Est. 1841 | YORK ST JOHN UNIVERSITY

York St John University, London

Department of Data Science

Course Name: LDS7006M – Blockchain

Creation and Deployment of Smart Contract for Future Net Ltd.

Student Name: Jamiu Adeyemi Arogundade

Student ID: 240024714

Submission Date: 30 May 2024

MSc Data Science
London Campus

YORK ST JOHN UNIVERSITY
Est. 1841

Module Title	Blockchain
Module code	LDS7006M
Student ID	240024714
Submission Date	30 – 05 - 2024

Student Declaration:

I confirm that I have read and understood the University Policy and Procedures on Academic Misconduct and that the work submitted is my own.	■
I confirm that I have processed and produced this submission in accordance with the University guidelines and the assessment brief regarding the use of generative AI. Where appropriate, I have acknowledged where and how it has been used. OR Where generative AI is not permitted in this assignment. I confirm that it has not been used in any part of the process or production of this submission https://www.yorksj.ac.uk/policies-and-documents/generative-artificial-intelligence/	■ ■

Table of Contents

List of Figures

List of Table

1. Introduction

Blockchain technology has evolved beyond cryptocurrency markets and is now used in various domains such as voting processes, supply chain management, and digital identity management. Firms are conducting trials with Blockchain for decentralised markets, verification of identity, and managing supply chain (Sharabati, 2024.). However, since the various actors that are part of the supply chain transaction don't necessarily have trust among each other, several common issues surface, for example, not all the actors give correct data to the manufacturing company (counterfeit goods case), not everyone gives on-time delivery of the goods when it's crucial for the next supplier in the chain and so on. It becomes hard to get the correct view of the actual goods flow within the supply chain. Hence, the blockchain is introduced in supply chain management such that every time a transaction is undergone between the parties, all the actors of the distributed database get updated based on that particular transaction.

This report details a blockchain design and implementation as a solution for Future Net Ltd., using Solidity smart contracts to automate supply chain transactions and Python to handle blockchain operations.

Objectives

- Design a basic blockchain with key features using Python.
- Develop a simple smart contract named TokenContract using Solidity.
- Deploy the smart contract on a test network and interact with it.
- Provide comprehensive documentation and testing for both the blockchain implementation and the smart contract.

1.1. Overview of Blockchain Technology

Blockchain is a public ledger technology in which all transactions made in digital assets are recorded chronologically and publicly. A blockchain securely records data across a decentralised network, ensures its accuracy and transparency, and prevents tampering with stored data. The network is used as a distributed ledger to secure products against counterfeit, parallel markets, and fraud through effectively tracking their origins. The decentralised and completely open and shared nature of blockchain infrastructure makes it possible for all parties involved in the production and marketing processes to access trustworthy and unalterable information on the product.

As an emerging technology for decentralised and distributed data sharing, Blockchain is also the backbone of most cryptocurrencies, and its decentralised, transparent, and tamper-proof nature has made it a popular choice for an increasing number of applications (Wang, 2021). Furthermore, blockchain can record every unit of data that is interested in a peer-to-peer network. It satisfies the following conditions:

(a) Availability - data can be read or written by any active node and

(b) Authenticity - either a real value or a fork is available.

It was introduced with the implementation of the Bitcoin cryptocurrency project in 2009 and it is based on distributed consensus, implying that parties do not have to deal with a centralised system manager who decided whether a state is legal or not.

1.2. Key Features of Blockchain

Decentralisation:

"Decentralisation involves transferring authority and responsibility for public functions from the central government to subordinate or quasi-independent government organisations and/or the private sector. Blockchain technology is a great example of this, as it uses nodes that interact directly with each other instead of through a central node. This is also known as Peer-to-Peer (P2P) networking. All peers/nodes have equal privileges and are equipotent participants in the network. They share a portion of their resources (such as processing power, network bandwidth, disk storage, etc.) with other network participants. The peers both consume and provide resources" (Mourouzis, 2019).

Immutability:

Immutability is a fundamental concept in blockchain technology, referring to the inability to alter or modify data once it has been recorded on the blockchain (Nakamoto, 2008). This means that once a transaction is confirmed and added to the blockchain, it cannot be deleted, edited, or tampered with in any way (Wood, 2014). The immutability of blockchain data is ensured through the use of cryptographic hashes, which create a permanent and unalterable record of all transactions (Antonopoulos, 2014). This ensures the integrity and transparency of the blockchain, as any attempts to alter the data would be detectable and would invalidate the entire chain (Pilkington, 2016).

Cryptography:

A blockchain uses cryptography for each transaction and consensus models to replace central authorities. Recent transactions are ordered into a block and linked to the chain. The blockchain is open-access and immutable, making it difficult for entities to alter past transactions. This establishes trust independent of a central authority (Sultan, 2018).

Smart Contract:

Several different definitions exist for a smart contract. This lack of a clear definition stems from the rapid evolution of this technology. At its core, a smart contract is a self-executing computer program that triggers predefined actions when certain conditions are met. These programs are used to extend the functionality of public blockchains, allowing these decentralised systems to process more complex logics. Smart contracts are implemented as part of various public blockchains, especially in enterprise or public sectors. These smart contracts have various centralised service alternatives, which operated as platforms, and allow a variety of application fields like healthcare, supply chain, or educational systems. Modern public blockchains are programmable, so smart contracts are designed to automate complex tasks and execute them under specific rules (Mourouzis, 2019).

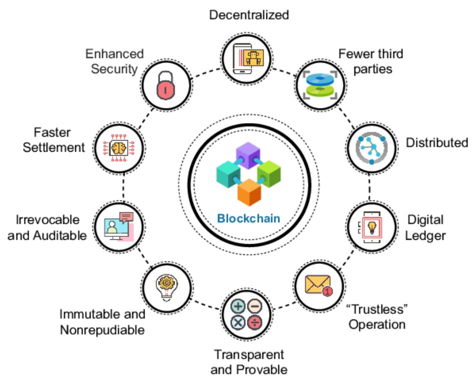


Figure 1: Features of Blockchain (Hazra, 2022)

1.3. Type of Blockchain

Public Blockchain: In a public blockchain, anyone can read, use for transactions, and contribute to consensus. Adhering to the "Code is Law" principle from the open-source movement and cypherpunk philosophy, public operates without a central register or trusted third party for governance. The network's nodes decide whether to integrate the proposed modifications made by developers (Karafiloski E, 2017).

Private Blockchain: A private blockchain is a restricted and permissioned version that operates within a closed network. It is used within organisations, where only specific members have access. This type is well-suited for enterprises and businesses that want to use blockchain technology for internal purposes only. One key difference is their accessibility; public blockchains are highly accessible, while private blockchains are limited to a select group of individuals. Additionally, a private blockchain is more centralised, with a single authority responsible for maintaining the network (Dong, 2023).

Consortium Blockchain: Consortium blockchain, which is also recognised as federated blockchain, is the optimal solution for organisations that necessitate the utilisation of both public and private blockchains (Sheth, 2019).This category involves the participation of various organisations who bear the responsibility of facilitating access to specific nodes for the purposes of reading, writing, and auditing the blockchain. Due to the absence of a centralised governing body, the control is partially decentralised. It lies within the line between fully public (fully decentralised) and private (fully centralised) blockchain networks.

1.4. Advantages and Disadvantages of Blockchain

Advantages	Disadvantages
Enhanced data protection through cryptography and decentralisation.	Potential performance issues due to distribution.
Distributed ledger ensures equal access to information.	Proof-of-work consensus mechanisms can be energy-intensive.
Reduces the risk of a single point of failure.	Regulatory uncertainties and potential misuse.
Tamper-proof data records once on the blockchain.	Difficult to undo incorrect transactions or fraud.
Peer-to-peer transactions without intermediaries.	Technical expertise required for understanding.
Timestamped and transparent transaction history.	Expensive and impractical for large storage.
Automated self-executing contracts with predefined conditions.	Longer transaction processing time in some blockchains.
Pseudonymity for added privacy.	Slow decision-making and consensus challenges.
Internet-based accessibility for anyone with an internet connection.	Potential vulnerabilities in implementation.

Table 1: Advantages and Disadvantages of Blockchain (Dong, 2023)

2. Blockchain (Python)

The blockchain system arranges data into blocks using a specific hashing algorithm and data structure. Each node processes and encodes transaction data, grouping it into blocks, which are then linked to the main blockchain with a timestamp. This process involves technical components such as blocks, chain structures, hashing algorithms, Merkle trees, and timestamps, ensuring secure and organized data arrangement. (Zhu, Guo & Zhang, 2021).

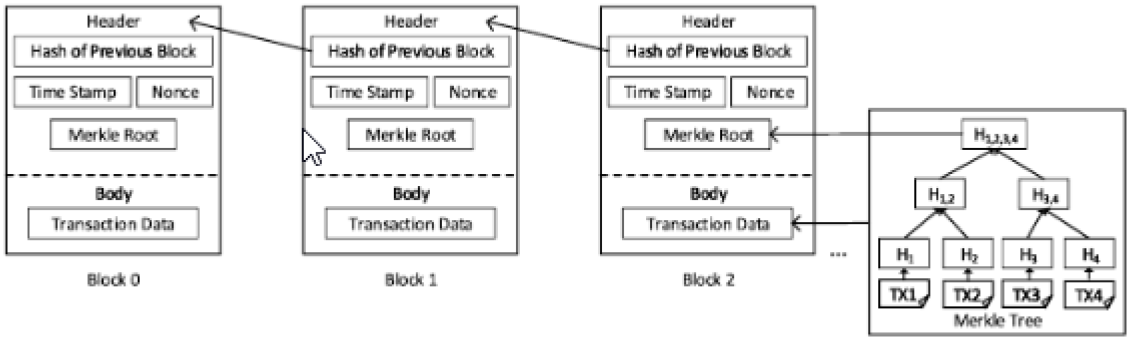


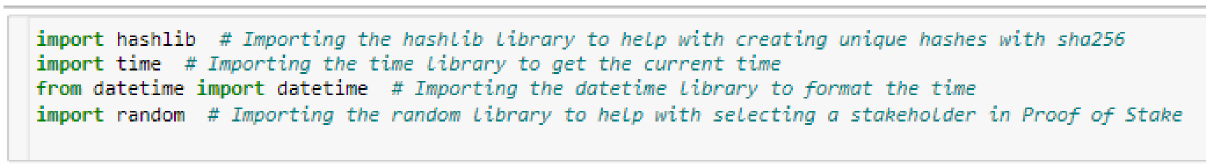
Figure 2: The structure of a Blockchain (Liang, 2020).

2.1. Blockchain Design

This program is designed to give an insight into the fundamental principles of blockchain technology, including the creation, validation, and management of blocks within a chain. This implementation features both Proof of Work and Proof of Stake consensus mechanisms, allowing users to experience different methods of achieving consensus within a distributed network.

The blockchain begins with the creation of a Genesis Block, which serves as the foundational block of the chain. Users are then prompted to decide if they want to mine additional blocks, selecting a consensus mechanism to use for each block. For PoW, miners solve computational puzzles to add blocks, while PoS randomly selects validators based on their stake. The blockchain's integrity is continuously checked, ensuring that each block's hash is valid and resonates with the previous block.

2.1.1. Import Libraries



```python
import hashlib  # Importing the hashlib library to help with creating unique hashes with sha256
import time  # Importing the time library to get the current time
from datetime import datetime  # Importing the datetime library to format the time
import random  # Importing the random library to help with selecting a stakeholder in Proof of Stake
```

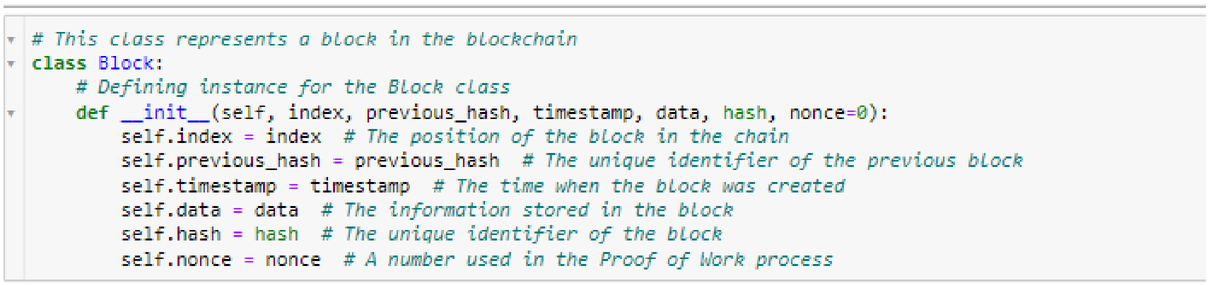
Figure 3: Importing Libraries

We start by importing the following libraries:

- **hashlib:** This library assists in generating unique digital fingerprints, known as hashes, which are crucial in blockchain for maintaining data integrity. The unique digital fingerprints are generated with SHA256 algorithm.
- **time:** This library provides functions to retrieve the current time, which is utilised for timestamping blocks within the blockchain.
- **datetime:** This library formats timestamps into readable date and time strings.
- **random:** This library assists in the random selection of a stakeholder in the Proof of Stake (PoS) consensus mechanism.

2.1.2. Defining the Block class

In a blockchain network, transactions are validated and recorded in a block consisting of a header and a body. The header contains the hash of the previous block, timestamp, Nonce, and Merkle root, while the transaction data is stored in the body. Storing the previous block's hash in the current block ensures the blockchain grows as new blocks are linked to it, detecting any tampering. The timestamp records block creation time, Nonce is used in block creation and verification, and the Merkle tree labels leaf nodes with transaction hashes and non-leaf nodes with concatenated hashes of child nodes (Liang, 2020).



```python
# This class represents a block in the blockchain
class Block:
    # Defining instance for the Block class
    def __init__(self, index, previous_hash, timestamp, data, hash, nonce=0):
        self.index = index  # The position of the block in the chain
        self.previous_hash = previous_hash  # The unique identifier of the previous block
        self.timestamp = timestamp  # The time when the block was created
        self.data = data  # The information stored in the block
        self.hash = hash  # The unique identifier of the block
        self.nonce = nonce  # A number used in the Proof of Work process
```

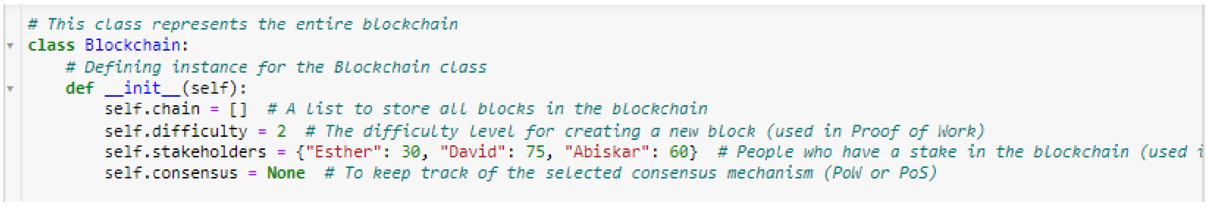
Figure 4: Defining the Block class.

Block is a basic unit of a blockchain containing data, a hash, a reference to the previous block's hash, a timestamp, and a nonce index. These are explained below:

- **hash:** The unique identifier of the block, derived from its contents.

- **previous_hash:** The hash of the previous block.
- **timestamp:** The time at which the block was created.
- **data:** The data stored in the block (e.g., transaction details).
- **nonce:** A number used in the proof-of-work process to generate a valid hash.

2.1.3. Defining Blockchain Class



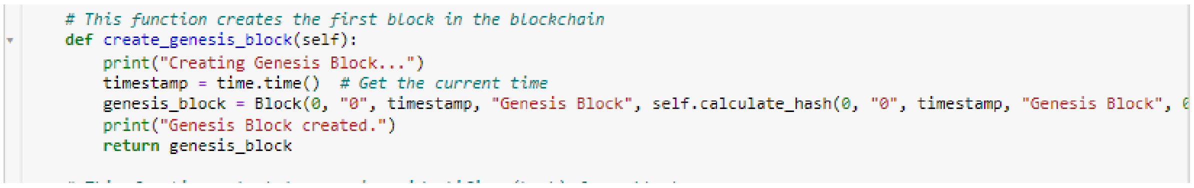
```
# This class represents the entire blockchain
class Blockchain:
    # Defining instance for the Blockchain class
    def __init__(self):
        self.chain = []  # A list to store all blocks in the blockchain
        self.difficulty = 2  # The difficulty level for creating a new block (used in Proof of Work)
        self.stakeholders = {"Esther": 30, "David": 75, "Abiskar": 60}  # People who have a stake in the blockchain (used i
        self.consensus = None  # To keep track of the selected consensus mechanism (PoW or PoS)
```

Figure 5: Defining Blockchain Class

The class blockchain consists of a chain of blocks, with each block containing a record of the transactions. It includes methods for adding blocks, validating the chain, and managing the consensus mechanisms such as PoW and PoS.

- **chain:** A series of connecting nodes or blocks.
- **difficulty:** It determines how hard to create a new block in Proof of Work.
- **stakeholders:** Participants in Proof of Stake that have a stake in the blockchain.
- **consensus:** Either Proof of Work or Proof of Stake mechanism.

2.1.4. Creating Genesis Block

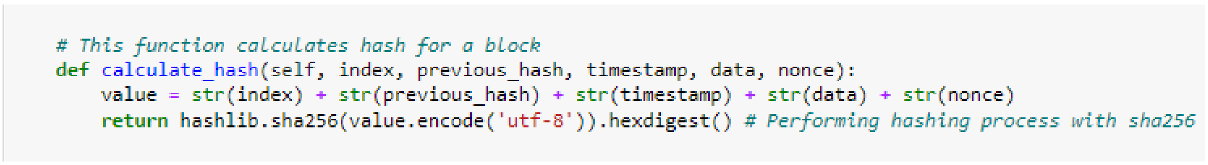


```
    # This function creates the first block in the blockchain
    def create_genesis_block(self):
        print("Creating Genesis Block...")
        timestamp = time.time()  # Get the current time
        genesis_block = Block(0, "0", timestamp, "Genesis Block", self.calculate_hash(0, "0", timestamp, "Genesis Block", 0
        print("Genesis Block created.")
        return genesis_block
```

Figure 6: Creating the Genesis Block

Genesis block is the initial block in the blockchain and acts as the origin for all subsequent blocks. It is created manually and does not reference any previous block. The genesis block's hash is computed using its own data and a nonce value. It has no previous block so it’s ‘previous_block’ is set to 0.

2.1.5. Calculating the Block Hash



```python
# This function calculates hash for a block
def calculate_hash(self, index, previous_hash, timestamp, data, nonce):
    value = str(index) + str(previous_hash) + str(timestamp) + str(data) + str(nonce)
    return hashlib.sha256(value.encode('utf-8')).hexdigest() # Performing hashing process with sha256
```

Figure 7: Calculating the Block Hash

Hash calculation is the generation of a unique digital fingerprint of the block's contents to ensure data integrity. Any alteration to the block will produce a different hash.

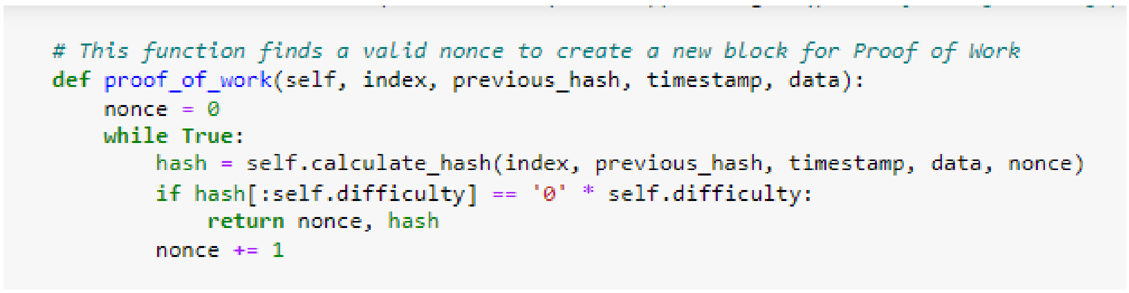
SHA256

SHA256 is a cryptographic function used in providing encryption by obtaining a 256-bit hash from the original transaction record of any length.

The SHA256 hash function has a fixed length, consistent timing, one-way nature, and randomness. Fixed length ensures consistent output, while timing means it takes the same time to compute regardless of input length. It is practically impossible to derive the original input from the hash, and even similar inputs produce very different hashes. Additionally, Bitcoin's proof of work relies on the SHA256 function. (Ye C, 2018, qouted in Dong, 2023).

2.1.6. Proof of Work Mechanism (PoW)

The Proof of Work is a critical and foundational feature of blockchain technology. Its main purpose is to achieve consensus and secure the blockchain network by adding new blocks to the chain. The fundamental steps of the mechanism include nodes monitoring and temporarily storing network data, verifying the records, using their computational power to test different random numbers, generating block information, and broadcasting the new block to the network. Once the other nodes pass the verification process, the block is added to the blockchain, and a node is added to the height of the main chain, increasing its height by one. The PoW method aims to establish a reward mechanism to incentivise other nodes in the network to solve a difficult to solve but easy to verify SHA256 mathematical problem, ultimately adding to the security and integrity of the blockchain network (Gervais, 2016; Gemeliarana, 2018; Shi, 2016; quoted in Dong, 2023).

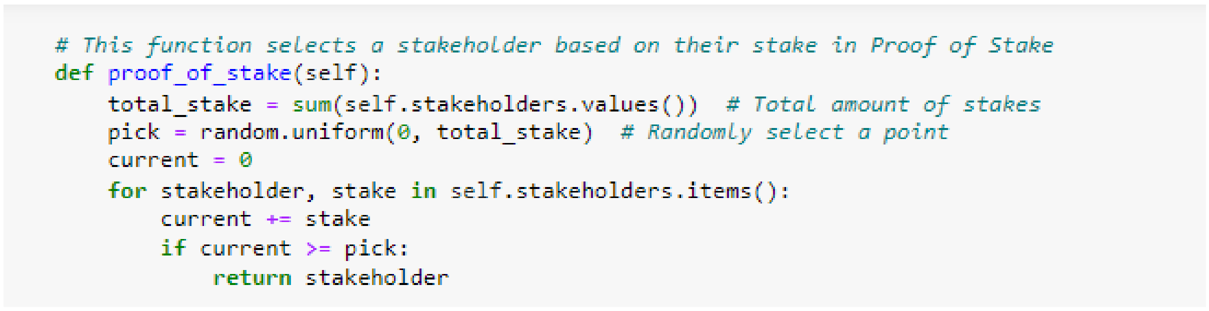


```python
# This function finds a valid nonce to create a new block for Proof of Work
def proof_of_work(self, index, previous_hash, timestamp, data):
    nonce = 0
    while True:
        hash = self.calculate_hash(index, previous_hash, timestamp, data, nonce)
        if hash[:self.difficulty] == '0' * self.difficulty:
            return nonce, hash
        nonce += 1
```

Figure 8: Defining Proof of Work Consensus Mechanism

2.1.7. Proof of Stake Mechanism (PoS)

The Proof-of-Stake (PoS) consensus protocol is made for open blockchains that anyone can join. In this system, a group of validators proposes the next transaction(s) to go in the ledger as blocks. Because there are lots of validators, PoS is a bit tricky to manage. The block generation time depends on what difficulty level is set. If multiple validators solve the problem at once, it causes a split in the block chain. Overall, PoS uses less energy than the PoW consensus. (Li, 2017).

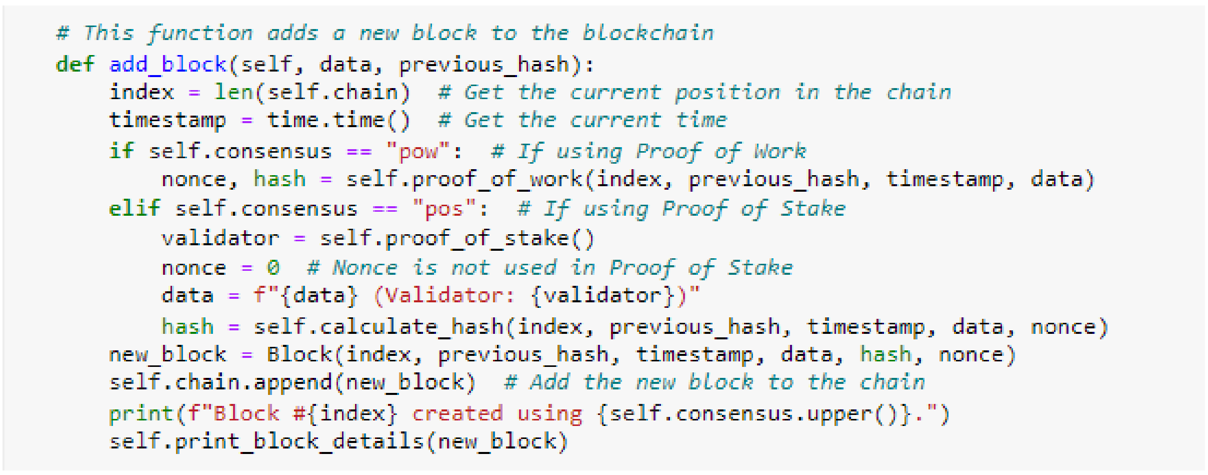


```python
# This function selects a stakeholder based on their stake in Proof of Stake
def proof_of_stake(self):
    total_stake = sum(self.stakeholders.values())  # Total amount of stakes
    pick = random.uniform(0, total_stake)  # Randomly select a point
    current = 0
    for stakeholder, stake in self.stakeholders.items():
        current += stake
        if current >= pick:
            return stakeholder
```

Figure 9: Defining Proof-of-Stake Consensus Mechanism.

2.1.8. Adding a New Block: Consensus Mechanism

Depending on whether the consensus mechanism is PoW or PoS, the process of adding a new block can differ. The new block is added to the blockchain after validation and has similar structures as the genesis block, except that it is linked to a previous block. A consensus algorithm is super important in a blockchain network because it helps all the peers agree on the state of the distributed ledger. It's like a protocol that lets every node in the network figure out the current data state and trust other peers. The network uses a block creation process called "block mining" that gives people an incentive to participate (Wang, 2019). A functional consensus mechanism is necessary for a blockchain network to operate and meet various demands. A secure and universally accepted consensus method is a significant factor that could prompt users of the system. Achieving agreement on blockchain consensus requires high performance in transaction management and low-energy consumption. The blockchain progression also requires support from an uncountable number of participating nodes. An immutable database for transactions and consensus elicits trust from users, as they recognize blockchain's operation as transparent and free from control by any single institution (Naheed Khan, 2021). Proof of Work, introduced by Bitcoin, requires computational power proportional to the sum of block lengths for block addition. Proof of Stake is a trade-off between Proof of Work and Proof of Address, where minters validate transactions, and the next minter is selected based on staked cryptocurrency. However, Proof of Stake has issues with initial stake amounts, which led to the introduction of Proof of Address and Delegated Proof of Stake (DPoS) (Li, 2020).



```python
# This function adds a new block to the blockchain
def add_block(self, data, previous_hash):
    index = len(self.chain)  # Get the current position in the chain
    timestamp = time.time()  # Get the current time
    if self.consensus == "pow":  # If using Proof of Work
        nonce, hash = self.proof_of_work(index, previous_hash, timestamp, data)
    elif self.consensus == "pos":  # If using Proof of Stake
        validator = self.proof_of_stake()
        nonce = 0  # Nonce is not used in Proof of Stake
        data = f"{data} (Validator: {validator})"
        hash = self.calculate_hash(index, previous_hash, timestamp, data, nonce)
    new_block = Block(index, previous_hash, timestamp, data, hash, nonce)
    self.chain.append(new_block)  # Add the new block to the chain
    print(f"Block #{index} created using {self.consensus.upper()}.")
    self.print_block_details(new_block)
```

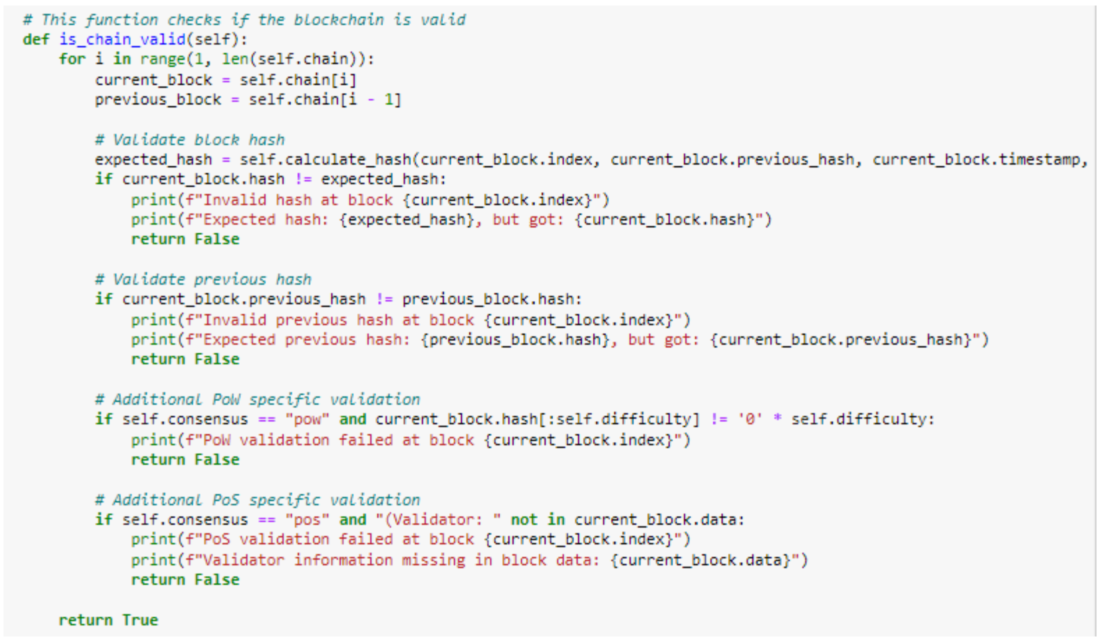
Figure 10: Function to add new blocks.

Note nonce is set to 0 in PoS because the miner is not solving complex mathematical problem. Validators in Proof of Stake get to create new blocks depending on the number of their stakes.

2.1.9. Validating the Blockchain

Validation is done to ensures the integrity and immutability of the blockchain. This is done by checking the hashes of the previous block and the newly mined hashes. The validation is a joint task of the connected nodes who acts as the validators and verifying each new blocks of transactions before it’s appended to the chain. A change in the chain by an actor will result to a new block to be instantly created to prevent attackers from gaining access to the blockchain.

In the code below, a check was done for the validity of the blockchain for both PoW and PoS consensus and additional checks was done based on the consensus to prevent malicious attacks.

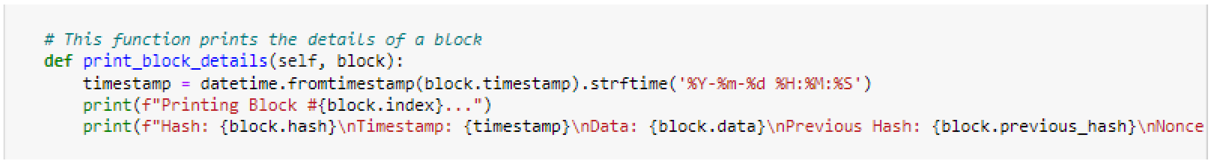


```python
# This function checks if the blockchain is valid
def is_chain_valid(self):
    for i in range(1, len(self.chain)):
        current_block = self.chain[i]
        previous_block = self.chain[i - 1]

        # Validate block hash
        expected_hash = self.calculate_hash(current_block.index, current_block.previous_hash, current_block.timestamp,
        if current_block.hash != expected_hash:
            print(f"Invalid hash at block {current_block.index}")
            print(f"Expected hash: {expected_hash}, but got: {current_block.hash}")
            return False

        # Validate previous hash
        if current_block.previous_hash != previous_block.hash:
            print(f"Invalid previous hash at block {current_block.index}")
            print(f"Expected previous hash: {previous_block.hash}, but got: {current_block.previous_hash}")
            return False

        # Additional PoW specific validation
        if self.consensus == "pow" and current_block.hash[:self.difficulty] != '0' * self.difficulty:
            print(f"PoW validation failed at block {current_block.index}")
            return False

        # Additional PoS specific validation
        if self.consensus == "pos" and "(Validator: " not in current_block.data:
            print(f"PoS validation failed at block {current_block.index}")
            print(f"Validator information missing in block data: {current_block.data}")
            return False

    return True
```

Figure 11: Function for Blockchain Validation.

2.1.10. Printing Block Details

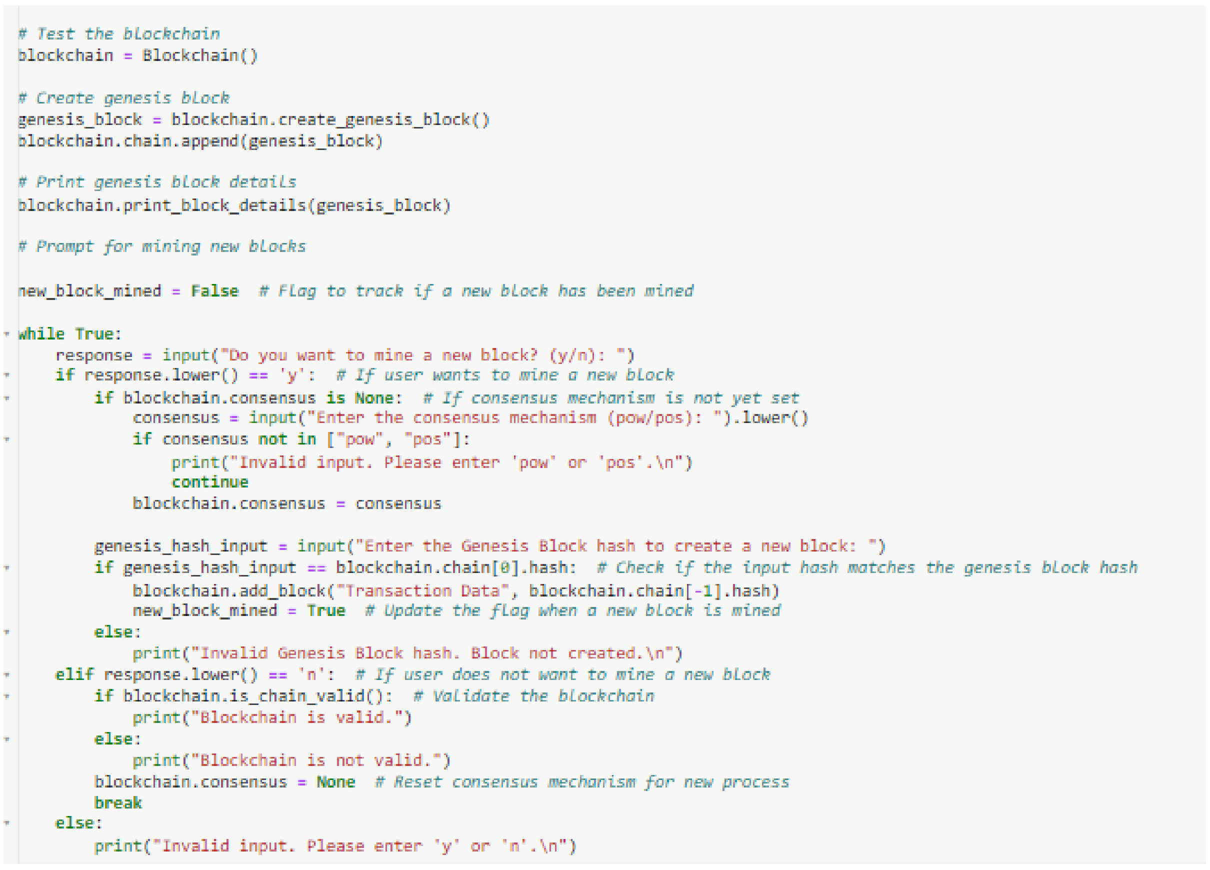
Printing the block details allows to provide readable output format of the block's detail including, its hash, timestamp, data, previous hash, and the nonce. This python provided the genesis block details to printed at first and n input prompt is displayed to enquire if the user would like to mine a new block.



```python
# This function prints the details of a block
def print_block_details(self, block):
    timestamp = datetime.fromtimestamp(block.timestamp).strftime('%Y-%m-%d %H:%M:%S')
    print(f"Printing Block #{block.index}...")
    print(f"Hash: {block.hash}\nTimestamp: {timestamp}\nData: {block.data}\nPrevious Hash: {block.previous_hash}\nNonce
```

Figure 12: Function for Printing the Blockchain.

2.1.11. Running the Blockchain



```python
# Test the blockchain
blockchain = Blockchain()

# Create genesis block
genesis_block = blockchain.create_genesis_block()
blockchain.chain.append(genesis_block)

# Print genesis block details
blockchain.print_block_details(genesis_block)

# Prompt for mining new blocks

new_block_mined = False  # Flag to track if a new block has been mined

while True:
    response = input("Do you want to mine a new block? (y/n): ")
    if response.lower() == 'y':  # If user wants to mine a new block
        if blockchain.consensus is None:  # If consensus mechanism is not yet set
            consensus = input("Enter the consensus mechanism (pow/pos): ").lower()
            if consensus not in ["pow", "pos"]:
                print("Invalid input. Please enter 'pow' or 'pos'.\n")
                continue
            blockchain.consensus = consensus

        genesis_hash_input = input("Enter the Genesis Block hash to create a new block: ")
        if genesis_hash_input == blockchain.chain[0].hash:  # Check if the input hash matches the genesis block hash
            blockchain.add_block("Transaction Data", blockchain.chain[-1].hash)
            new_block_mined = True  # Update the flag when a new block is mined
        else:
            print("Invalid Genesis Block hash. Block not created.\n")
    elif response.lower() == 'n':  # If user does not want to mine a new block
        if blockchain.is_chain_valid():  # Validate the blockchain
            print("Blockchain is valid.")
        else:
            print("Blockchain is not valid.")
        blockchain.consensus = None  # Reset consensus mechanism for new process
        break
    else:
        print("Invalid input. Please enter 'y' or 'n'.\n")
```

Figure 13: Running the Blockchain.

This section initialises and runs the blockchain, allowing user interaction for mining new blocks and validating the blockchain.

First, an instance of the Blockchain class is created, initialising an empty blockchain with predefined stakeholders and a difficulty level. The first block, known as the Genesis Block, is then mined, and added to the blockchain. This block has no previous block, so its previous hash is set to "0". Next, a flag, “new_block_mined”, is set to keep track of whether a new block has been mined.

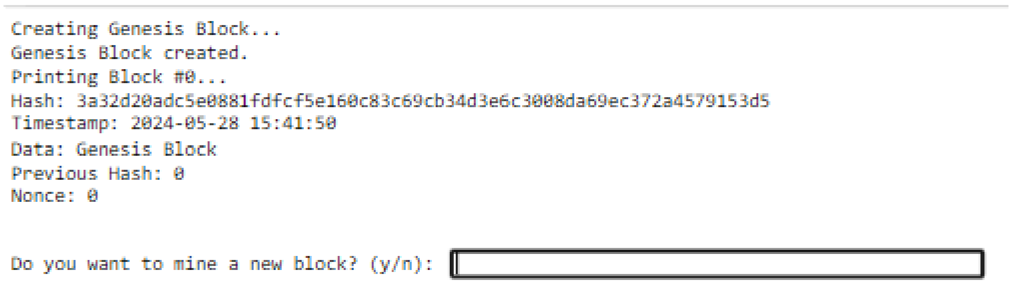
A while loop is created for user interaction. This loop repeatedly prompts the user to decide if they want to mine a new block. If the user chooses not to mine a new block, only the Genesis Block will be present. If the user chooses to mine a new block, the program checks if a consensus mechanism has been set. If not, it prompts the user to enter a preferred consensus mechanism, which can be either Proof of Work or Proof of Stake. This ensures that the blockchain knows which consensus mechanism to use for creating new blocks.

For additional security, the user is prompted to enter the Genesis Block hash to proceed with the new block mining. This ensures that the user is aware of the initial state of the blockchain. If the entered block hash matches the Genesis Block hash, a new block is added to the blockchain using the selected consensus mechanism. The "new_block_mined" flag is set to True.

If the user chooses not to mine any more blocks, the program validates the entire blockchain to ensure its integrity. If the blockchain is valid, it prints "Blockchain is valid." Otherwise, it prints "Blockchain is not valid." The consensus mechanism is then reset to None, and the loop breaks, ending the program.

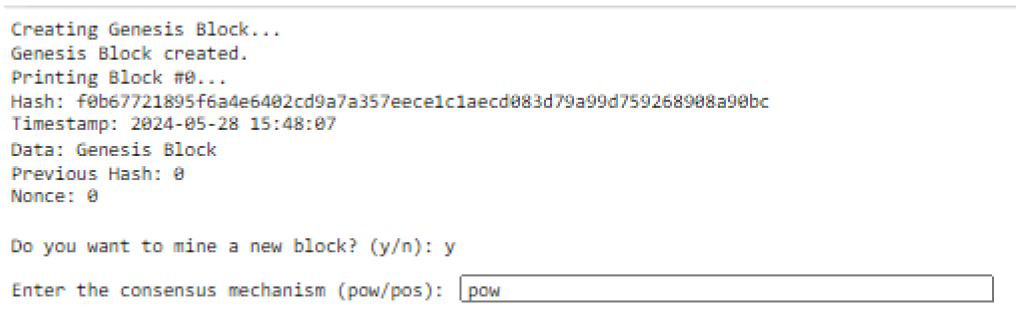
2.2. Results

Figure 14: Genesis block created and a prompt asking if user want to mine a new block.



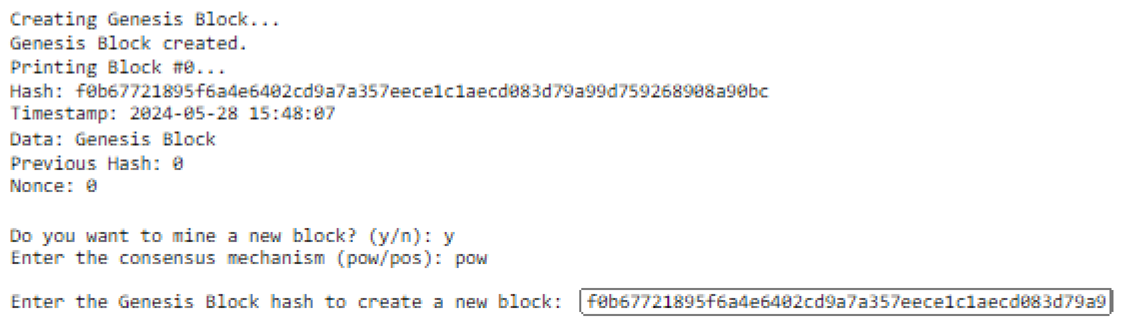
```
Creating Genesis Block...
Genesis Block created.
Printing Block #0...
Hash: 3a32d20adc5e0881fdfcf5e160c83c69cb34d3e6c3008da69ec372a4579153d5
Timestamp: 2024-05-28 15:41:50
Data: Genesis Block
Previous Hash: 0
Nonce: 0

Do you want to mine a new block? (y/n):
```

Figure 15: User enter 'pow' to mine a new block with Proof of Work Mechanism.



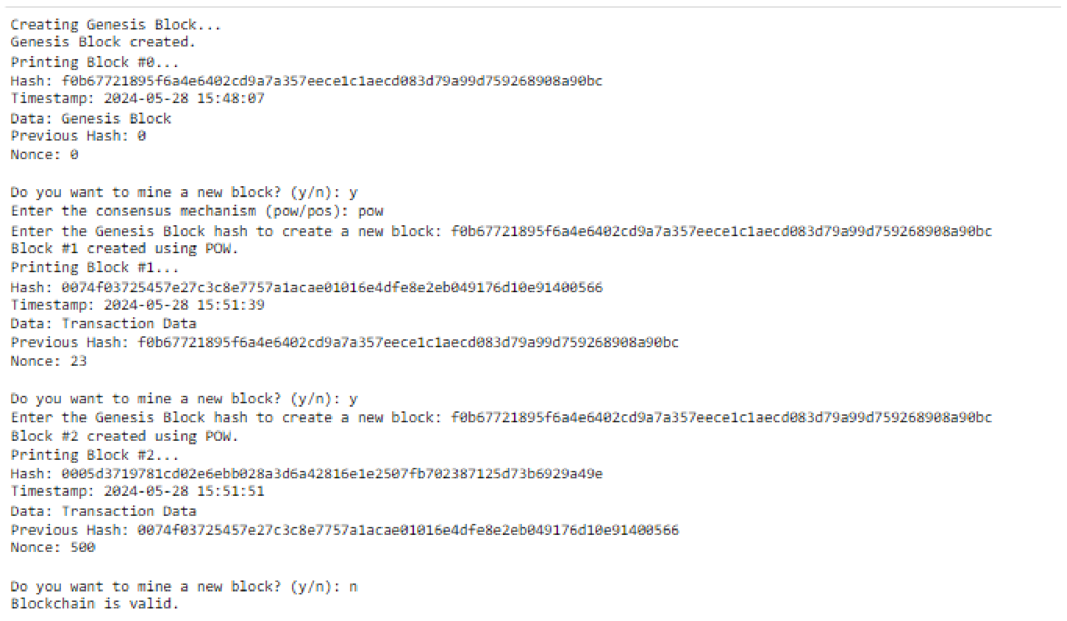
```
Creating Genesis Block...
Genesis Block created.
Printing Block #0...
Hash: f0b67721895f6a4e6402cd9a7a357eece1c1aecd083d79a99d759268908a90bc
Timestamp: 2024-05-28 15:48:07
Data: Genesis Block
Previous Hash: 0
Nonce: 0

Do you want to mine a new block? (y/n): y

Enter the consensus mechanism (pow/pos): pow
```

Figure 16: Program prompting the user to enter the Genesis block hash before mining a new block, hence, enhancing security.



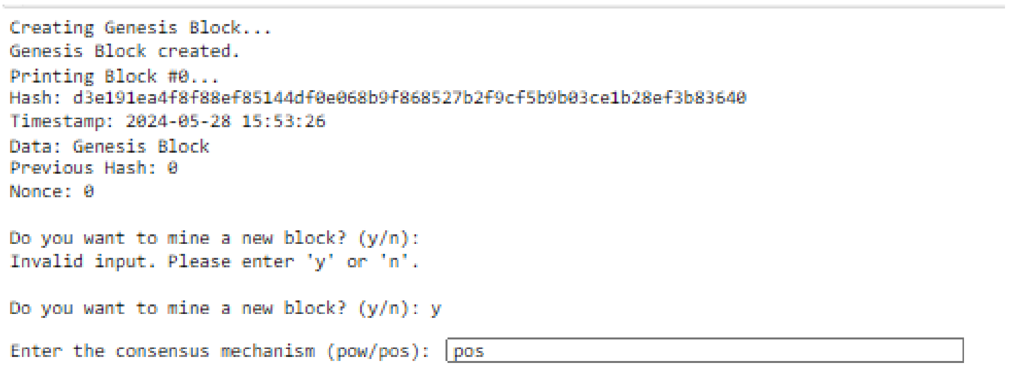
```
Creating Genesis Block...
Genesis Block created.
Printing Block #0...
Hash: f0b67721895f6a4e6402cd9a7a357eece1c1aecd083d79a99d759268908a90bc
Timestamp: 2024-05-28 15:48:07
Data: Genesis Block
Previous Hash: 0
Nonce: 0

Do you want to mine a new block? (y/n): y
Enter the consensus mechanism (pow/pos): pow

Enter the Genesis Block hash to create a new block: f0b67721895f6a4e6402cd9a7a357eece1c1aecd083d79a9
```

Figure 17: New blocks are mined until user enter 'n' to stop mining and the blockchain was Validated.



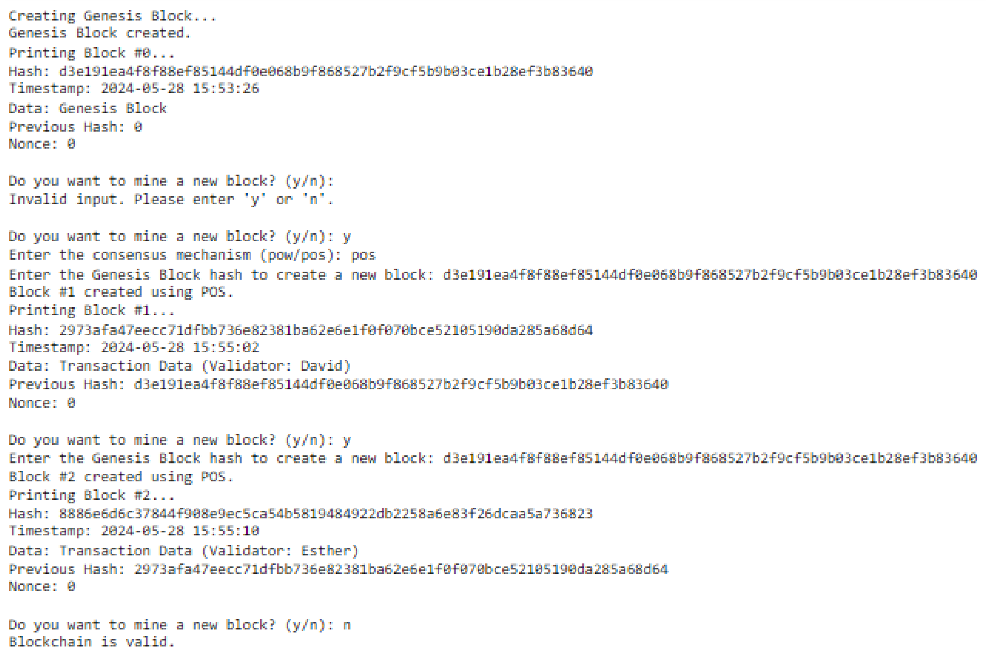
```
Creating Genesis Block...
Genesis Block created.
Printing Block #0...
Hash: f0b67721895f6a4e6402cd9a7a357eece1c1aecd083d79a99d759268908a90bc
Timestamp: 2024-05-28 15:48:07
Data: Genesis Block
Previous Hash: 0
Nonce: 0

Do you want to mine a new block? (y/n): y
Enter the consensus mechanism (pow/pos): pow
Enter the Genesis Block hash to create a new block: f0b67721895f6a4e6402cd9a7a357eece1c1aecd083d79a99d759268908a90bc
Block #1 created using POW.
Printing Block #1...
Hash: 0074f03725457e27c3c8e7757a1acae01016e4dfe8e2eb049176d10e91400566
Timestamp: 2024-05-28 15:51:39
Data: Transaction Data
Previous Hash: f0b67721895f6a4e6402cd9a7a357eece1c1aecd083d79a99d759268908a90bc
Nonce: 23

Do you want to mine a new block? (y/n): y
Enter the Genesis Block hash to create a new block: f0b67721895f6a4e6402cd9a7a357eece1c1aecd083d79a99d759268908a90bc
Block #2 created using POW.
Printing Block #2...
Hash: 0005d3719781cd02e6ebb028a3d6a42816e1e2507fb702387125d73b6929a49e
Timestamp: 2024-05-28 15:51:51
Data: Transaction Data
Previous Hash: 0074f03725457e27c3c8e7757a1acae01016e4dfe8e2eb049176d10e91400566
Nonce: 500

Do you want to mine a new block? (y/n): n
Blockchain is valid.
```

Figure 18: User enter 'pos' here to mine new blocks with the Proof of Stake consensus mechanism.



```
Creating Genesis Block...
Genesis Block created.
Printing Block #0...
Hash: d3e191ea4f8f88ef85144df0e068b9f868527b2f9cf5b9b03ce1b28ef3b83640
Timestamp: 2024-05-28 15:53:26
Data: Genesis Block
Previous Hash: 0
Nonce: 0

Do you want to mine a new block? (y/n):
Invalid input. Please enter 'y' or 'n'.

Do you want to mine a new block? (y/n): y

Enter the consensus mechanism (pow/pos):  pos
```

Figure 19: New blocks were mined and the blockchain was validated after using user entered not to mine new block. The Genesis block was entered at every attempt to mine new block to enhance security.



```
Creating Genesis Block...
Genesis Block created.
Printing Block #0...
Hash: d3e191ea4f8f88ef85144df0e068b9f868527b2f9cf5b9b03ce1b28ef3b83640
Timestamp: 2024-05-28 15:53:26
Data: Genesis Block
Previous Hash: 0
Nonce: 0

Do you want to mine a new block? (y/n):
Invalid input. Please enter 'y' or 'n'.

Do you want to mine a new block? (y/n): y
Enter the consensus mechanism (pow/pos): pos
Enter the Genesis Block hash to create a new block: d3e191ea4f8f88ef85144df0e068b9f868527b2f9cf5b9b03ce1b28ef3b83640
Block #1 created using POS.
Printing Block #1...
Hash: 2973afa47eecc71dfbb736e82381ba62e6e1f0f070bce52105190da285a68d64
Timestamp: 2024-05-28 15:55:02
Data: Transaction Data (Validator: David)
Previous Hash: d3e191ea4f8f88ef85144df0e068b9f868527b2f9cf5b9b03ce1b28ef3b83640
Nonce: 0

Do you want to mine a new block? (y/n): y
Enter the Genesis Block hash to create a new block: d3e191ea4f8f88ef85144df0e068b9f868527b2f9cf5b9b03ce1b28ef3b83640
Block #2 created using POS.
Printing Block #2...
Hash: 8886e6d6c37844f908e9ec5ca54b5819484922db2258a6e83f26dcaa5a736823
Timestamp: 2024-05-28 15:55:10
Data: Transaction Data (Validator: Esther)
Previous Hash: 2973afa47eecc71dfbb736e82381ba62e6e1f0f070bce52105190da285a68d64
Nonce: 0

Do you want to mine a new block? (y/n): n
Blockchain is valid.
```

3. Smart Contract (Solidity)

The concept of smart contracts was first introduced in the mid-1990s, well before the application of blockchain technology (Szabo, 2016). Defined as 'a set of promises in digital format, including protocols for the fulfilment of these promises,' smart contracts essentially capture algorithmic elements of a contract and encode them into computer programs, using cryptographic methods to ensure protection from tampering. This results in contracts that are self-verifiable, self-enforceable, and tamper-proof (Manimuthu, 2021).

Smart contracts serve multiple functions, automating transaction processes, managing user agreements, providing utility to other contracts, ensuring fairness among partners, and facilitating asset transfers upon meeting predetermined agreements (Manupati, 2022). Stored within the blockchain network, each smart contract is marked with a unique address. Smart contracts are formulated by identifying the desired outcome or agreement each party wishes to achieve and setting the necessary validation conditions, followed by implementation across various blockchain nodes. Validation is carried out through a consensus mechanism, which then updates the ledger (Swan, 2015).

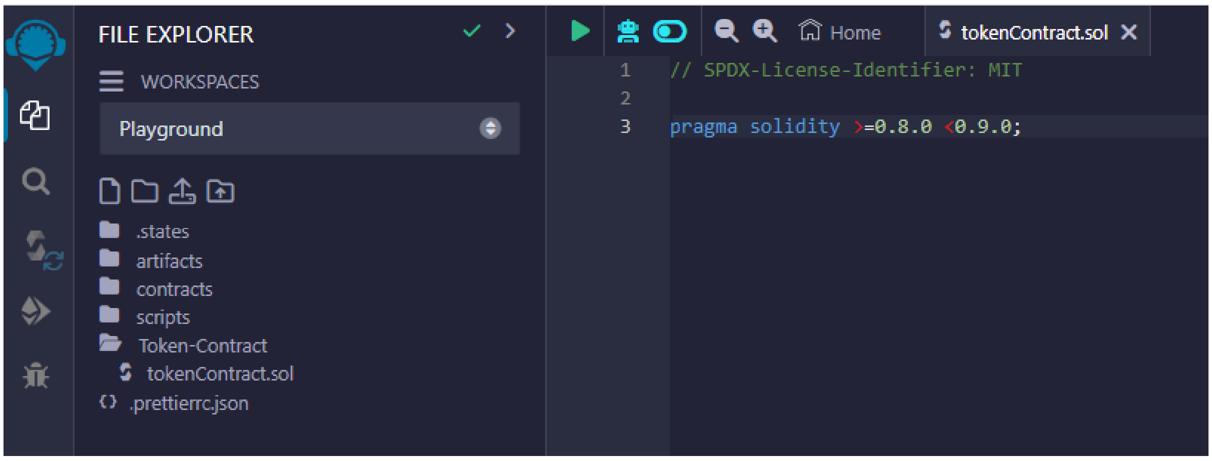
The recent digital transformation experienced by firms and supply chains through Industry 4.0 technologies presents significant strategic considerations and economic implications. While the adoption of Industry 4.0 technologies has gained popularity in recent years due to their potential benefits and role in the digital world (Frank, 2019), further analysis is needed to understand the rapid and increasing trends. Specifically, it is crucial to determine whether Industry 4.0 technologies offer real operational advantages and concrete market opportunities (Hofmann, 2017), or if their adoption is simply a contemporary trend driven by technological progress and government development plans.

3.1. Development of TokenContract

A simple smart contract named TokenContract was developed using Solidity with Remix IDE. This contract implements the basic ERC20 token functionality, including setting a total supply state, minting new tokens, approving other addresses to transfer or spend tokens on the behalf of the owner, and transferring tokens between accounts. The contract also keeps track of the transactions history with a function to get the transaction history.

Solidity is a language designed for creating smart contracts on different blockchain networks, with a primary focus on Ethereum. (https://docs.soliditylang.org/en) and Remix IDE is a code editor for creating a smart contract.

3.1.1. Creating the Solidity (sol) file and setting the license



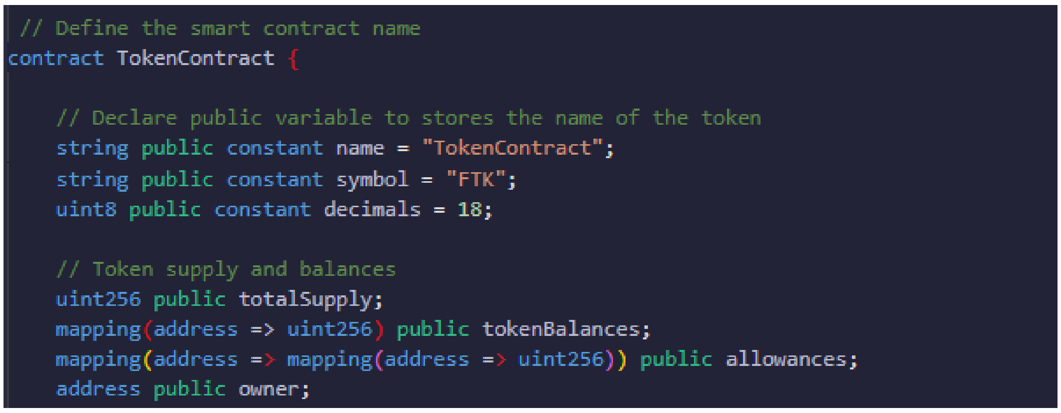
```
// SPDX-License-Identifier: MIT

pragma solidity >=0.8.0 <0.9.0;
```

Figure 20: Created Solidity file and set the license to MIT

Declaring License: The standard license where the code will be released is the MIT license.

Solidity Version: The version is set between the version 0.8.0 and 0.9.0. This is to ensure that the code will be compatible with any version between the two limits.

3.1.2. Declaring the Contract



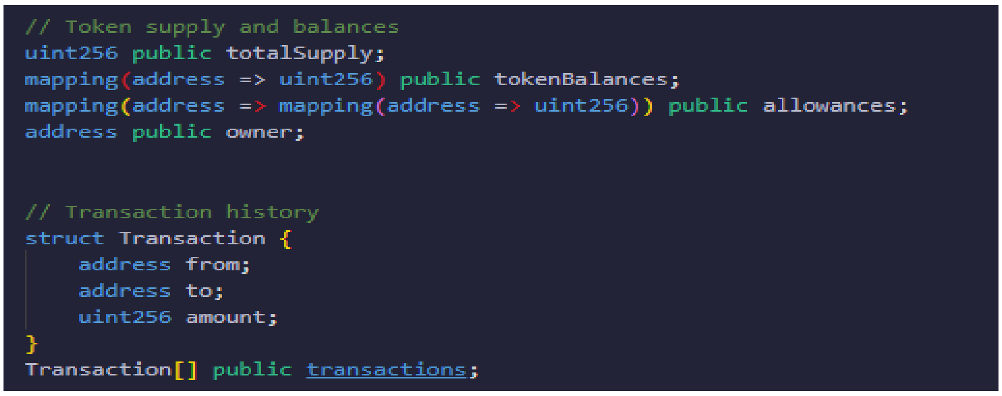
```
// Define the smart contract name
contract TokenContract {

    // Declare public variable to stores the name of the token
    string public constant name = "TokenContract";
    string public constant symbol = "FTK";
    uint8 public constant decimals = 18;

    // Token supply and balances
    uint256 public totalSupply;
    mapping(address => uint256) public tokenBalances;
    mapping(address => mapping(address => uint256)) public allowances;
    address public owner;
```

Figure 21: Declaring the contract state.

The contract TokenContract was declared here. And has the following information:

- Name: 'TokenContract' The name identifier for the contract.
- Symbol: "FTK-Future Token" to represent the name of the client.
- Decimal: Setting the decimal of the contract to 18 is the standard method of creating a smart contract to encourage fractional and micro transactions. Without the decimal the transaction will only be limited to whole unit for example if the balance is $10, making transaction of $9.5 will be impossible. It'll also affected the payment of gas fee.

3.1.3. Stating Variables and Data structures



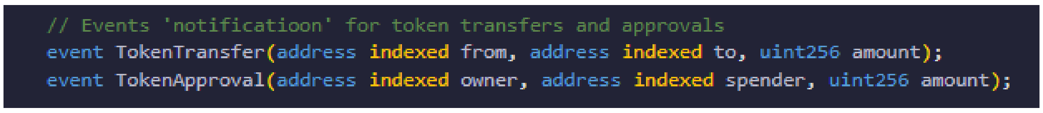
```
// Token supply and balances
uint256 public totalSupply;
mapping(address => uint256) public tokenBalances;
mapping(address => mapping(address => uint256)) public allowances;
address public owner;


// Transaction history
struct Transaction {
    address from;
    address to;
    uint256 amount;
}
Transaction[] public transactions;
```

Figure 22: Stating Variables and Data Structures

- Total Tokens: 'totalSupply' helps keep track of the total token supplied i.e. total token in circulation.
- Token Balance: 'tokenBalances' helps in mapping different addresses to their token balance.
- Allowances: 'allowances' maps owner and spender addresses to the number of tokens allowed to be spent by the spender on behalf of the owner.

- Struct Transaction: ‘Transaction’ struct stores information about transfer of the token. It stores both the sender and the receiver information and the amount of token too.
- Transactions Array: ‘transactions’ keeps the record of all transactions.

3.1.4. Events



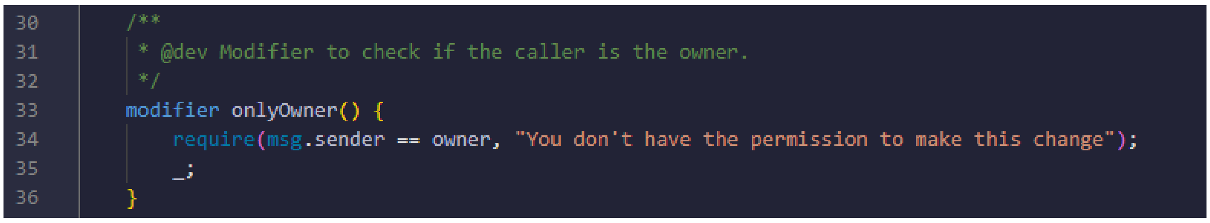
```
// Events 'notificatioon' for token transfers and approvals
event TokenTransfer(address indexed from, address indexed to, uint256 amount);
event TokenApproval(address indexed owner, address indexed spender, uint256 amount);
```

Figure 23: Logging the events.

The ‘TokenTransfer’ and ‘TokenApproval’ events log the transfers and approval of the token. The are both indexed for easy identification.

The ‘TokenTransfer’ log the event of token after from sender address to the receiver address where the ‘TokenApproval’ log the event of approval by owner for spender to transfer token on its behalf.

3.1.5. Modifier

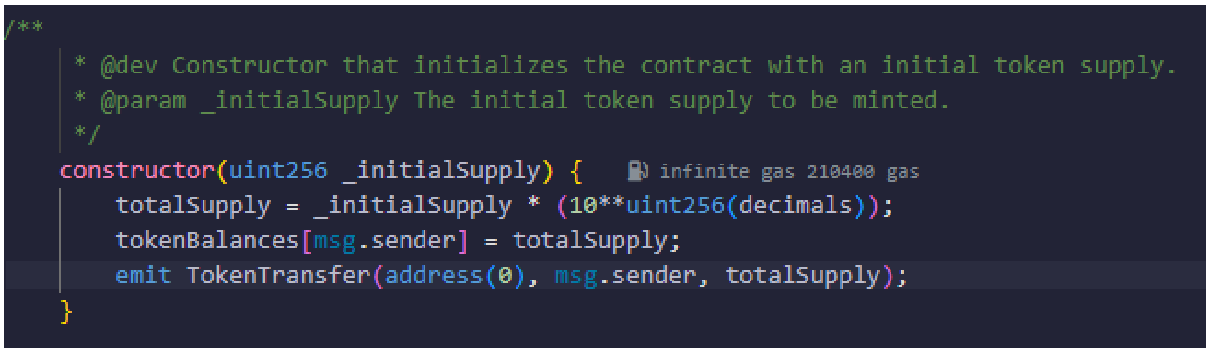


```
/**
 * @dev Modifier to check if the caller is the owner.
 */
modifier onlyOwner() {
    require(msg.sender == owner, "You don't have the permission to make this change");
    _;
}
```

Figure 24: Function Modifying to give total access to the Owner.

The modifier function works as an access control to check if a caller of a particular function has the verified access to the function they are accessing. The access here is granted to only the owner of the token.

3.1.6. Constructor



```
/**
    * @dev Constructor that initializes the contract with an initial token supply.
    * @param _initialSupply The initial token supply to be minted.
    */
    constructor(uint256 _initialSupply) {     infinite gas 210400 gas
        totalSupply = _initialSupply * (10**uint256(decimals));
        tokenBalances[msg.sender] = totalSupply;
        emit TokenTransfer(address(0), msg.sender, totalSupply);
    }
```

Figure 25: Constructor to Initialise the contract with initial token supply.

The constructor function initialises the contract to an initial token supply. It sets the total supply and assigns it to the deployer's balance when the contract is deployed. It takes a single parameter '_initialSupply' which is of type unsigned integer. The constructor is only called once in the contract creation.

The 'totalSupply' holds the total supply of the tokens. The '_initialSupply' is multiplied by 10*decimal (18) already declared when declaring the token information. For example,

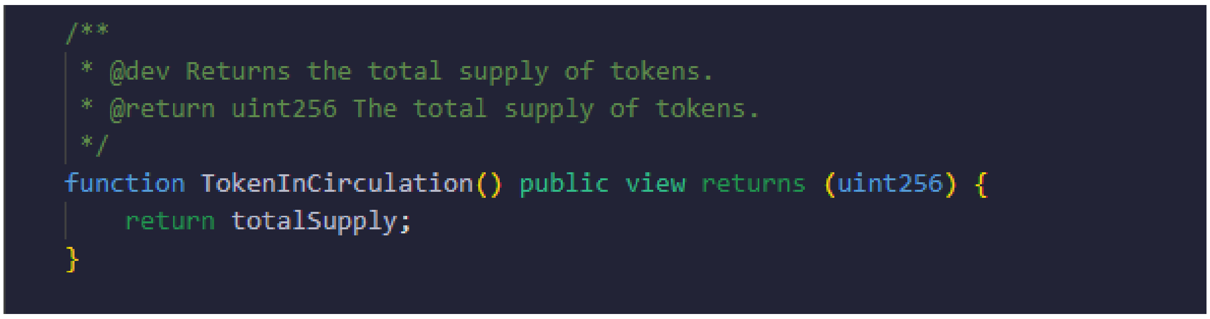
if the initial supply is 500, the totalSupply = 500 * 10**18 = 500000000000000000000.

The 'tokenBalances[msg.sender]' is global state in solidity that assigns the totalSupply token to the sender (deployer) balance.It also keep track of the addresses token balance.

The 'emit' is used to trigger an event in Solidity. It helped in logging the event of the creation of the initial supply of the token.

While the 'address(0) is a non-existence address, the event records that the totalSupply has been minted and transferred to the deployer address.

3.1.7. Total Supply

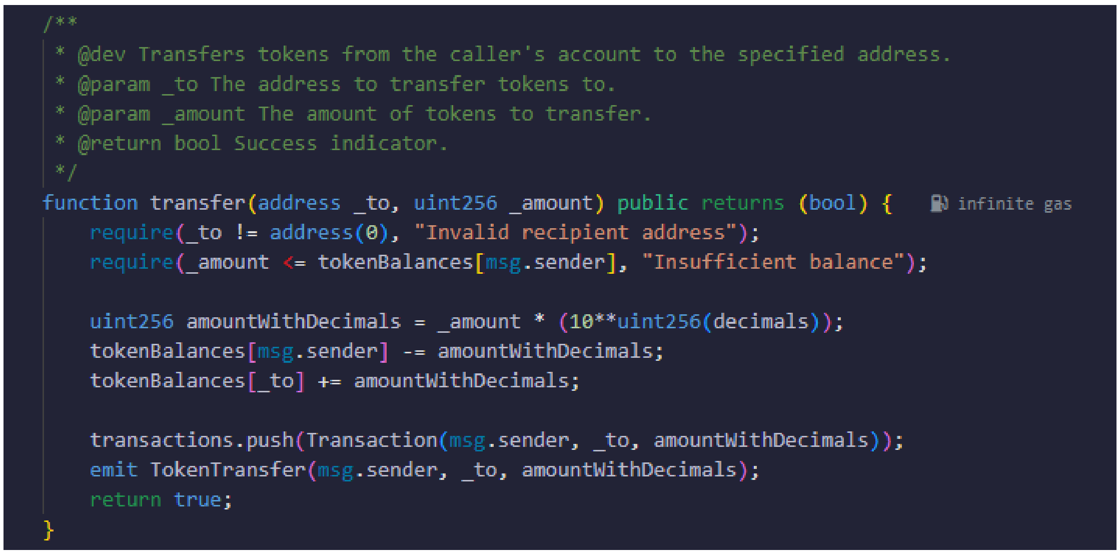


```
    /**
     * @dev Returns the total supply of tokens.
     * @return uint256 The total supply of tokens.
     */
    function TokenInCirculation() public view returns (uint256) {
        return totalSupply;
    }
```

Figure 26: Function to return the total amount of mined tokens.

The function is a public function and is called to return and verify the total token in circulation. It shows the total amount of token including both initial supply amount and the tokens that will be mined over the time.

3.1.8. Transfer



```
/**
 * @dev Transfers tokens from the caller's account to the specified address.
 * @param _to The address to transfer tokens to.
 * @param _amount The amount of tokens to transfer.
 * @return bool Success indicator.
 */
function transfer(address _to, uint256 _amount) public returns (bool) {    infinite gas
    require(_to != address(0), "Invalid recipient address");
    require(_amount <= tokenBalances[msg.sender], "Insufficient balance");

    uint256 amountWithDecimals = _amount * (10**uint256(decimals));
    tokenBalances[msg.sender] -= amountWithDecimals;
    tokenBalances[_to] += amountWithDecimals;

    transactions.push(Transaction(msg.sender, _to, amountWithDecimals));
    emit TokenTransfer(msg.sender, _to, amountWithDecimals);
    return true;
}
```

Figure 27: Function for transferring tokens between addresses.

The transfer function allows token holders to send tokens to other addresses. It's declared as a 'public' function meaning it can be call by any address with token.

The transfer function takes two parameters, '_to' and '_amount' representing the receiver address and the amount to send respectively.

The 'require' statement validates '_to' and '_amount'. If the receiver address is not the address(0), it will return the message "Invalid recipient address" and if the token balance is less than the amount trying to transfer, it will return the "Insufficient balance" message.

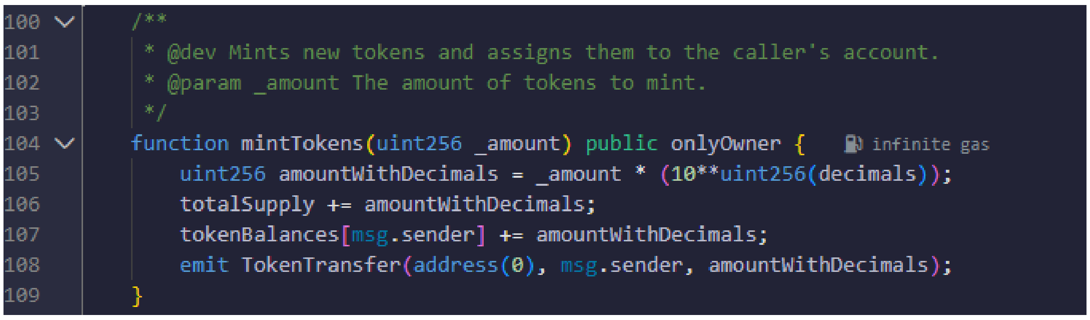
If the transaction is successful, the sender balance will decrease '-amountWithDecimals' and the receiver balance will increase '+amountWithDecimals' by the transfer amount.

'amountWithDecimals' is the transferred amount multiplied by 10**decimals (18).

The 'transaction.push' store the transaction information which includes the recipient address and the amount. While the 'emit TokenTransfer' is logged to track the transaction.

The transaction then returns 'True' if successfully.

3.1.9. Mint Token Function



```
/**
 * @dev Mints new tokens and assigns them to the caller's account.
 * @param _amount The amount of tokens to mint.
 */
function mintTokens(uint256 _amount) public onlyOwner {
    uint256 amountWithDecimals = _amount * (10**uint256(decimals));
    totalSupply += amountWithDecimals;
    tokenBalances[msg.sender] += amountWithDecimals;
    emit TokenTransfer(address(0), msg.sender, amountWithDecimals);
}
```

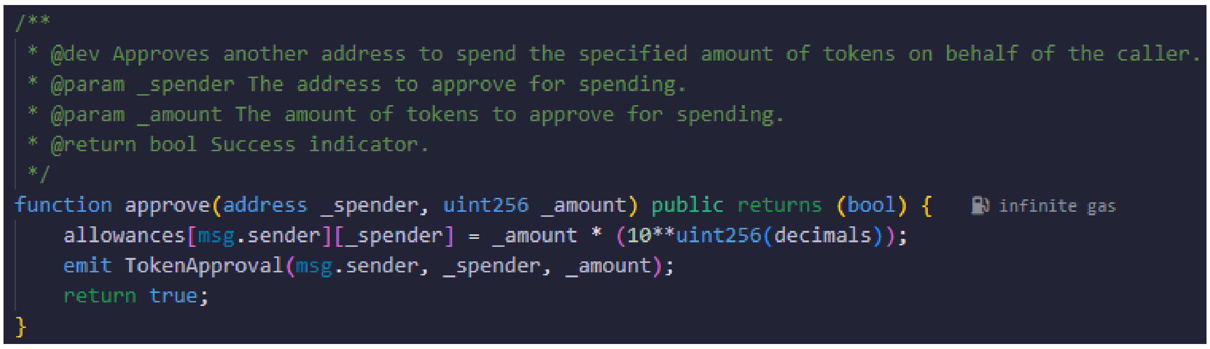
Figure 28: Function to mint new token when in need.

The ‘mintTokens’ function can only be call by an authorised user or address, the owner in this case. It takes a single parameter ‘_amount’ which is the amount to be minted.

After minting, the amount minted will be added to the initial token balance of the caller.

The function allows the creation of more tokens, thus, increasing the token balance which is useful where more tokens are needed over time.

3.1.10. Approve Function



```
/**
 * @dev Approves another address to spend the specified amount of tokens on behalf of the caller.
 * @param _spender The address to approve for spending.
 * @param _amount The amount of tokens to approve for spending.
 * @return bool Success indicator.
 */
function approve(address _spender, uint256 _amount) public returns (bool) {
    allowances[msg.sender][_spender] = _amount * (10**uint256(decimals));
    emit TokenApproval(msg.sender, _spender, _amount);
    return true;
}
```

Figure 29: Function to approve an address to spend on behalf of the owner.

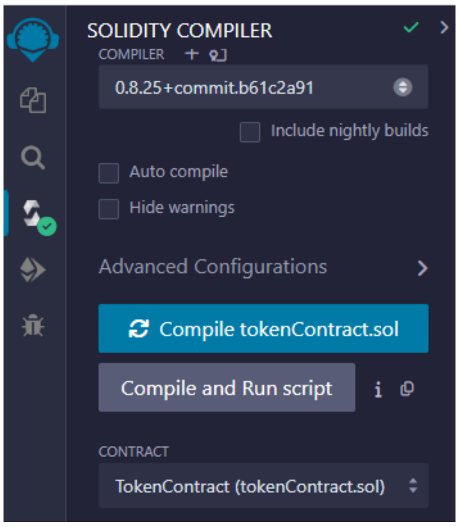
The approve function is used to grant access to another address ‘_spender’ to spend an approved amount to another address on behalf of the sender. This is useful where an owner wants to grant the ability to spend tokens to another account or contract, for example, in automated trading systems.

The approval function takes two parameters ‘_spender’ and ‘_amount’ which allows the owner to give token transfer permission to an address and specify the amount allowed to transfer. The spender can’t spend more than the specified amount.

The ‘allowances’ set the permission to the spender to spend on behalf of the owner.

3.2. Contract Deployment

After the solidity code has been written in Remix IDE, the next step is compiling the contract either by entering keyboard key *crtl+s* or by clicking on the run icon in the compiler IDE or by compiling from the "Solidity compiler" tile. The tile will show a green marked icon to signal a successful compilation.



Compiler: 0.8.25+commit.b61c2a91

Contract: TokenContract (tokenContract.sol)

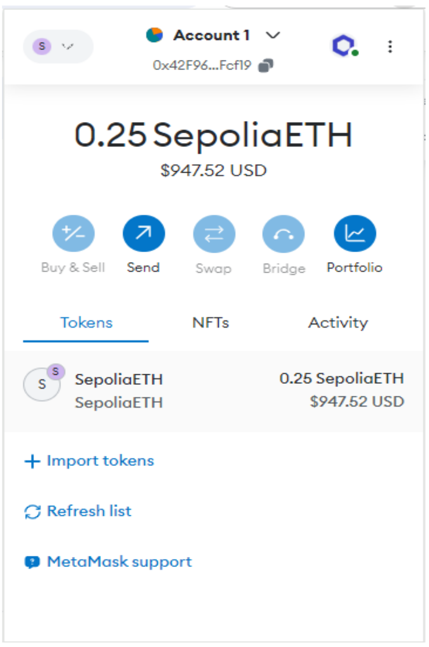
Figure 30: Compiling the smart contract.

The next step will be to deploy and run transaction. The contract deployment and transaction were done in the Remix IDE's DEPLOY AND RUN TRANSACTIONS tile.

Deployment of the compiled contract to an Ethereum test network, Sepolia, was done through Remix. The steps in deployment are highlighted below.

3.3. Sepolia Preparation:

Setting up the Sepolia test network in MetaMask and transferring of Ethereum into the wallet through the Chainlink Faucets.



0.25 SepoliaETH
$947.52 USD

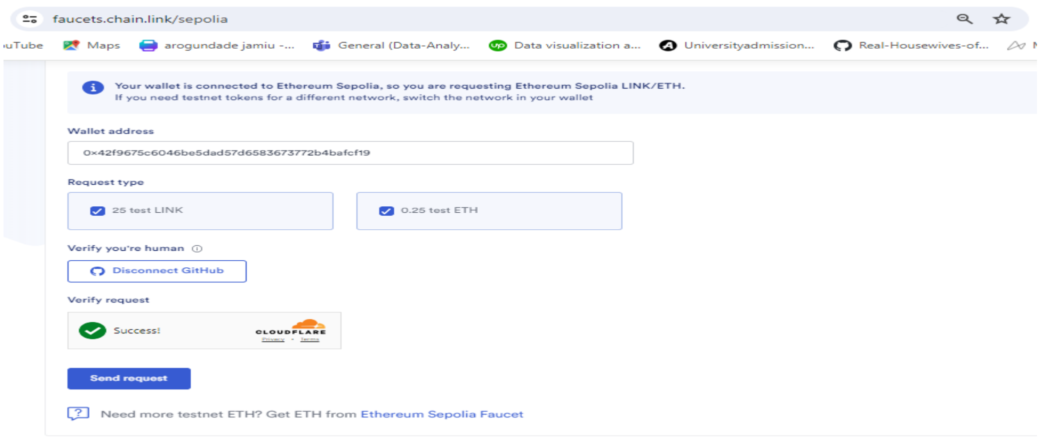
SepoliaETH
SepoliaETH
0.25 SepoliaETH
$947.52 USD

Figure 31: Sepolia Eth balance.

Sepolia was a proof-of-authority testnet created in October 2021 by Ethereum core developers and maintained ever since. Sepolia and other testnets like Goerli testnets were transitioned to a proof-of-stake consensus mechanism to mimic the Ethereum mainnet.

Testnets are created to replicate the functionality of a mainnet blockchain but are separated from the actual ledger. They are used by developers to safely test applications and smart contracts before introducing them to the live Ethereum network.

Sepolia, specifically, was made to imitate challenging network situations and has faster block times for quicker transaction confirmations and developer feedback. (Alchemy.com, 2023: Can be found on: https://www.alchemy.com/overviews/sepolia-testnet).



faucets.chain.link/sepolia

Your wallet is connected to Ethereum Sepolia, so you are requesting Ethereum Sepolia LINK/ETH.
If you need testnet tokens for a different network, switch the network in your wallet

Wallet address

0x42f9675c6046be5dad57d6583673772b4bafcf19

Request type

☑ 25 test LINK ☑ 0.25 test ETH

Verify you're human

Disconnect GitHub

Verify request

Success!

Send request

Need more testnet ETH? Get ETH from Ethereum Sepolia Faucet

Figure 32: Sending of test ETH from Chainlink Faucets.

Chainlink is a robust decentralised blockchain oracle network developed on the Ethereum platform. It plays a crucial role in securely transferring data from off-chain sources to on-chain smart contracts (available at https://chain.link/whitepaper).

3.4. Deploying Contract to Sepolia Network from Remix

Select 'Injected Provider – MetaMask' in the Environment dropdown.

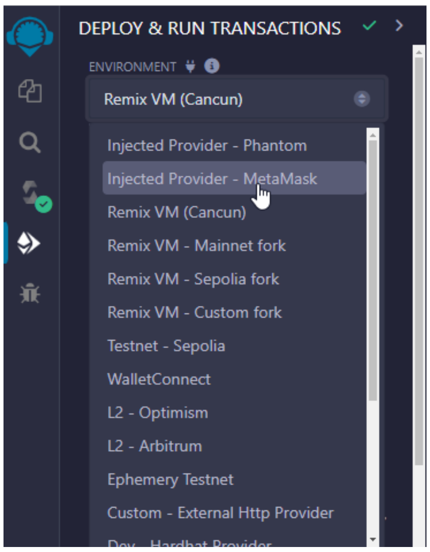
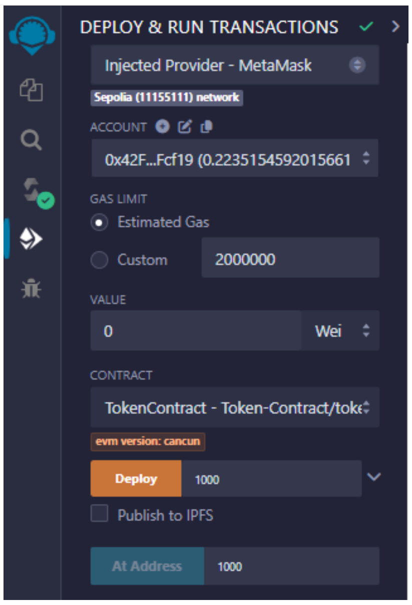


Figure 33: Deploying and running the transaction.

Enter the initial amount to supply and click Deploy.


DEPLOY & RUN TRANSACTIONS

Injected Provider - MetaMask

Sepolia (11155111) network

ACCOUNT: 0x42F...Fcf19 (0.2235154592015661

GAS LIMIT: Estimated Gas (selected); Custom 2000000

VALUE: 0 Wei

CONTRACT: TokenContract - Token-Contract/toke

evm version: cancun

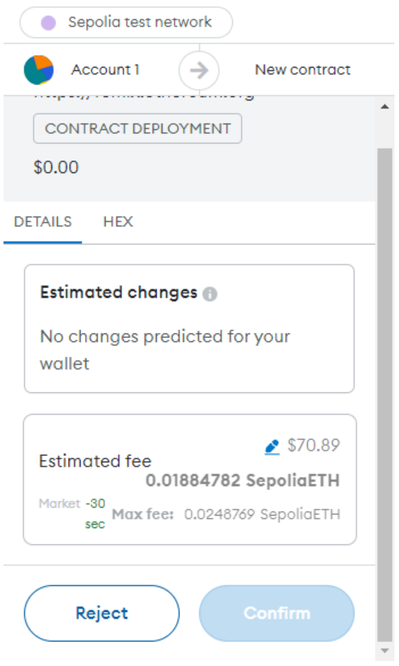
Deploy 1000

Publish to IPFS

At Address 1000

Figure 34: Entering the initial supply amount and deploying to the Sepolia TestNet on MetaMask.

After the deployment, the MetaMask app requested for confirmation sign. This confirmation will be prompted at every interaction with the Sepolia network.


Sepolia test network

Account 1 → New contract

CONTRACT DEPLOYMENT

$0.00

DETAILS HEX

Estimated changes

No changes predicted for your wallet

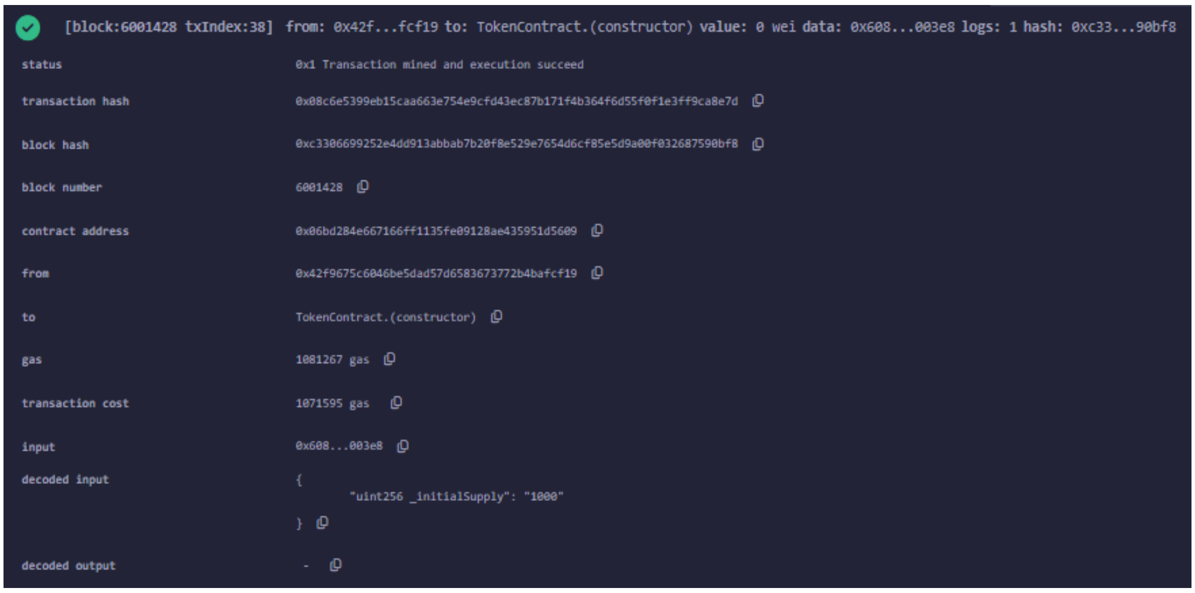
Estimated fee $70.89
0.01884782 SepoliaETH

Market -30 sec Max fee: 0.0248769 SepoliaETH

Reject Confirm

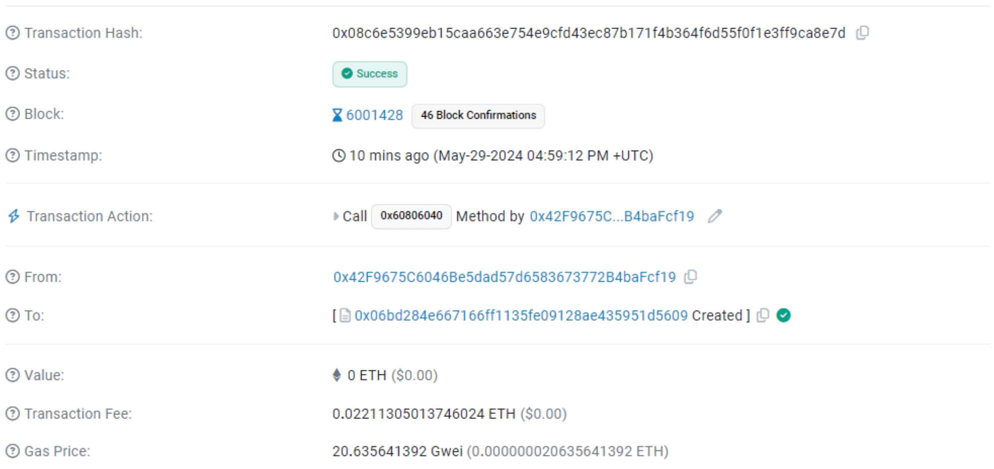
Figure 35: Confirming the transaction on MetaMask.

The contract was deployed after confirming the transaction.



```
[block:6001428 txIndex:38] from: 0x42f...fcf19 to: TokenContract.(constructor) value: 0 wei data: 0x608...003e8 logs: 1 hash: 0xc33...90bf8
status                  0x1 Transaction mined and execution succeed
transaction hash        0x08c6e5399eb15caa663e754e9cfd43ec87b171f4b364f6d55f0f1e3ff9ca8e7d
block hash              0xc3306699252e4dd913abbab7b20f8e529e7654d6cf85e5d9a00f032687590bf8
block number            6001428
contract address        0x06bd284e667166ff1135fe09128ae435951d5609
from                    0x42f9675c6046be5dad57d6583673772b4bafcf19
to                      TokenContract.(constructor)
gas                     1081267 gas
transaction cost        1071595 gas
input                   0x608...003e8
decoded input           {
                                "uint256 _initialSupply": "1000"
                        }
decoded output           -
```

Figure 36: Smart Contract Deployment Reciept on Remix Console.



Transaction Hash:	0x08c6e5399eb15caa663e754e9cfd43ec87b171f4b364f6d55f0f1e3ff9ca8e7d
Status:	Success
Block:	6001428 46 Block Confirmations
Timestamp:	10 mins ago (May-29-2024 04:59:12 PM +UTC)
Transaction Action:	Call 0x60806040 Method by 0x42F9675C...B4baFcf19
From:	0x42F9675C6046Be5dad57d6583673772B4baFcf19
To:	[0x06bd284e667166ff1135fe09128ae435951d5609 Created]
Value:	0 ETH ($0.00)
Transaction Fee:	0.02211305013746024 ETH ($0.00)
Gas Price:	20.635641392 Gwei (0.000000020635641392 ETH)

Figure 37: Smart Contract Deployment Transaction Confirmation on Etherscan.

The structure of the deployed contract:

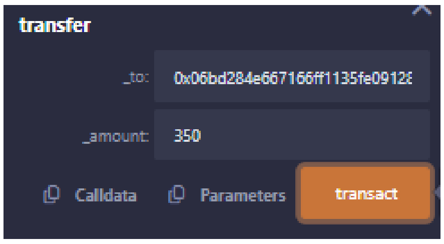
- Transaction Hash: A unique identifier for a transaction.
- Block: A data structure in the blockchain containing a group of transactions.
- Confirmations: The number of blocks added to the chain after the block containing the transaction.
- Timestamp: The time when the transaction was included in the blockchain.
- From: The sender's address.
- To: The recipient's address.
- Value: The amount of token transferred.

- Transaction Fee: The cost of processing the transaction. It is the addition of the transaction cost and the gas fee.
- Gas Price: The amount of Ether paid per unit of gas.

These components provide a detailed view of a transaction's status, origin, destination, and costs, ensuring transparency and traceability of the smart contract in the blockchain network.

3.5. Token Transfer

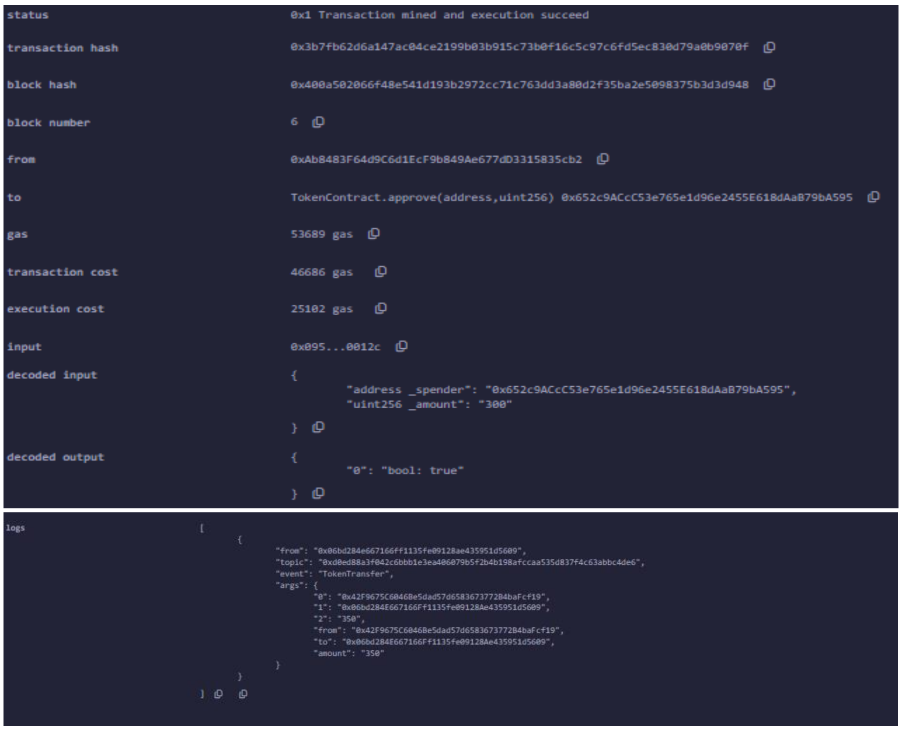
Transaction was ensured in the smart contract by transferring the token from one address to another. The receiver address and the transaction amount are required to send token from the blockchain.


transfer

_to: 0x06bd284e667166ff1135fe0912

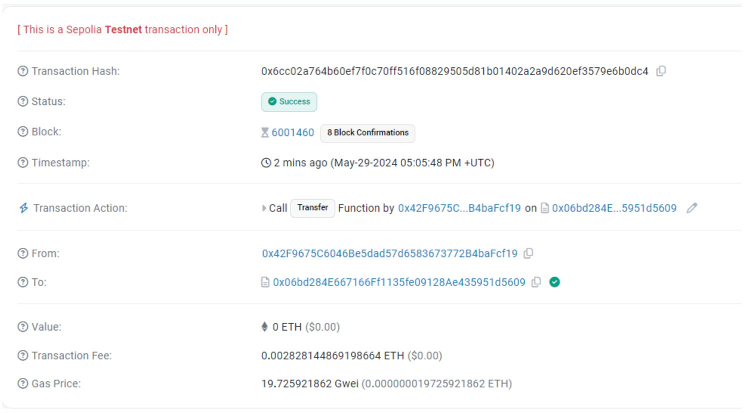
_amount: 350

Figure 38: Interacting with the Smart Contract Transferring the Token to another Address.


```
status                  0x1 Transaction mined and execution succeed
transaction hash        0x3b7fb62d6a147ac04ce2199b03b915c73b0f16c5c97c6fd5ec830d79a0b9070f
block hash              0x400a502066f48e541d193b2972cc71c763dd3a80d2f35ba2e5098375b3d3d948
block number            6
from                    0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to                      TokenContract.approve(address,uint256) 0x652c9ACcC53e765e1d96e2455E618dAaB79bA595
gas                     53689 gas
transaction cost        46686 gas
execution cost          25102 gas
input                   0x095...0012c
decoded input           {
                                "address _spender": "0x652c9ACcC53e765e1d96e2455E618dAaB79bA595",
                                "uint256 _amount": "300"
                        }
decoded output          {
                                "0": "bool: true"
                        }
```

```
logs                    [
                                {
                                        "from": "0x06bd284e667166ff1135fe09128ae435951d5609",
                                        "topic": "0xd0ed88a3f042c6bbb1e3ea406079b5f2b4b198afccaa535d837f4c63abbc4de6",
                                        "event": "TokenTransfer",
                                        "args": {
                                                "0": "0x42F9675C6046Be5dad57d6583673772B4baFcf19",
                                                "1": "0x06bd284E667166Ff1135fe09128Ae435951d5609",
                                                "2": "350",
                                                "from": "0x42F9675C6046Be5dad57d6583673772B4baFcf19",
                                                "to": "0x06bd284E667166Ff1135fe09128Ae435951d5609",
                                                "amount": "350"
                                        }
                                }
                        ]
```

Figure 39: Smart Contract Successful Transfer Confirmation/Log on Remix.


[This is a Sepolia **Testnet** transaction only]

Transaction Hash: 0x6cc02a764b60ef7f0c70ff516f08829505d81b01402a2a9d620ef3579e6b0dc4
Status: Success
Block: 6001460 8 Block Confirmations
Timestamp: 2 mins ago (May-29-2024 05:05:48 PM +UTC)
Transaction Action: Call Transfer Function by 0x42F9675C...B4baFcf19 on 0x06bd284E...5951d5609
From: 0x42F9675C6046Be5dad57d6583673772B4baFcf19
To: 0x06bd284E667166Ff1135fe09128Ae435951d5609
Value: 0 ETH ($0.00)
Transaction Fee: 0.002828144869198664 ETH ($0.00)
Gas Price: 19.725921862 Gwei (0.000000019725921862 ETH)

Figure 40: Smart Contract Successful Transfer Confirmation/Log on Etherscan.

3.6. Mining New Token

In the need of mining new token, the function to mine new tokens comes in handy in the smart contract. The function similarly as the initial supply token function except this is called manually.

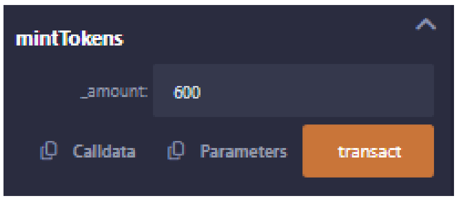

mintTokens
_amount: 600

Figure 41: Interacting with the Smart Contract by Minting Additional Tokens.

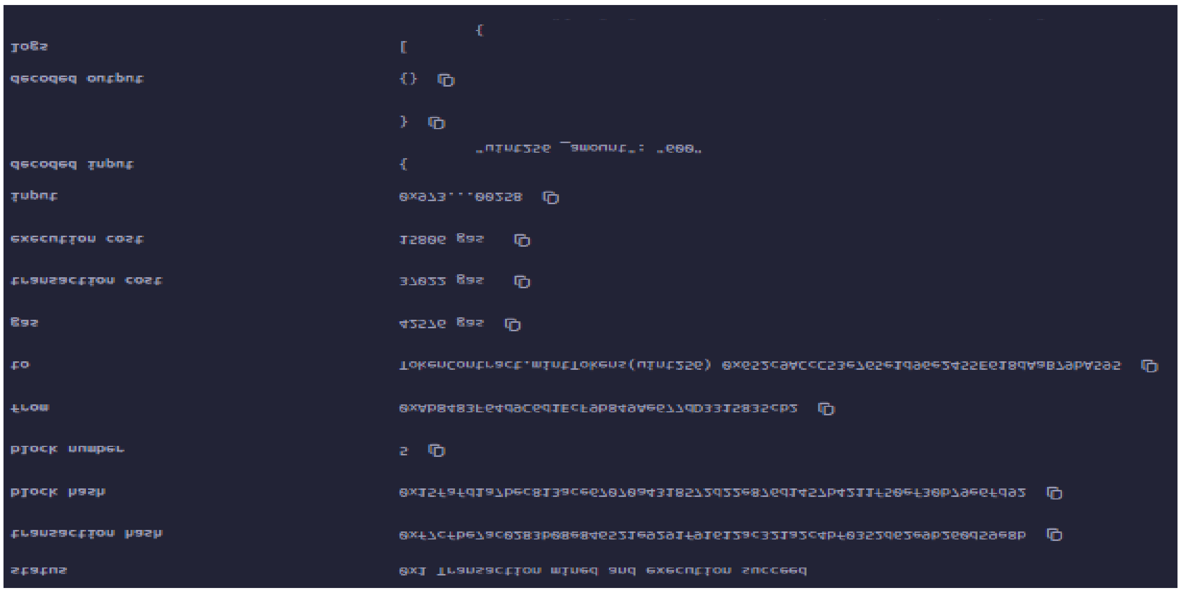
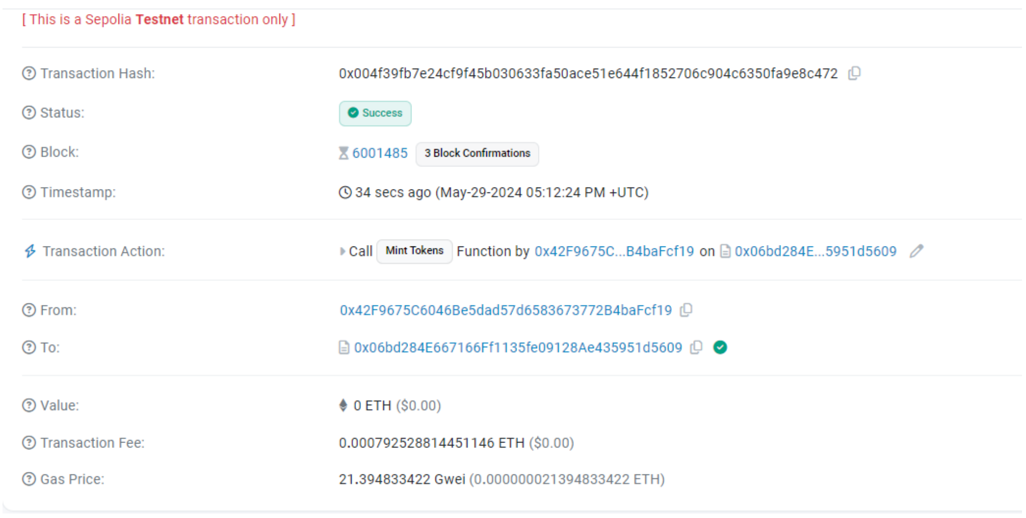

```
logs                 [
decoded output       {}
                     {
                       "uint256 _amount": "600"
decoded input        }
input                [illegible]
execution cost       [illegible] gas
transaction cost     [illegible] gas
gas                  [illegible] gas
to                   TokenContract.mintTokens(uint256) [illegible]
from                 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
block number         5
block hash           [illegible]
transaction hash     [illegible]
status               0x1 Transaction mined and execution succeed
```

Figure 42:Confirmation Log of Additional Token Mining on Remix.



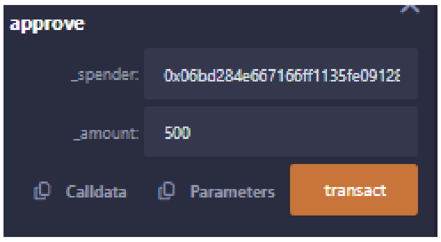
[This is a Sepolia **Testnet** transaction only]

Transaction Hash:	0x004f39fb7e24cf9f45b030633fa50ace51e644f1852706c904c6350fa9e8c472
Status:	Success
Block:	6001485 3 Block Confirmations
Timestamp:	34 secs ago (May-29-2024 05:12:24 PM +UTC)
Transaction Action:	Call Mint Tokens Function by 0x42F9675C...B4baFcf19 on 0x06bd284E...5951d5609
From:	0x42F9675C6046Be5dad57d6583673772B4baFcf19
To:	0x06bd284E667166Ff1135fe09128Ae435951d5609
Value:	0 ETH ($0.00)
Transaction Fee:	0.000792528814451146 ETH ($0.00)
Gas Price:	21.394833422 Gwei (0.000000021394833422 ETH)

Figure 43: Confirmation Log of Additional Token Mining on Etherscan.

3.7. Approval

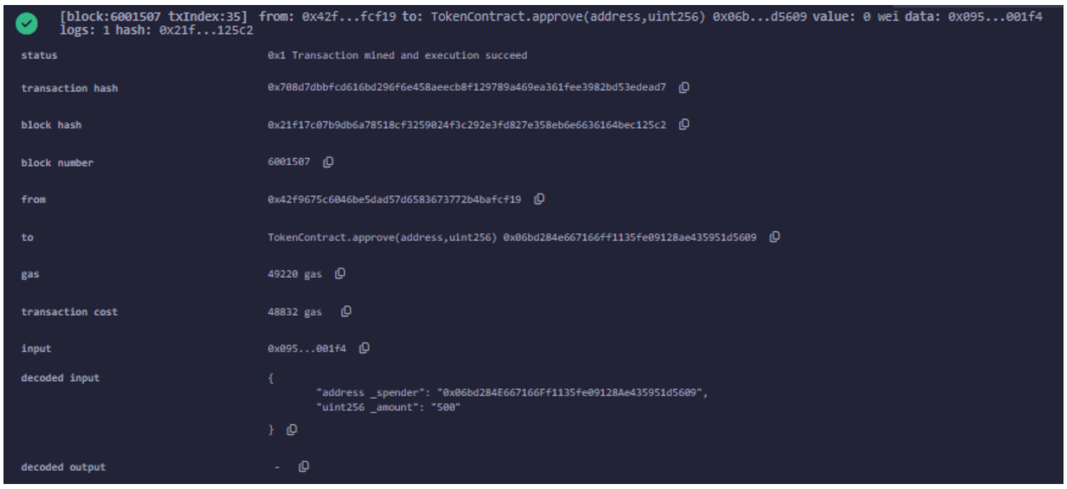
The purpose of the approve function is for the owner of the smart contract giving permission to another account/address to spend on their behalf. As explained earlier in part of the report, the approved address won't be able to spend more than the amount specially approved for them.



approve

_spender:	0x06bd284e667166ff1135fe0912
_amount:	500

Figure 44: Interacting with the Smart Contract by Giving Approval to Another Account to Send out Tokens on Behalf of the Owner.



```
[block:6001507 txIndex:35]  from: 0x42f...fcf19 to: TokenContract.approve(address,uint256) 0x06b...d5609 value: 0 wei data: 0x095...001f4
logs: 1 hash: 0x21f...125c2
status                  0x1 Transaction mined and execution succeed
transaction hash        0x708d7dbbfcd616bd296f6e458aeecb8f129789a469ea361fee3982bd53edead7
block hash              0x21f17c07b9db6a78518cf3259024f3c292e3fd827e358eb6e6636164bec125c2
block number            6001507
from                    0x42f9675c6046be5dad57d6583673772b4bafcf19
to                      TokenContract.approve(address,uint256) 0x06bd284e667166ff1135fe09128ae435951d5609
gas                     49220 gas
transaction cost        48832 gas
input                   0x095...001f4
decoded input           {
                                "address _spender": "0x06bd284E667166Ff1135fe09128Ae435951d5609",
                                "uint256 _amount": "500"
                        }
decoded output           -
```

Figure 45: Log Confirming Successfully Approving Another Account to Send out Tokens on Behalf of the Owner.

4. Testing

Test Case Logging for Blockchain Simulation (Python).

Test Case No.	Test Case Description	Expected Output	Exact Output
Test0	Declaring the Block properties.	Hash ID, Data, Previous Hash, Timestamp, and Nonce.	✓ Test Successful.
Test1	Adding new Block for Proof of Work.	Hash ID, Data, Previous Hash, Timestamp, and Nonce.	✓ Test Successful.
Test2	Adding new Block for Proof of Stake.	Hash ID, Data (Validator), Previous Hash, and Timestamp.	✓ Test Successful.
Test3	Validating Proof of Work.	“Blockchain is valid”	✓ Test Successful.
Test4	Validating Proof of Stake.	“Blockchain is valid”	✓ Test Successful.

Table 2: Test Case for Blockchain Simulation in Python.

Test Case Logging for Smart Contract (Solidity).

Test Case No.	Test Case Description	Expected Output	Exact Output
Test0	Declaring of the smart contract variables for deployment.	Transaction Hash, Block Hash, Block number, Timestamp, Sender's address, Receiver's address, Value, Transaction fee, Transaction cost, Gas price, and Input.	✓ Test Successful.
Test1	Transferring tokens to another address.	Transaction Hash, Block Hash, Block number, Timestamp, Sender's address, Receiver's address, Value, Transaction fee, Transaction cost, Gas price, and Input.	✓ Test Successful.
Test2	Minting New Tokens	Transaction Hash, Block Hash, Block number, Timestamp, Sender's address, Receiver's address, Value, Transaction fee, Transaction cost, Gas price, and Input.	✓ Test Successful.
Test3	Approving another address to spend on behalf of the token owner	Transaction Hash, Block Hash, Block number, Timestamp, Sender's address, Receiver's address, Value, Transaction fee, Gas price, and Input.	✓ Test Successful.

Table 3: Test Case for Smart Contract Deployment (Solidity).

5. Limitations and Caveats

Proof of Work (PoW) Consensus Mechanism: PoW involves solving complex puzzles using a lot of computational power, which can make the process slow and energy intensive. This can result in high electricity costs and environmental concerns. Proof of Stake (PoS) was created to address these issues.

Proof of Stake (PoS) Consensus Mechanism: PoS tends to benefit those with the largest stakes, potentially leading to centralisation. Early adopters or loyal users with smaller stakes may struggle to compete for rewards, reducing their incentives. It may be necessary to provide special rewards for loyal users without high staking power.

High Gas Fees for Smart Contract Deployment: Deploying and running smart contracts on the Ethereum network can be expensive due to gas fees, which increase with the complexity and number of functions within the contract. This could make it too costly for some use cases. The widespread adoption of the Web3 could potentially eliminate the concept of gas fees altogether.

Vulnerability to Bugs and Attacks: Smart contracts are susceptible to bugs and exploits by malicious actors, which can lead to potential financial losses and security breaches. It is important to check for access control and implement re-entrancy guards to address security issues in smart contracts.

Requirement of Specialised Knowledge: Developing and implementing blockchain solutions requires extensive knowledge of cryptography, which is not widely available. This makes it challenging for many organisations to effectively implement blockchain technology.

Immutability: While the immutability of blockchain ensures data integrity, it also means that errors or fraudulent transactions cannot be reversed, which can be problematic in certain situations. Special functions can be designed to retrieve transactions if the caller has the essential security access to support their claim.

6. Conclusion

The project objective is to prioritise the development and implementation of an advanced Blockchain solution customised for the optimisation of Future Net Ltd.'s supply chain operations. The key advantage of deploying Blockchain technology is its ability to ensure transparency and efficiency throughout the network. In addition, utilising Smart Contracts helps to streamline transactions and reinforce security protocols.

The results of a series of tests clearly indicate that the blockchain solution can be seamlessly incorporated into real-world scenarios. However, it is crucial to recognise that, despite the number of benefits linked to Blockchain technology and Smart Contracts, there are certain built-in constraints and obstacles.

It is important for Future Net Ltd to have a complete understanding of these limitations as part of their implementation strategy, allowing them to establish the necessary precautions and formulate effective strategies for addressing these challenges.

7. References

Alchemy.com, h.-t., 2023. *What is the Sepolia Testnet?,* s.l.: Available at https://www.alchemy.com/overviews/sepolia-testnet.

Breidenbach, L. C. C. B. C. A. C. S. E. A. J. F. K. A. M. B. M. D. M. S. N. A. T. F. T. a. F. Z., 2021. *Chainlink 2.0: Next Steps,* s.l.: Available at https://chain.link/whitepaper.

Dong, S. A. K. L. M. &. K. J., 2023. Blockchain technology and application: An overview. PeerJ Computer Science, 9. https://doi.org/10.7717/peerj-cs.1705.

Frank, A. G. D. L. S. &. A. N. F., 2019. *Industry 4.0 technologies: Implementation patterns in manufacturing companies..* s.l.:International Journal of Production Economics, 210, 15-26. https://doi.org/10.1016/j.ijpe.2019.01.004.

Gemeliarana, I. a. S. R., 2018. Evaluation of proof of work (POW) blockchains security network on selfish mining. *International seminar on research of information technology and intelligent systems (ISRITI),* Issue IEEE, pp. 126-130.

Gervais, A. a. K. G. O. a. W. K. a. G. V. a. R. H. a. C. S., 2016. *On the Security and Performance of Proof of Work Blockchains.* New York, NY, USA: Association for Computing Machinery.

Hazra, A. &. A. A. &. A. M., 2022. *Blockchain-aided Integrated Edge Framework of Cybersecurity for Internet of Things.,* s.l.: IEEE Consumer Electronics Magazine. PP. 1-1. 10.1109/MCE.2022.3141068..

Hazra, A. &. A. A. &. A. M., 2022. Blockchain-aided Integrated Edge Framework of Cybersecurity for Internet of Things. IEEE Consumer Electronics Magazine. PP. 1-1. 10.1109/MCE.2022.3141068..

Hofmann, E. &. R. M., 2017. Industry 4.0 and the current status as well as future prospects on logistics.. *Computers in Industry,* Issue https://doi.org/10.1016/j.compind.2017.04.002, pp. 89, 23-34..

Karafiloski E, M. A., 2017. Blockchain solutions for big data challenges: A literature review. In: *IEEE EUROCON 2017 -17th International Conference on Smart Technologies.* s.l.:IEEE, pp. 763-768.

Liang, Y.-C., 2020. Blockchain for Dynamic Spectrum Management. In: *Dynamic Spectrum Management, Signals and Communication.* s.l.:s.n., pp. 121-146.

Li, W. A. S. B. J. K. G., 2017. Securing Proof-of-Stake Blockchain Protocols.. *Lecture Notes in Computer Science,* 10436(Springer, Cham. https://doi.org/10.1007/978-3-319-67816-0_17).

Li, X. J. P. C. T. L. X. &. W. Q., 2020. A Survey on the Security of Blockchain Systems.. In: *Future Generation Computer Systems, 107.* s.l.:https://doi.org/10.1016/j.future.2017.08.020, pp. 841-853.

Manimuthu, A. V. G. V. Y. S. V. R. S. a. S. C. L. K., 2021. *Design and Development of Automobile Assembly Model Using Federated Artificial Intelligence with Smart Contract..* s.l.:International Journal of Production Research 0 (0): 1–25..

Manupati, V. K. T. S. M. R. S. P. Y. S. a. P. M., 2022. *"Recovery Strategies for a Disrupted Supply Chain Network: Leveraging Blockchain Technology in Pre- and Post-disruption Scenarios.".* s.l.:International Journal of Production Economics 245 (March): 108389..

Mourouzis, T. &. T. J., 2019. Introduction to Decentralization and Smart Contracts..

Naheed Khan, S. L. F. G.-G. C. B. E. &. B.-H. A., 2021. Blockchain smart contracts: Applications, challenges, and future trends.. Issue ncbi.nlm.nih.gov.

Sharabati, A.-A. A. a. E. R. J., 2024.. Blockchain Technology Implementation in Supply Chain Management: A Literature Review. *Sustainability.*

Sheth, H. a. D. J., 2019. “Overview of Blockchain Technology”. *Asian Journal For Convergence In Technology (AJCT). Available at: https://asianssr.org/index.php/ajct/article/view/728 (Accessed: 28May2024).,* Issue ISSN -2350-1146.

Shi, N., 2016. A new proof-of-work mechanism for bitcoin. *Financial Innovation, 2,* pp. 1-8.

Sultan, K. R. U. a. L. R., 2018. Conceptualizing blockchains: Characteristics & applications. arXiv preprint arXiv:1806.03693..

Swan, M., 2015. *Blockchain: Blueprint for a New Economy.* Sebastopol, CA: O’Reilly Media, Inc..

Szabo, N., 2016. *“Smart Contracts: 12 Use Cases for Business & Beyond A Technology, Legal & Regulatory Introduction.”.* Washington, D.C., United States: Chamber of Digital Commerce. https://digitalchamber.org/chamber-of-digital-commerce-releases-smart-contracts-white-paper/.

Wang, J. L. X. L. Y. H. Y. &. Y. X., 2021. *Blockchain-enabled wireless communications: a new paradigm towards 6G.* s.l.:s.n.

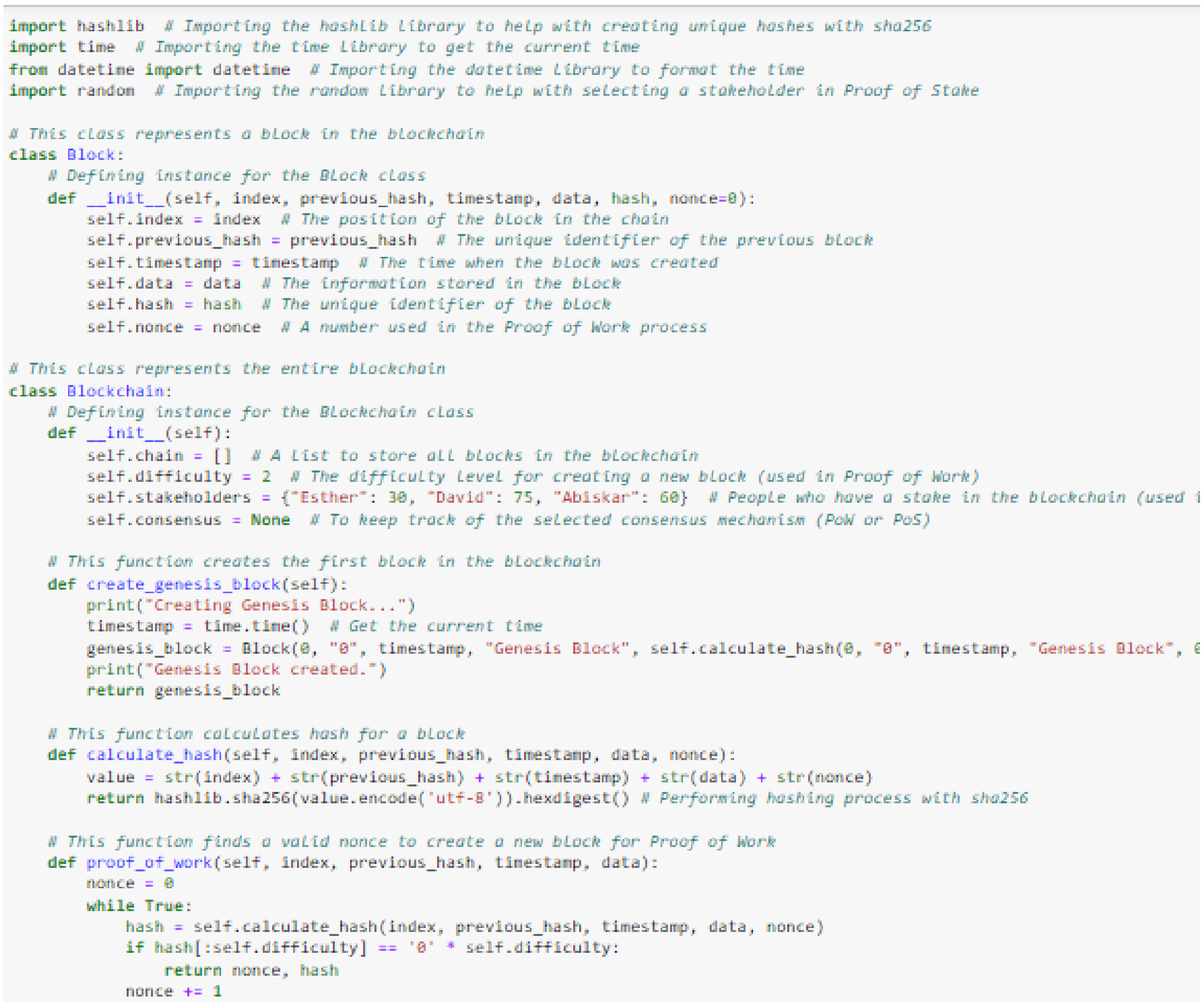
Wang, W. H. D. H. P. X. Z. N. D. W. P. W. Y. a. K. D., 2019. A survey on consensus mechanisms and mining strategy management in blockchain networks.. Issue Ieee Access, 7, pp.22328-22370..

Yaga, D. M. P. R. N. &. S. K., 2019. Blockchain Tecknology Overview.

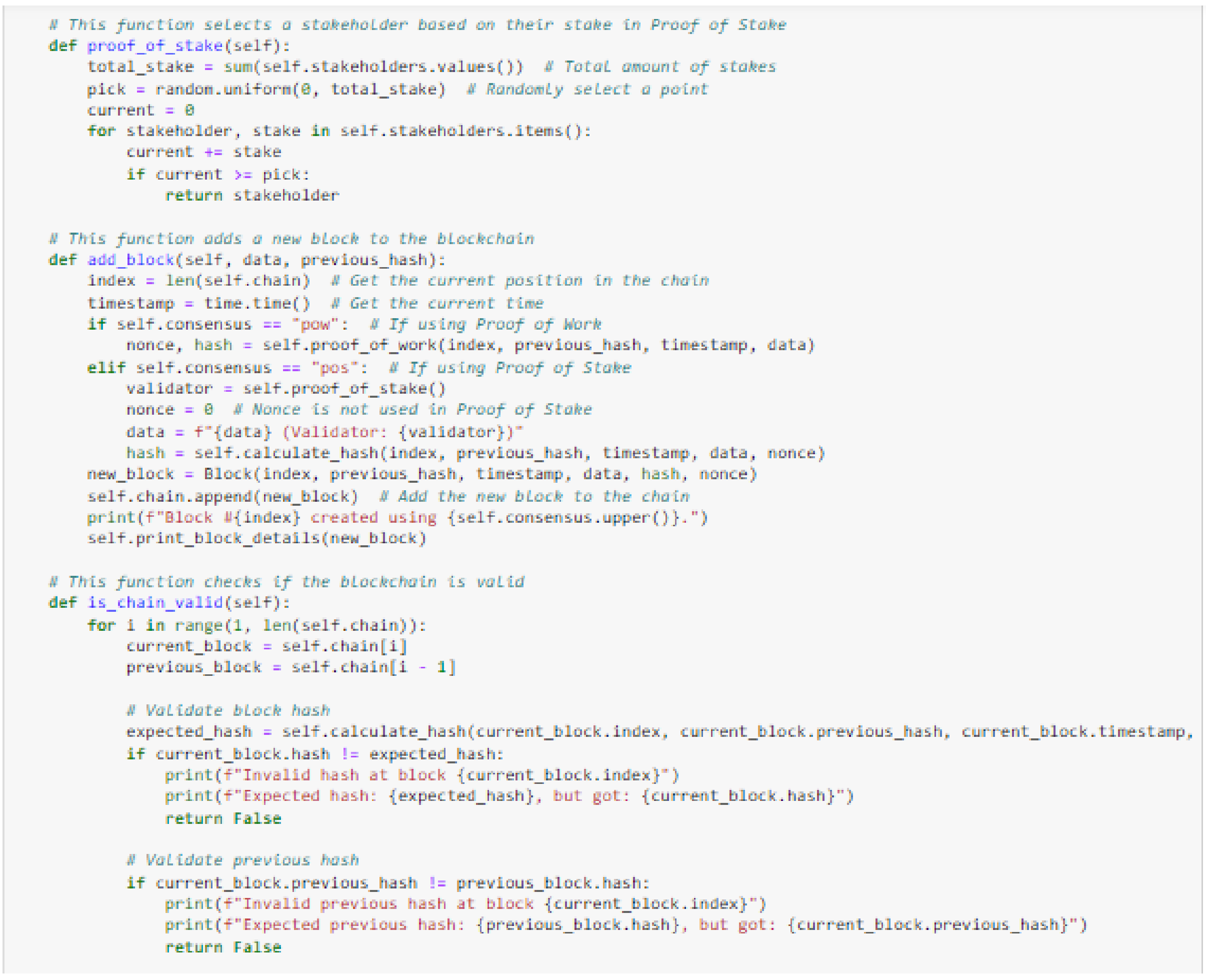
Ye C, L. G. C. H. G. Y. F. A., 2018. Analysis of security in blockchain: case study in 51%-attack detecting. *5th International Conference on Dependable Systems and Their Applications (DSA),* p. 15–24.

8. Appendices

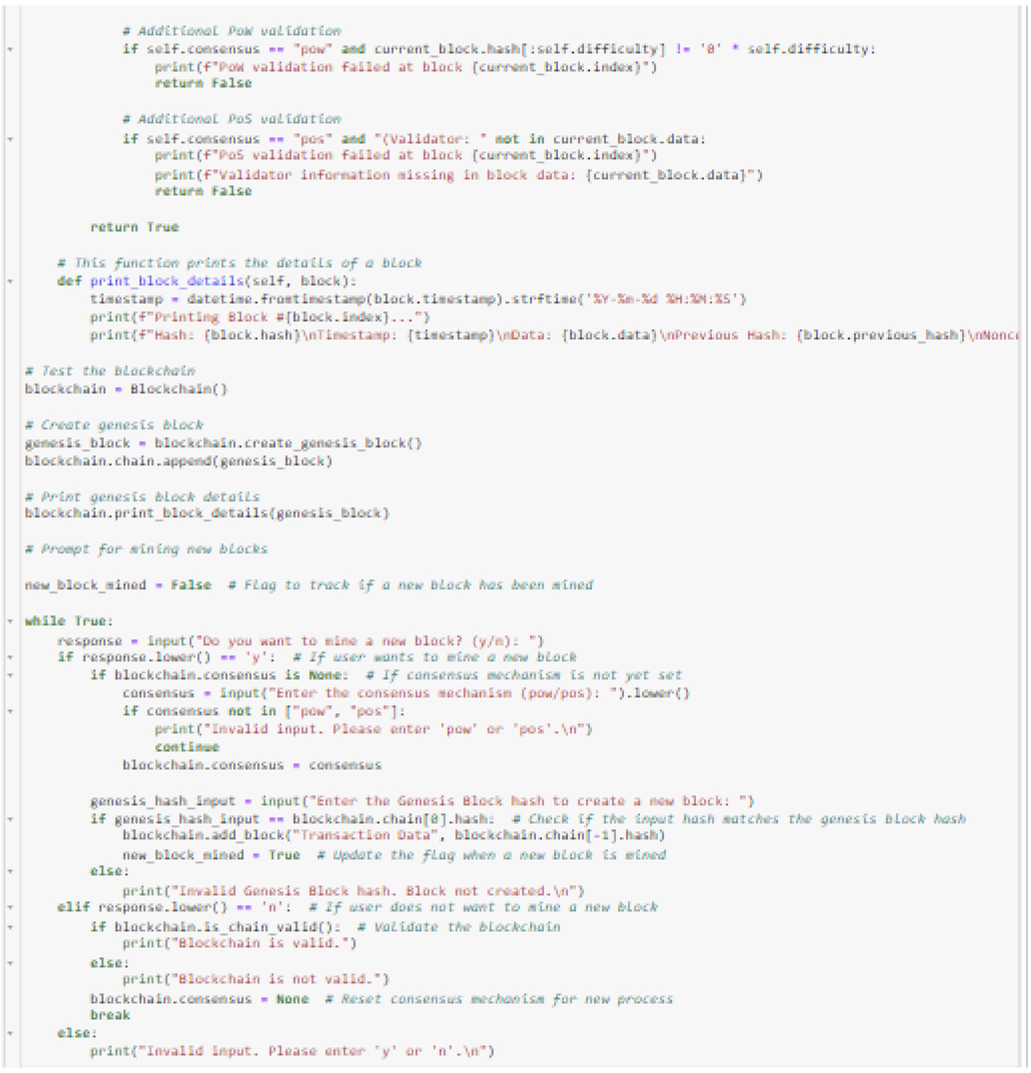
Blockchain (Python)



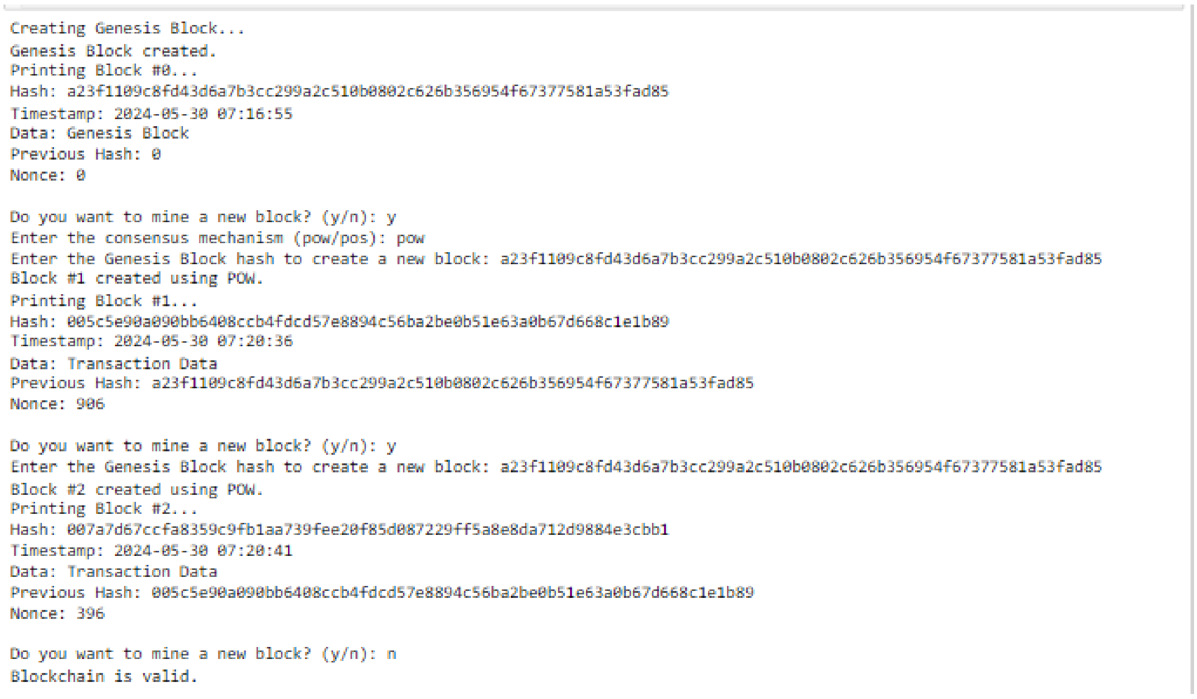
```
import hashlib  # Importing the hashlib library to help with creating unique hashes with sha256
import time  # Importing the time library to get the current time
from datetime import datetime  # Importing the datetime library to format the time
import random  # Importing the random library to help with selecting a stakeholder in Proof of Stake

# This class represents a block in the blockchain
class Block:
    # Defining instance for the Block class
    def __init__(self, index, previous_hash, timestamp, data, hash, nonce=0):
        self.index = index  # The position of the block in the chain
        self.previous_hash = previous_hash  # The unique identifier of the previous block
        self.timestamp = timestamp  # The time when the block was created
        self.data = data  # The information stored in the block
        self.hash = hash  # The unique identifier of the block
        self.nonce = nonce  # A number used in the Proof of Work process

# This class represents the entire blockchain
class Blockchain:
    # Defining instance for the Blockchain class
    def __init__(self):
        self.chain = []  # A list to store all blocks in the blockchain
        self.difficulty = 2  # The difficulty level for creating a new block (used in Proof of Work)
        self.stakeholders = {"Esther": 30, "David": 75, "Abiskar": 60}  # People who have a stake in the blockchain (used i
        self.consensus = None  # To keep track of the selected consensus mechanism (PoW or PoS)

    # This function creates the first block in the blockchain
    def create_genesis_block(self):
        print("Creating Genesis Block...")
        timestamp = time.time()  # Get the current time
        genesis_block = Block(0, "0", timestamp, "Genesis Block", self.calculate_hash(0, "0", timestamp, "Genesis Block", 0
        print("Genesis Block created.")
        return genesis_block

    # This function calculates hash for a block
    def calculate_hash(self, index, previous_hash, timestamp, data, nonce):
        value = str(index) + str(previous_hash) + str(timestamp) + str(data) + str(nonce)
        return hashlib.sha256(value.encode('utf-8')).hexdigest() # Performing hashing process with sha256

    # This function finds a valid nonce to create a new block for Proof of Work
    def proof_of_work(self, index, previous_hash, timestamp, data):
        nonce = 0
        while True:
            hash = self.calculate_hash(index, previous_hash, timestamp, data, nonce)
            if hash[:self.difficulty] == '0' * self.difficulty:
                return nonce, hash
            nonce += 1
```



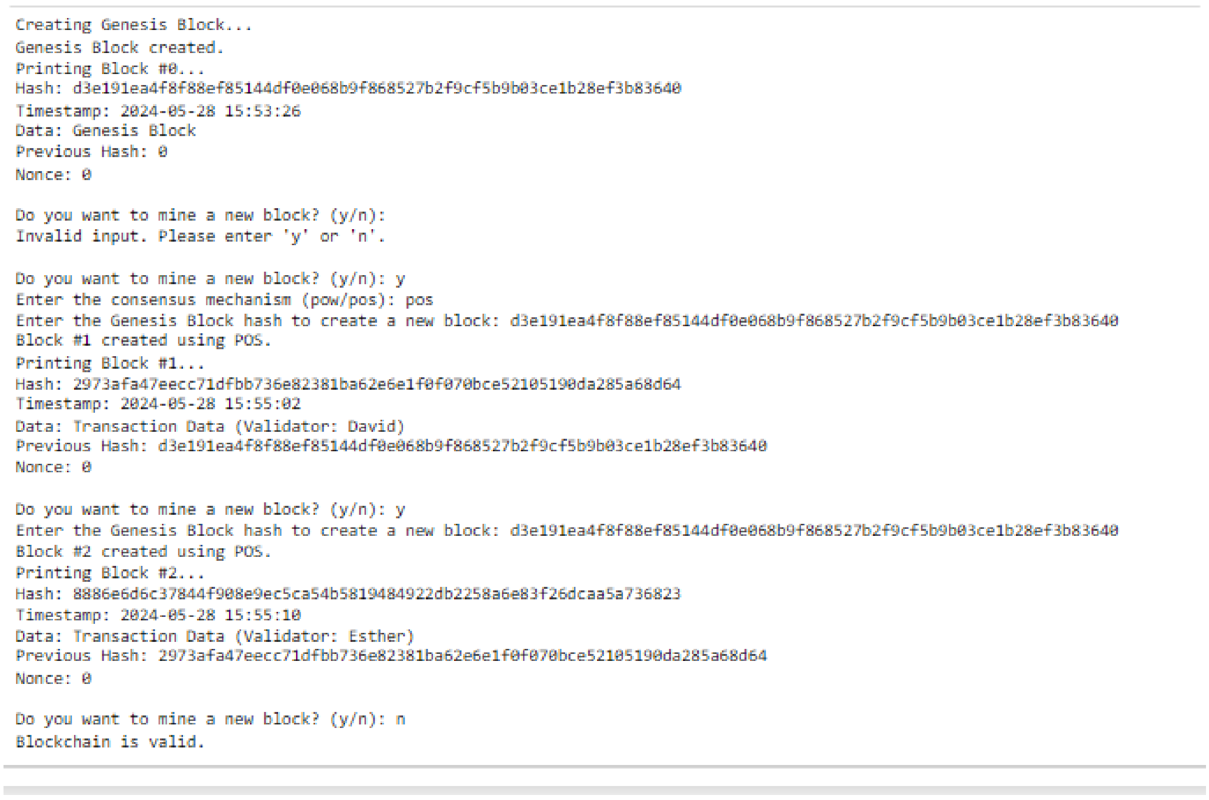
```
    # This function selects a stakeholder based on their stake in Proof of Stake
    def proof_of_stake(self):
        total_stake = sum(self.stakeholders.values())  # Total amount of stakes
        pick = random.uniform(0, total_stake)  # Randomly select a point
        current = 0
        for stakeholder, stake in self.stakeholders.items():
            current += stake
            if current >= pick:
                return stakeholder

    # This function adds a new block to the blockchain
    def add_block(self, data, previous_hash):
        index = len(self.chain)  # Get the current position in the chain
        timestamp = time.time()  # Get the current time
        if self.consensus == "pow":  # If using Proof of Work
            nonce, hash = self.proof_of_work(index, previous_hash, timestamp, data)
        elif self.consensus == "pos":  # If using Proof of Stake
            validator = self.proof_of_stake()
            nonce = 0  # Nonce is not used in Proof of Stake
            data = f"{data} (Validator: {validator})"
            hash = self.calculate_hash(index, previous_hash, timestamp, data, nonce)
        new_block = Block(index, previous_hash, timestamp, data, hash, nonce)
        self.chain.append(new_block)  # Add the new block to the chain
        print(f"Block #{index} created using {self.consensus.upper()}.")
        self.print_block_details(new_block)

    # This function checks if the blockchain is valid
    def is_chain_valid(self):
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i - 1]

            # Validate block hash
            expected_hash = self.calculate_hash(current_block.index, current_block.previous_hash, current_block.timestamp,
            if current_block.hash != expected_hash:
                print(f"Invalid hash at block {current_block.index}")
                print(f"Expected hash: {expected_hash}, but got: {current_block.hash}")
                return False

            # Validate previous hash
            if current_block.previous_hash != previous_block.hash:
                print(f"Invalid previous hash at block {current_block.index}")
                print(f"Expected previous hash: {previous_block.hash}, but got: {current_block.previous_hash}")
                return False
```



```
            # Additional PoW validation
            if self.consensus == "pow" and current_block.hash[:self.difficulty] != '0' * self.difficulty:
                print(f"PoW validation failed at block {current_block.index}")
                return False

            # Additional PoS validation
            if self.consensus == "pos" and "(Validator: " not in current_block.data:
                print(f"PoS validation failed at block {current_block.index}")
                print(f"Validator information missing in block data: {current_block.data}")
                return False

        return True

    # This function prints the details of a block
    def print_block_details(self, block):
        timestamp = datetime.fromtimestamp(block.timestamp).strftime('%Y-%m-%d %H:%M:%S')
        print(f"Printing Block #{block.index}...")
        print(f"Hash: {block.hash}\nTimestamp: {timestamp}\nData: {block.data}\nPrevious Hash: {block.previous_hash}\nNonce

# Test the blockchain
blockchain = Blockchain()

# Create genesis block
genesis_block = blockchain.create_genesis_block()
blockchain.chain.append(genesis_block)

# Print genesis block details
blockchain.print_block_details(genesis_block)

# Prompt for mining new blocks

new_block_mined = False  # Flag to track if a new block has been mined

while True:
    response = input("Do you want to mine a new block? (y/n): ")
    if response.lower() == 'y':  # If user wants to mine a new block
        if blockchain.consensus is None:  # If consensus mechanism is not yet set
            consensus = input("Enter the consensus mechanism (pow/pos): ").lower()
            if consensus not in ["pow", "pos"]:
                print("Invalid input. Please enter 'pow' or 'pos'.\n")
                continue
            blockchain.consensus = consensus

        genesis_hash_input = input("Enter the Genesis Block hash to create a new block: ")
        if genesis_hash_input == blockchain.chain[0].hash:  # Check if the input hash matches the genesis block hash
            blockchain.add_block("Transaction Data", blockchain.chain[-1].hash)
            new_block_mined = True  # Update the flag when a new block is mined
        else:
            print("Invalid Genesis Block hash. Block not created.\n")
    elif response.lower() == 'n':  # If user does not want to mine a new block
        if blockchain.is_chain_valid():  # Validate the blockchain
            print("Blockchain is valid.")
        else:
            print("Blockchain is not valid.")
        blockchain.consensus = None  # Reset consensus mechanism for new process
        break
    else:
        print("Invalid input. Please enter 'y' or 'n'.\n")
```

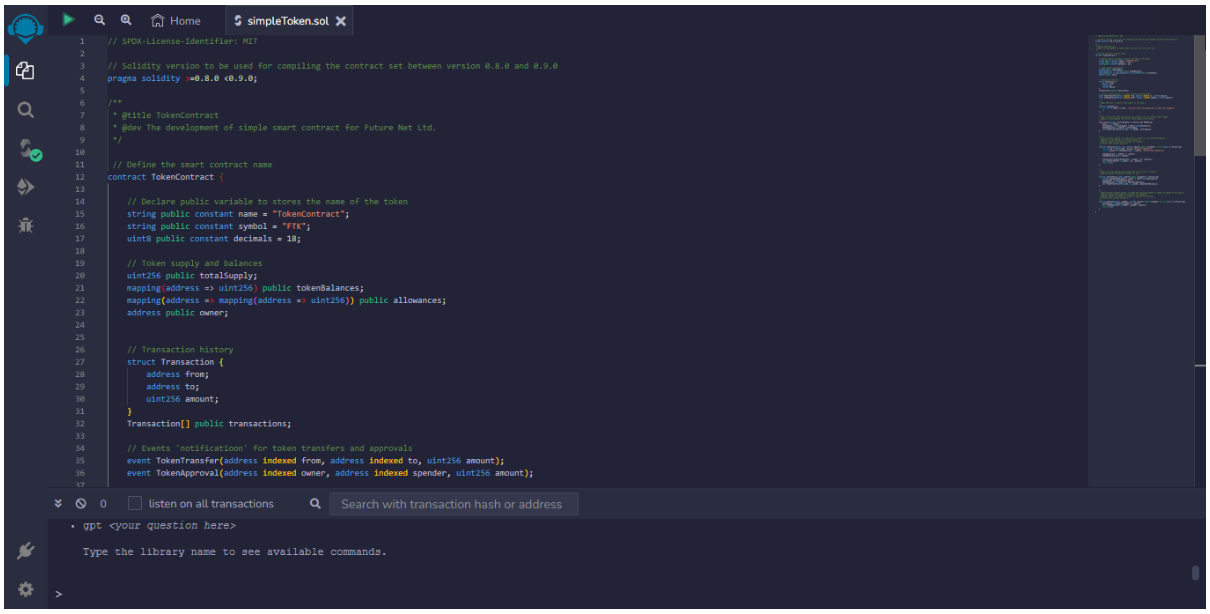


```
Creating Genesis Block...
Genesis Block created.
Printing Block #0...
Hash: a23f1109c8fd43d6a7b3cc299a2c510b0802c626b356954f67377581a53fad85
Timestamp: 2024-05-30 07:16:55
Data: Genesis Block
Previous Hash: 0
Nonce: 0

Do you want to mine a new block? (y/n): y
Enter the consensus mechanism (pow/pos): pow
Enter the Genesis Block hash to create a new block: a23f1109c8fd43d6a7b3cc299a2c510b0802c626b356954f67377581a53fad85
Block #1 created using POW.
Printing Block #1...
Hash: 005c5e90a090bb6408ccb4fdcd57e8894c56ba2be0b51e63a0b67d668c1e1b89
Timestamp: 2024-05-30 07:20:36
Data: Transaction Data
Previous Hash: a23f1109c8fd43d6a7b3cc299a2c510b0802c626b356954f67377581a53fad85
Nonce: 906

Do you want to mine a new block? (y/n): y
Enter the Genesis Block hash to create a new block: a23f1109c8fd43d6a7b3cc299a2c510b0802c626b356954f67377581a53fad85
Block #2 created using POW.
Printing Block #2...
Hash: 007a7d67ccfa8359c9fb1aa739fee20f85d087229ff5a8e8da712d9884e3cbb1
Timestamp: 2024-05-30 07:20:41
Data: Transaction Data
Previous Hash: 005c5e90a090bb6408ccb4fdcd57e8894c56ba2be0b51e63a0b67d668c1e1b89
Nonce: 396

Do you want to mine a new block? (y/n): n
Blockchain is valid.
```



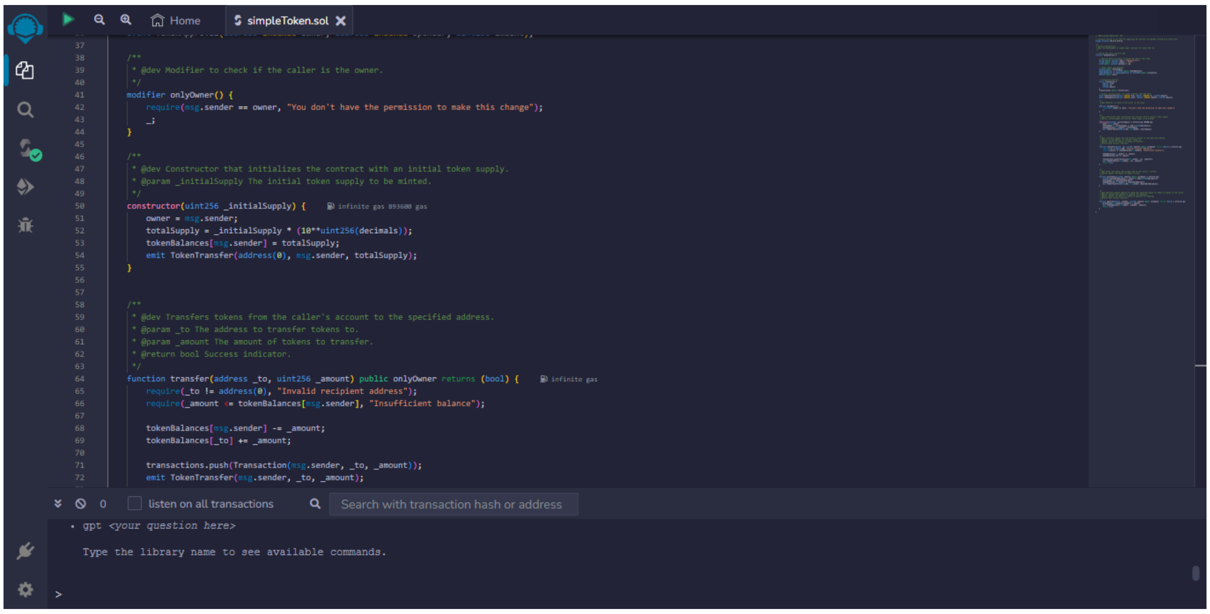
```
Creating Genesis Block...
Genesis Block created.
Printing Block #0...
Hash: d3e191ea4f8f88ef85144df0e068b9f868527b2f9cf5b9b03ce1b28ef3b83640
Timestamp: 2024-05-28 15:53:26
Data: Genesis Block
Previous Hash: 0
Nonce: 0

Do you want to mine a new block? (y/n): 
Invalid input. Please enter 'y' or 'n'.

Do you want to mine a new block? (y/n): y
Enter the consensus mechanism (pow/pos): pos
Enter the Genesis Block hash to create a new block: d3e191ea4f8f88ef85144df0e068b9f868527b2f9cf5b9b03ce1b28ef3b83640
Block #1 created using POS.
Printing Block #1...
Hash: 2973afa47eecc71dfbb736e82381ba62e6e1f0f070bce52105190da285a68d64
Timestamp: 2024-05-28 15:55:02
Data: Transaction Data (Validator: David)
Previous Hash: d3e191ea4f8f88ef85144df0e068b9f868527b2f9cf5b9b03ce1b28ef3b83640
Nonce: 0

Do you want to mine a new block? (y/n): y
Enter the Genesis Block hash to create a new block: d3e191ea4f8f88ef85144df0e068b9f868527b2f9cf5b9b03ce1b28ef3b83640
Block #2 created using POS.
Printing Block #2...
Hash: 8886e6d6c37844f908e9ec5ca54b5819484922db2258a6e83f26dcaa5a736823
Timestamp: 2024-05-28 15:55:10
Data: Transaction Data (Validator: Esther)
Previous Hash: 2973afa47eecc71dfbb736e82381ba62e6e1f0f070bce52105190da285a68d64
Nonce: 0

Do you want to mine a new block? (y/n): n
Blockchain is valid.
```

Smart Contract (Solidity)


Home
simpleToken.sol
listen on all transactions
Search with transaction hash or address

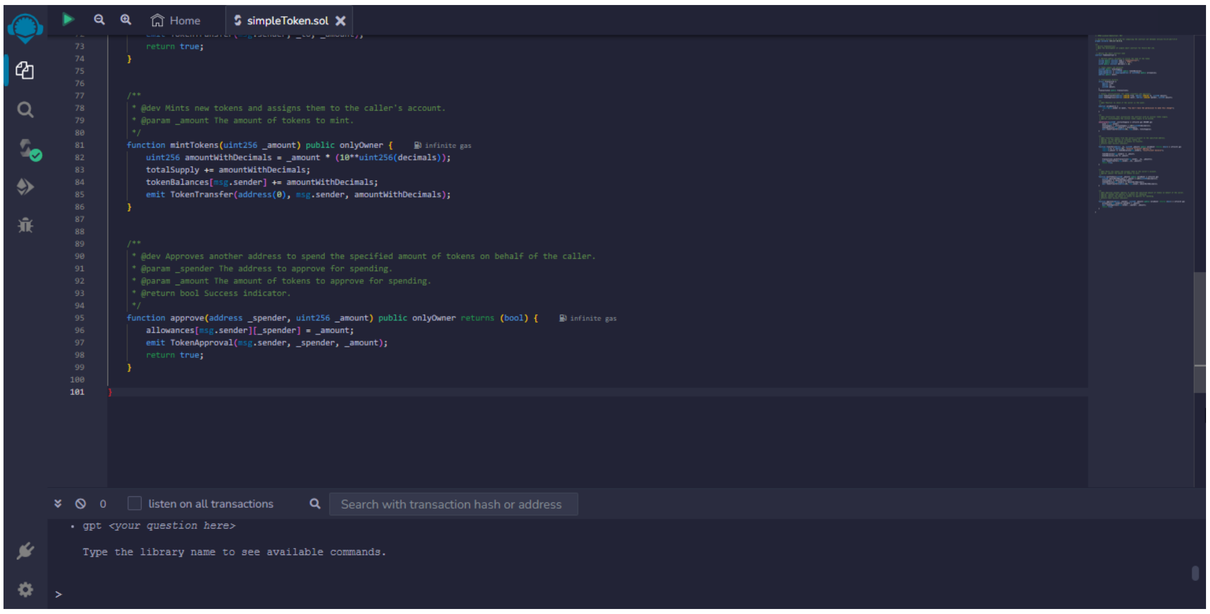
```
// SPDX-License-Identifier: MIT

// Solidity version to be used for compiling the contract set between version 0.8.0 and 0.9.0
pragma solidity >=0.8.0 <0.9.0;

/**
 * @title TokenContract
 * @dev The development of simple smart contract for Future Net Ltd.
 */

 // Define the smart contract name
contract TokenContract {

    // Declare public variable to stores the name of the token
    string public constant name = "TokenContract";
    string public constant symbol = "FTK";
    uint8 public constant decimals = 18;

    // Token supply and balances
    uint256 public totalSupply;
    mapping(address => uint256) public tokenBalances;
    mapping(address => mapping(address => uint256)) public allowances;
    address public owner;


    // Transaction history
    struct Transaction {
        address from;
        address to;
        uint256 amount;
    }
    Transaction[] public transactions;

    // Events 'notificatioon' for token transfers and approvals
    event TokenTransfer(address indexed from, address indexed to, uint256 amount);
    event TokenApproval(address indexed owner, address indexed spender, uint256 amount);
```

```
0
• gpt <your question here>

  Type the library name to see available commands.

>
```


Home
simpleToken.sol
listen on all transactions
Search with transaction hash or address

```

    /**
     * @dev Modifier to check if the caller is the owner.
     */
    modifier onlyOwner() {
        require(msg.sender == owner, "You don't have the permission to make this change");
        _;
    }

    /**
     * @dev Constructor that initializes the contract with an initial token supply.
     * @param _initialSupply The initial token supply to be minted.
     */
    constructor(uint256 _initialSupply) {     infinite gas 893600 gas
        owner = msg.sender;
        totalSupply = _initialSupply * (10**uint256(decimals));
        tokenBalances[msg.sender] = totalSupply;
        emit TokenTransfer(address(0), msg.sender, totalSupply);
    }


    /**
     * @dev Transfers tokens from the caller's account to the specified address.
     * @param _to The address to transfer tokens to.
     * @param _amount The amount of tokens to transfer.
     * @return bool Success indicator.
     */
    function transfer(address _to, uint256 _amount) public onlyOwner returns (bool) {     infinite gas
        require(_to != address(0), "Invalid recipient address");
        require(_amount <= tokenBalances[msg.sender], "Insufficient balance");

        tokenBalances[msg.sender] -= _amount;
        tokenBalances[_to] += _amount;

        transactions.push(Transaction(msg.sender, _to, _amount));
        emit TokenTransfer(msg.sender, _to, _amount);
```

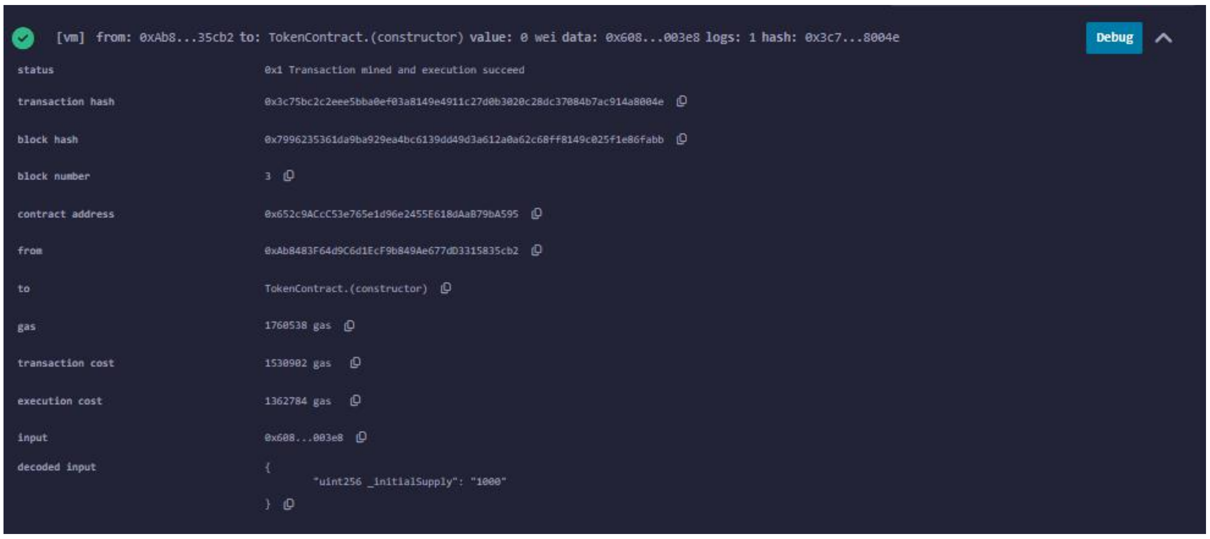
```
0
• gpt <your question here>

  Type the library name to see available commands.

>
```



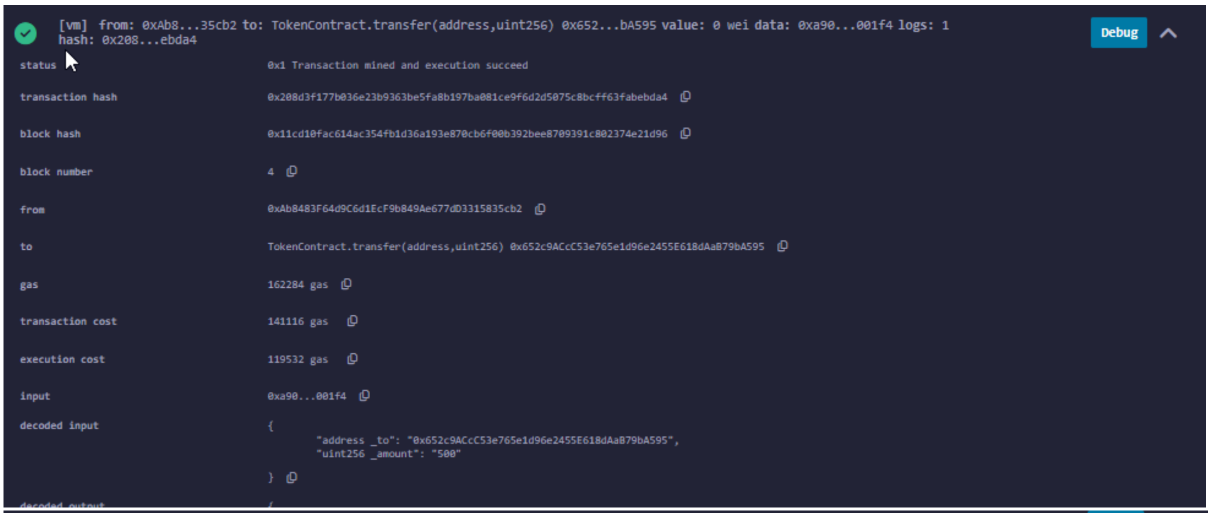
```
        return true;
    }


    /**
     * @dev Mints new tokens and assigns them to the caller's account.
     * @param _amount The amount of tokens to mint.
     */
    function mintTokens(uint256 _amount) public onlyOwner {     infinite gas
        uint256 amountWithDecimals = _amount * (10**uint256(decimals));
        totalSupply += amountWithDecimals;
        tokenBalances[msg.sender] += amountWithDecimals;
        emit TokenTransfer(address(0), msg.sender, amountWithDecimals);
    }


    /**
     * @dev Approves another address to spend the specified amount of tokens on behalf of the caller.
     * @param _spender The address to approve for spending.
     * @param _amount The amount of tokens to approve for spending.
     * @return bool Success indicator.
     */
    function approve(address _spender, uint256 _amount) public onlyOwner returns (bool) {     infinite gas
        allowances[msg.sender][_spender] = _amount;
        emit TokenApproval(msg.sender, _spender, _amount);
        return true;
    }

}
```

0
listen on all transactions
Search with transaction hash or address

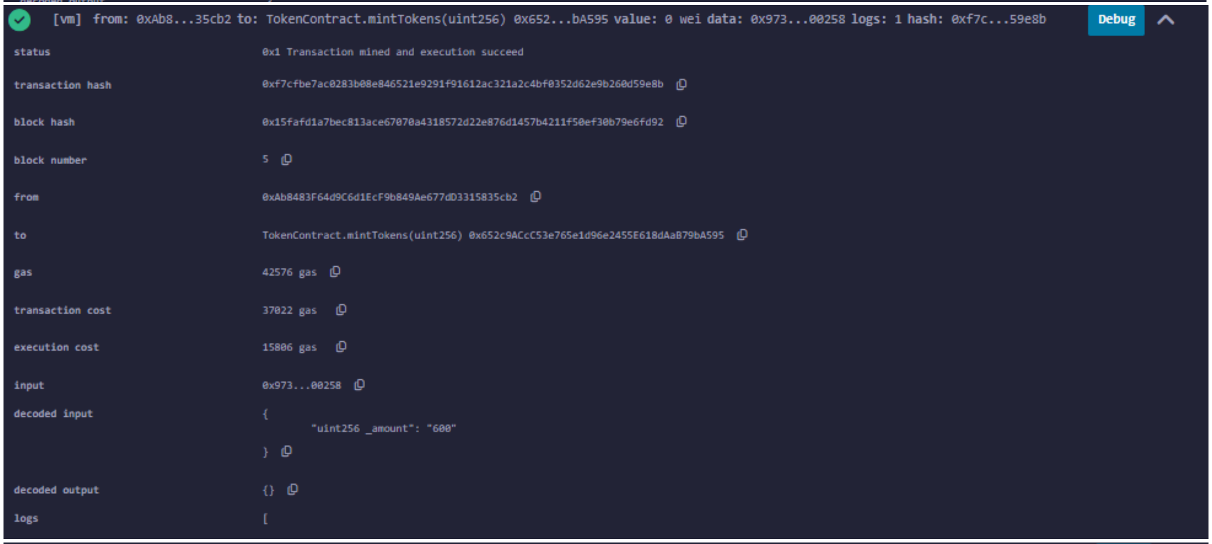
```
• gpt <your question here>

  Type the library name to see available commands.

>
```



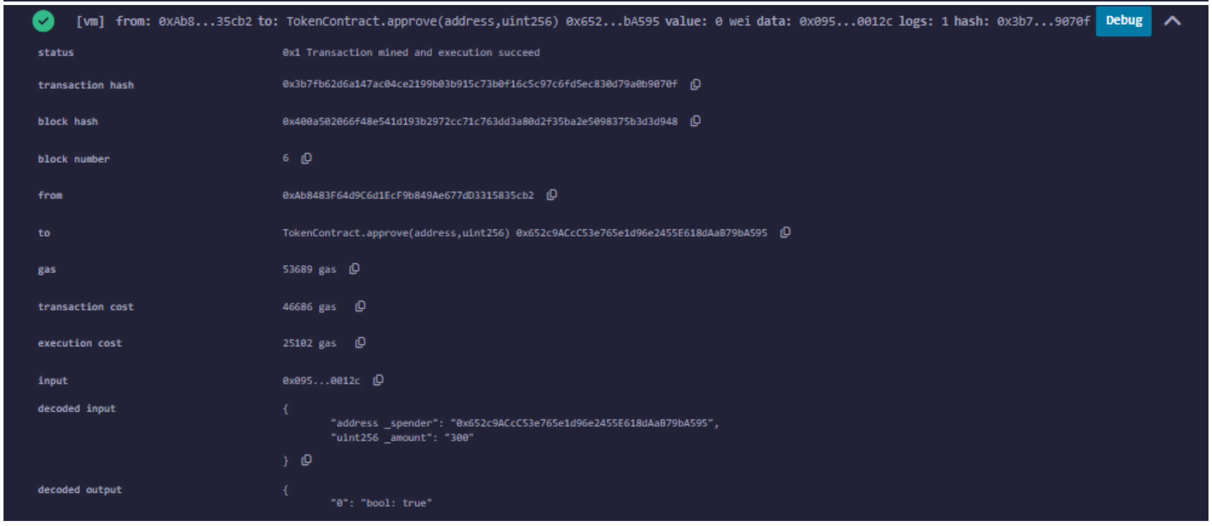
```
[vm] from: 0xAb8...35cb2 to: TokenContract.(constructor) value: 0 wei data: 0x608...003e8 logs: 1 hash: 0x3c7...8004e
status                 0x1 Transaction mined and execution succeed
transaction hash       0x3c75bc2c2eee5bba0ef03a8149e4911c27d0b3020c28dc37084b7ac914a8004e
block hash             0x7996235361da9ba929ea4bc6139dd49d3a612a0a62c68ff8149c025f1e86fabb
block number           3
contract address       0x652c9ACcC53e765e1d96e2455E618dAaB79bA595
from                   0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to                     TokenContract.(constructor)
gas                    1760538 gas
transaction cost       1530902 gas
execution cost         1362784 gas
input                  0x608...003e8
decoded input          {
                           "uint256 _initialSupply": "1000"
                       }
```



```
[vm] from: 0xAb8...35cb2 to: TokenContract.transfer(address,uint256) 0x652...bA595 value: 0 wei data: 0xa90...001f4 logs: 1
hash: 0x208...ebda4
status                 0x1 Transaction mined and execution succeed
transaction hash       0x208d3f177b036e23b9363be5fa8b197ba081ce9f6d2d5075c8bcff63fabebda4
block hash             0x11cd10fac614ac354fb1d36a193e870cb6f00b392bee8709391c802374e21d96
block number           4
from                   0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to                     TokenContract.transfer(address,uint256) 0x652c9ACcC53e765e1d96e2455E618dAaB79bA595
gas                    162284 gas
transaction cost       141116 gas
execution cost         119532 gas
input                  0xa90...001f4
decoded input          {
                              "address _to": "0x652c9ACcC53e765e1d96e2455E618dAaB79bA595",
                              "uint256 _amount": "500"
                       }
decoded output         {
```



```
[vm] from: 0xAb8...35cb2 to: TokenContract.mintTokens(uint256) 0x652...bA595 value: 0 wei data: 0x973...00258 logs: 1 hash: 0xf7c...59e8b
status                 0x1 Transaction mined and execution succeed
transaction hash       0xf7cfbe7ac0283b08e846521e9291f91612ac321a2c4bf0352d62e9b260d59e8b
block hash             0x15fafd1a7bec813ace67070a4318572d22e876d1457b4211f50ef30b79e6fd92
block number           5
from                   0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to                     TokenContract.mintTokens(uint256) 0x652c9ACcC53e765e1d96e2455E618dAaB79bA595
gas                    42576 gas
transaction cost       37022 gas
execution cost         15806 gas
input                  0x973...00258
decoded input          {
                              "uint256 _amount": "600"
                       }
decoded output         {}
logs                   [
```



```
[vm] from: 0xAb8...35cb2 to: TokenContract.approve(address,uint256) 0x652...bA595 value: 0 wei data: 0x095...0012c logs: 1 hash: 0x3b7...9070f
status                 0x1 Transaction mined and execution succeed
transaction hash       0x3b7fb62d6a147ac04ce2199b03b915c73b0f16c5c97c6fd5ec830d79a0b9070f
block hash             0x400a502066f48e541d193b2972cc71c763dd3a80d2f35ba2e5098375b3d3d948
block number           6
from                   0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to                     TokenContract.approve(address,uint256) 0x652c9ACcC53e765e1d96e2455E618dAaB79bA595
gas                    53689 gas
transaction cost       46686 gas
execution cost         25102 gas
input                  0x095...0012c
decoded input          {
                              "address _spender": "0x652c9ACcC53e765e1d96e2455E618dAaB79bA595",
                              "uint256 _amount": "300"
                       }
decoded output         {
                              "0": "bool: true"
```