contract ensured the integrity of the data. Testing on Sepolia confirmed that the contract functions as expected, providing hands-on experience with Ethereum development and deploying contracts on a test network. This lab helped reinforce key concepts in blockchain development, including contract deployment, state management, and interaction with test networks.

**LAB 7: Peer-to-Peer Blockchain Network using Sockets**

**BlockChain Lab Report (Date: 15-05-2025)**

##### Objective

* Simulate a decentralized blockchain network using sockets.
* Understand how peers communicate and share data without a central authority.
* Implement block broadcasting and validation between peer nodes.

###### Software/Tools Required

* Python 3.8+
* VS Code / PyCharm / Jupyter Notebook
* hashlib module (standard in Python)

###### Tools & Requirements

* Python 3.8+
* Terminal or Command Prompt
* Text editor (VS Code / PyCharm)

###### Tasks

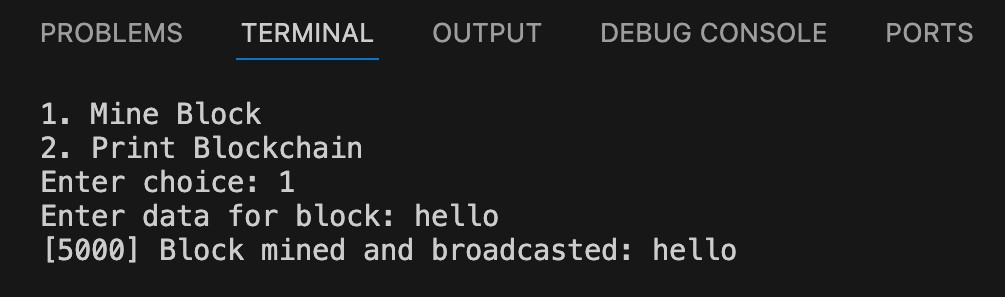
|  |
| --- |
| **import socket** |
| **import threading** |
| **import json** |
| **import time** |
| **import hashlib** |
| **import sys** |
|  |
| **# -------------------- Block and Blockchain --------------------** |
|  |
| **class Block:** |
| **def \_\_init\_\_(self, index, timestamp, data, previous\_hash):** |
| **self.index = index** |
| **self.timestamp = timestamp** |
| **self.data = data** |
| **self.previous\_hash = previous\_hash** |
| **self.hash = self.calculate\_hash()** |
|  |
| **def calculate\_hash(self):** |
| **block\_string = f"{self.index}{self.timestamp}{self.data}{self.previous\_hash}"** |
| **return hashlib.sha256(block\_string.encode()).hexdigest()** |
|  |
| **def to\_dict(self):** |

|  |
| --- |
| **return {** |
| **'index': self.index,** |
| **'timestamp': self.timestamp,** |
| **'data': self.data,** |
| **'previous\_hash': self.previous\_hash,** |
| **'hash': self.hash** |
| **}** |
|  |
| **@staticmethod** |
| **def from\_dict(block\_dict):** |
| **block = Block(** |
| **block\_dict['index'],** |
| **block\_dict['timestamp'],** |
| **block\_dict['data'],** |
| **block\_dict['previous\_hash']** |
| **)** |
| **block.hash = block\_dict['hash']** |
| **return block** |
|  |
| **class Blockchain:** |
| **def \_\_init\_\_(self):** |
| **self.chain = [self.create\_genesis\_block()]** |
|  |
| **def create\_genesis\_block(self):** |
| **fixed\_timestamp = 1650000000.0** |
| **genesis\_block = Block(0, fixed\_timestamp, "Genesis Block", "0")** |
| **genesis\_block.hash = genesis\_block.calculate\_hash()** |
| **return genesis\_block** |
|  |
| **def get\_last\_block(self):** |
| **return self.chain[-1]** |
|  |
| **def add\_block(self, block):** |
| **if self.is\_valid\_new\_block(block, self.get\_last\_block()):** |
| **self.chain.append(block)** |
| **return True** |
| **return False** |
|  |
| **def is\_valid\_new\_block(self, new\_block, previous\_block):** |
| **if previous\_block.index + 1 != new\_block.index:** |
| **return False** |
| **if previous\_block.hash != new\_block.previous\_hash:** |

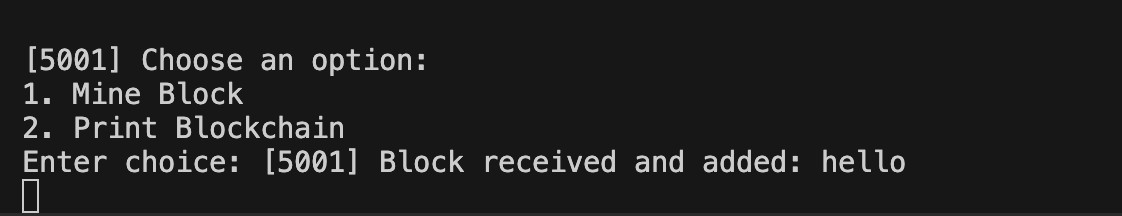
|  |
| --- |
| **return False** |
| **if new\_block.hash != new\_block.calculate\_hash():** |
| **return False** |
| **return True** |
|  |
| **def to\_list(self):** |
| **return [block.to\_dict() for block in self.chain]** |
|  |
| **# -------------------- Peer Node --------------------** |
|  |
| **class PeerNode:** |
| **def \_\_init\_\_(self, port, peer\_ports):** |
| **self.port = port** |
| **self.peers = peer\_ports** |
| **self.connections = []** |
| **self.blockchain = Blockchain()** |
| **self.lock = threading.Lock()** |
|  |
| **self.start\_server()** |
| **self.connect\_to\_peers()** |
|  |
| **threading.Thread(target=self.command\_line\_interface, daemon=True).start()** |
|  |
| **def start\_server(self):** |
| **server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)** |
| **server.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)** |
|  |
| **try:** |
| **server.bind(('localhost', self.port))** |
| **except OSError as e:** |
| **print(f"[{self.port}] ERROR: {e}")** |
| **sys.exit(1)** |
|  |
| **server.listen()** |
| **print(f"[{self.port}] Listening for peers...")** |
| **threading.Thread(target=self.accept\_connections, args=(server,), daemon=True).start()** |
|  |
| **def accept\_connections(self, server):** |
| **while True:** |
| **conn, addr = server.accept()** |
| **self.connections.append(conn)** |
| **threading.Thread(target=self.handle\_connection, args=(conn,), daemon=True).start()** |

|  |
| --- |
|  |
| **def connect\_to\_peers(self):** |
| **for peer\_port in self.peers:** |
| **try:** |
| **sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)** |
| **sock.connect(('localhost', peer\_port))** |
| **self.connections.append(sock)** |
| **threading.Thread(target=self.handle\_connection, args=(sock,), daemon=True).start()** |
| **except Exception as e:** |
| **print(f"[{self.port}] Could not connect to peer {peer\_port}: {e}")** |
|  |
| **def handle\_connection(self, conn):** |
| **while True:** |
| **try:** |
| **data = conn.recv(4096)** |
| **if not data:** |
| **break** |
| **block\_dict = json.loads(data.decode())** |
| **block = Block.from\_dict(block\_dict)** |
|  |
| **with self.lock:** |
| **if self.blockchain.add\_block(block):** |
| **print(f"[{self.port}] Block received and added: {block.data}")** |
| **else:** |
| **print(f"[{self.port}] Received invalid or stale block.")** |
| **except:** |
| **break** |
| **conn.close()** |
|  |
| **def broadcast\_block(self, block):** |
| **block\_json = json.dumps(block.to\_dict()).encode()** |
| **for conn in self.connections:** |
| **try:** |
| **conn.sendall(block\_json)** |
| **except:** |
| **continue** |
|  |
| **def command\_line\_interface(self):** |
| **while True:** |
| **print(f"\n[{self.port}] Choose an option:")** |
| **print("1. Mine Block")** |
| **print("2. Print Blockchain")** |
| **choice = input("Enter choice: ")** |
|  |
| **if choice == '1':** |
| **data = input("Enter data for block: ")** |
| **with self.lock:** |
| **last\_block = self.blockchain.get\_last\_block()** |
| **new\_block = Block(** |
| **index=last\_block.index + 1,** |
| **timestamp=time.time(),** |
| **data=data,** |
| **previous\_hash=last\_block.hash** |
| **)** |
| **self.blockchain.add\_block(new\_block)** |
| **self.broadcast\_block(new\_block)** |
| **print(f"[{self.port}] Block mined and broadcasted: {data}")** |
| **elif choice == '2':** |
| **print(f"\n[{self.port}] Blockchain:")** |
| **for block in self.blockchain.chain:** |
| **print(f"Index: {block.index}, Data: {block.data}, Hash: {block.hash[:10]}...")** |
|  |
| **# -------------------- Main --------------------** |
|  |
| **if \_\_name\_\_ == "\_\_main\_\_":** |
| **if len(sys.argv) < 2:** |
| **print("Usage: python p2p\_blockchain.py <your\_port> <peer\_port\_1> <peer\_port\_2> ...")** |
| **sys.exit(1)** |
|  |
| **port = int(sys.argv[1])** |
| **peers = [int(p) for p in sys.argv[2:]]** |
|  |
| **node = PeerNode(port, peers)** |
|  |
| **# Keep the main thread alive** |
| **while True:** |
| **time.sleep(1)** |

###### Output Terminal 1



###### Terminal 2



###### Terminal 3



###### Observation Report

1. **Describe how blocks are transmitted over the network.**

When a node mines a block, it serializes the block into JSON format and sends it via socket to all connected peers. Each peer receives the block, deserializes it, validates it, and adds it to their local blockchain if valid.

1. **Explain what happens when a node receives an invalid or stale block.**

The node checks if the block is the correct next index and if the previous hash matches its last block. If it fails these checks, it prints "Received invalid or stale block" and discards it.

1. **Limitations and Improvements:**

* **No Proof-of-Work** – No mining difficulty, making it insecure against spam or tampering.
* **No Dynamic Peer Discovery** – Peers must be manually listed at startup.
* **No Chain Sync on Startup** – New peers don’t sync the full blockchain from others.

* **Improvements**:

○ Add proof-of-work for better security.

○ Implement peer discovery using gossip protocols or DHT.

○ Sync chain history when a new peer joins.

**Lab 7 : Implementing Merkle Tree and Blockchain with State Alteration**

###### Objective

* To implement a Merkle Tree for transaction integrity verification.
* To develop a basic Blockchain that supports:

○ Genesis block creation

○ New block mining with transactions

○ Tamper detection by altering state (transactions)

○ Merkle Root recalculation on state changes

###### Software/Tools Required

* Python 3.8+
* VS Code / PyCharm / Jupyter Notebook
* hashlib module (standard in Python)

###### Assignment Description

You are required to implement a blockchain that uses Merkle Trees to verify the integrity of transactions. The blockchain should support block creation (mining), transaction addition, and tampering simulation by altering existing transactions.

###### Tasks

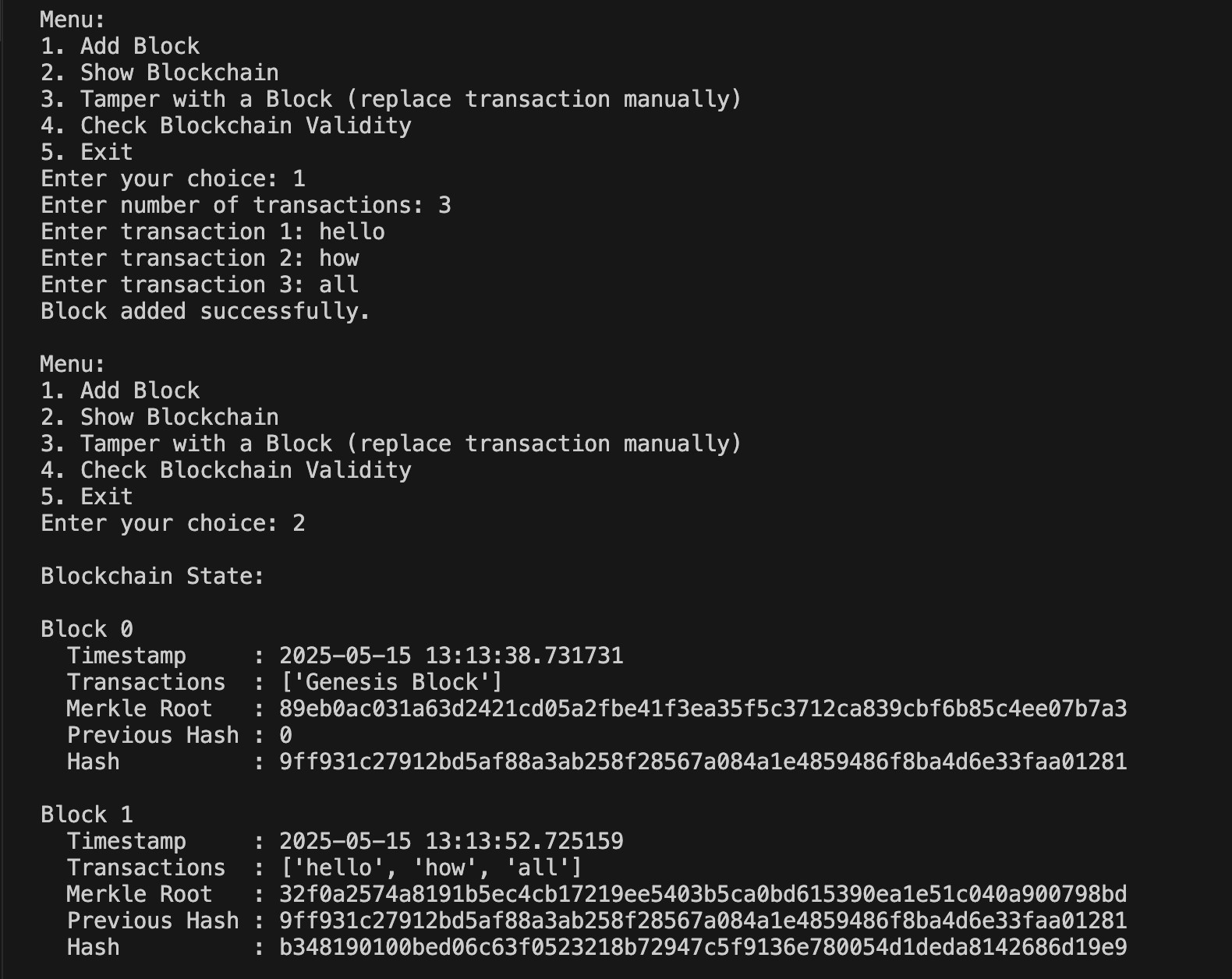
|  |
| --- |
| **import hashlib** |
| **import datetime** |
|  |
| **# Merkle Tree Implementation** |
| **class MerkleTree:** |
| **def \_\_init\_\_(self, transactions):** |
| **self.transactions = transactions** |
| **self.root = self.build\_merkle\_root(transactions)** |
|  |

|  |
| --- |
| **def hash\_pair(self, a, b):** |
| **return hashlib.sha256((a + b).encode()).hexdigest()** |
|  |
| **def build\_merkle\_root(self, transactions):** |
| **if not transactions:** |
| **return ''** |
| **temp = [hashlib.sha256(tx.encode()).hexdigest() for tx in transactions]** |
|  |
| **while len(temp) > 1:** |
| **if len(temp) % 2 != 0:** |
| **temp.append(temp[-1]) # Duplicate last element if odd** |
| **temp = [self.hash\_pair(temp[i], temp[i+1]) for i in range(0, len(temp), 2)]** |
|  |
| **return temp[0]** |
|  |
| **# Block structure** |
| **class Block:** |
| **def \_\_init\_\_(self, index, transactions, previous\_hash):** |
| **self.index = index** |
| **self.timestamp = str(datetime.datetime.now())** |
| **self.transactions = transactions** |
| **self.merkle\_root = MerkleTree(transactions).root** |
| **self.previous\_hash = previous\_hash** |
| **self.hash = self.compute\_hash()** |
|  |
| **def compute\_hash(self):** |
| **block\_string = str(self.index) + self.timestamp + str(self.transactions) + self.merkle\_root + self.previous\_hash** |
| **return hashlib.sha256(block\_string.encode()).hexdigest()** |
|  |
| **# Blockchain structure** |
| **class Blockchain:** |
| **def \_\_init\_\_(self):** |
| **self.chain = []** |
| **self.create\_genesis\_block()** |
|  |
| **def create\_genesis\_block(self):** |
| **genesis\_block = Block(0, ["Genesis Block"], "0")** |
| **self.chain.append(genesis\_block)** |
|  |
| **def add\_block(self, transactions):** |
| **last\_block = self.chain[-1]** |
| **new\_block = Block(len(self.chain), transactions, last\_block.hash)** |

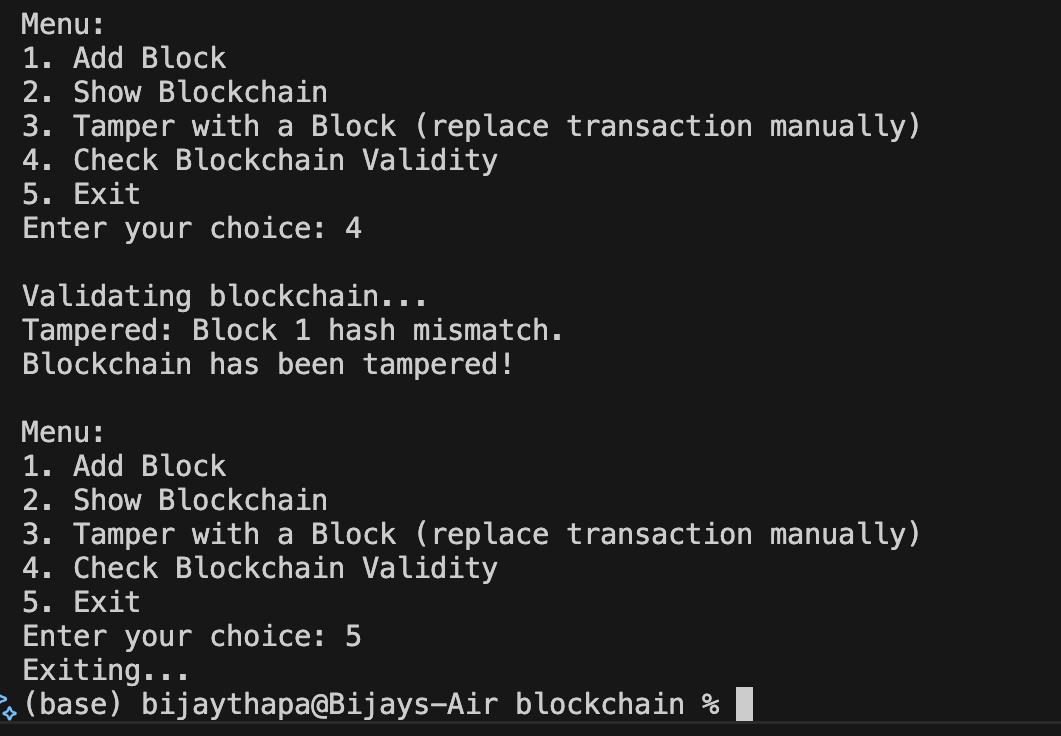
|  |
| --- |
| **self.chain.append(new\_block)** |
|  |
| **def is\_valid(self):** |
| **for i in range(1, len(self.chain)):** |
| **current = self.chain[i]** |
| **previous = self.chain[i - 1]** |
|  |
| **recalculated\_merkle\_root = MerkleTree(current.transactions).root** |
| **recalculated\_hash = hashlib.sha256(** |
| **(str(current.index) + current.timestamp + str(current.transactions) +** |
| **recalculated\_merkle\_root + current.previous\_hash).encode()** |
| **).hexdigest()** |
|  |
| **if current.hash != recalculated\_hash:** |
| **print(f"Tampered: Block {current.index} hash mismatch.")** |
| **return False** |
|  |
| **if current.previous\_hash != previous.hash:** |
| **print(f"Tampered: Block {current.index} previous hash mismatch.")** |
| **return False** |
|  |
| **return True** |
|  |
| **def print\_chain(self):** |
| **for block in self.chain:** |
| **print(f"\nBlock {block.index}")** |
| **print(f" Timestamp : {block.timestamp}")** |
| **print(f" Transactions : {block.transactions}")** |
| **print(f" Merkle Root : {block.merkle\_root}")** |
| **print(f" Previous Hash : {block.previous\_hash}")** |
| **print(f" Hash : {block.hash}")** |
|  |
| **# Demo with user input** |
| **if \_\_name\_\_ == "\_\_main\_\_":** |
| **bc = Blockchain()** |
|  |
| **while True:** |
| **print("\nMenu:")** |
| **print("1. Add Block")** |
| **print("2. Show Blockchain")** |
| **print("3. Tamper with a Block (replace transaction manually)")** |
| **print("4. Check Blockchain Validity")** |

|  |
| --- |
| **print("5. Exit")** |
|  |
| **choice = input("Enter your choice: ")** |
|  |
| **if choice == '1':** |
| **num = int(input("Enter number of transactions: "))** |
| **transactions = []** |
| **for i in range(num):** |
| **tx = input(f"Enter transaction {i+1}: ")** |
| **transactions.append(tx)** |
| **bc.add\_block(transactions)** |
| **print("Block added successfully.")** |
|  |
| **elif choice == '2':** |
| **print("\nBlockchain State:")** |
| **bc.print\_chain()** |
|  |
| **elif choice == '3':** |
| **idx = int(input("Enter block index to tamper (must be >= 1): "))** |
| **if 1 <= idx < len(bc.chain):** |
| **print(f"Current transactions: {bc.chain[idx].transactions}")** |
| **tamper\_index = int(input(f"Enter transaction index to tamper (0 to {len(bc.chain[idx].transactions) - 1}): "))** |
| **if 0 <= tamper\_index < len(bc.chain[idx].transactions):** |
| **new\_tx = input("Enter new tampered transaction: ")** |
| **bc.chain[idx].transactions[tamper\_index] = new\_tx** |
| **# Do NOT recompute hash or merkle root (simulate tampering)** |
| **print("Block tampered (transaction changed without updating hash and Merkle root).")** |
| **else:** |
| **print("Invalid transaction index.")** |
| **else:** |
| **print("Invalid block index.")** |
|  |
| **elif choice == '4':** |
| **print("\nValidating blockchain...")** |
| **if bc.is\_valid():** |
| **print("Blockchain is valid.")** |
| **else:** |
| **print("Blockchain has been tampered!")** |
|  |
| **elif choice == '5':** |
| **print("Exiting...")** |
| **break** |
|  |
| **else:** |
| **print("Invalid choice. Try again.")** |
|  |

**Output**







**Observation Report**

**What is Merkle Root?**

The **Merkle root** is a single cryptographic hash that summarizes all the transactions in a block. It is computed using a binary tree of hashes (Merkle Tree), where each leaf node represents the hash of a transaction, and each non-leaf node represents the hash of the concatenation of its two child nodes. This allows for efficient and secure verification of data integrity in a block.

**What Happens When You Tamper with a Transaction?**

When a transaction in a block is altered, the hash of that transaction changes. This affects the corresponding leaf node in the Merkle Tree, leading to a change in the Merkle root. Consequently, the block's hash also changes. Since each block references the hash of the previous block, any tampering breaks this link, making the blockchain invalid. Thus, even a small modification can be easily detected.

**How Blockchain Ensures Data Integrity?**

Blockchain ensures data integrity using the following mechanisms:

* **Cryptographic Hashing**: Each block’s content is hashed. Even a minor change in data causes a completely different hash.

* **Merkle Trees**: Allows quick verification of the integrity of transactions.

* **Chained Blocks**: Each block contains the hash of the previous block, forming a tamper-evident chain.

* **Consensus Mechanism (in full systems)**: Ensures that all nodes agree on a single valid version of the blockchain.

LAB 10: Implement a blockchain with a mining process in Python, simulating the Proof-ofWork consensus algorithm

Features

* **Block Structure:** Each block contains the following attributes:
  1. index: Position of the block in the chain

○ timestamp: Time of block creation

○ data: Payload/data stored in the block

○ previous\_hash: Hash of the previous block

○ nonce: Counter used for mining

○ hash: SHA-256 hash of the block’s contents

* **SHA-256 Hashing:** Ensures secure and immutable block contents.
* **Mining with Proof-of-Work:** 
  1. Involves finding a nonce such that the block’s hash has a required number of leading zeros.

○ Difficulty is **adjustable**, allowing simulation of how mining becomes harder over time.

* **Genesis Block Creation:** The initial block of the blockchain is created manually, serving as the foundation for subsequent blocks.
* **Block Addition:** New blocks are mined and added to the chain after successfully solving the PoW challenge.

**Code Structure and Explanation**

1. **Block Class:** 
   1. Defines the structure of a block.

○ Includes a method to calculate the hash using SHA-256.

○ Implements the mining process that finds a valid hash according to the difficulty level.

1. **Blockchain Class:** 
   1. Maintains the chain of blocks.

○ Initializes the genesis block.

○ Provides a method to add new blocks by invoking the PoW mechanism.

1. **Adjustable Difficulty:** 
   1. Allows control over mining complexity.

○ Demonstrates the impact of increased difficulty on mining time and computational effort.

CODE

|  |
| --- |
| import hashlib |
| import time |
| class Block: |
| def \_\_init\_\_(self, index, timestamp, data, previous\_hash=''): |
| self.index = index |
| self.timestamp = timestamp |
| self.data = data |
| self.previous\_hash = previous\_hash |
| self.nonce = 0 |
| self.hash = self.calculate\_hash() |
|  |
| def calculate\_hash(self): |
| value = f"{self.index}{self.timestamp}{self.data}{self.previous\_hash}{self.nonce}" |
| return hashlib.sha256(value.encode()).hexdigest() |
|  |
| def mine\_block(self, difficulty): |
| print(f"Mining block {self.index}...") |
| target = '0' \* difficulty |
| while self.hash[:difficulty] != target: |
| self.nonce += 1 |
| self.hash = self.calculate\_hash() |
| print(f"Block {self.index} mined: {self.hash}") |
| class Blockchain: |
| def \_\_init\_\_(self): |
| self.chain = [self.create\_genesis\_block()] |
| self.difficulty = 4 # Adjustable mining difficulty |
|  |

|  |
| --- |
| def create\_genesis\_block(self): |
| return Block(0, time.time(), "Genesis Block", "0") |
|  |
| def get\_latest\_block(self): |
| return self.chain[-1] |
|  |
| def add\_block(self, new\_block): |
| new\_block.previous\_hash = self.get\_latest\_block().hash |
| new\_block.mine\_block(self.difficulty) |
| self.chain.append(new\_block) |
|  |
| def is\_chain\_valid(self): |
| for i in range(1, len(self.chain)): |
| cur = self.chain[i] |
| prev = self.chain[i - 1] |
| if cur.hash != cur.calculate\_hash(): |
| print("Invalid block hash") |
| return False |
| if cur.previous\_hash != prev.hash: |
| print("Invalid chain linkage") |
| return False |
| return True |
|  |
|  |
| if \_\_name\_\_ == "\_\_main\_\_": |
| my\_blockchain = Blockchain() |
| my\_blockchain.add\_block(Block(1, time.time(), "Block 1 Data")) |
| my\_blockchain.add\_block(Block(2, time.time(), "Block 2 Data")) |
| print("\nBlockchain validity:", my\_blockchain.is\_chain\_valid()) |
| for block in my\_blockchain.chain: |
| print(f"\nBlock {block.index}:") |
| print(f"Timestamp : {time.ctime(block.timestamp)}") |
| print(f"Data : {block.data}") |
| print(f"Previous Hash : {block.previous\_hash}") |
| print(f"Hash : {block.hash}") |
| print(f"Nonce : {block.nonce}") |

OUTPUT

