04.01.2019

Mansur Muaz Ekici
110510030

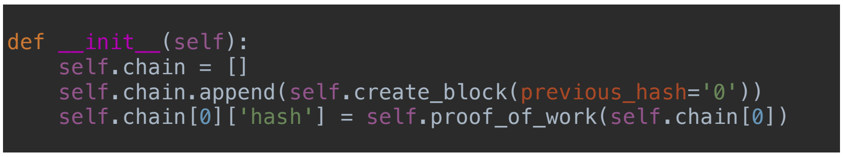
Blockchain & Cryptocurrencies

Project Phase 2

Blockchain phase

In first phase, I’ve built my secure blockchain by coding in Python. It runs on local host at specific port by Flask library. So, you can mine a new block, get the whole chain and verify the validity of the chain by http requests to local host. The role of the node will be storing the random generated data securely which means, if it changes, or deletes, the blockchain will not be valid no longer.

First of all, I’ve created a class called `blockchain` and in the `init` function, I’ve created a new empty chain. Then, I’ve created the genesis block, which is the first block in the blockchain, with 0 previous hash by `create_block` function which explained below. Finally, with the ease of `proof_of_work` function, which explained below, the nonce of the block is calculated so that the whole hash of block starts with 5 leading zeros. This difficulty can be change by the `leading_zeros` and `difficulty` variables in the blockchain class.

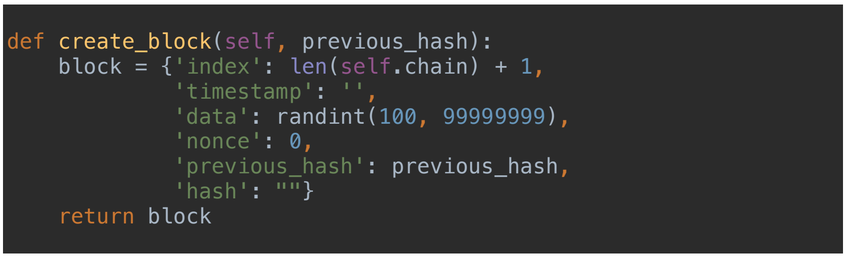


```
def __init__(self):
    self.chain = []
    self.chain.append(self.create_block(previous_hash='0'))
    self.chain[0]['hash'] = self.proof_of_work(self.chain[0])
```

A block, contains the;

- **Block Index**
- **Time Stamp** of mining time
- Random generated **Data**
- **Nonce**
- **Previous Hash**
- **Hash**

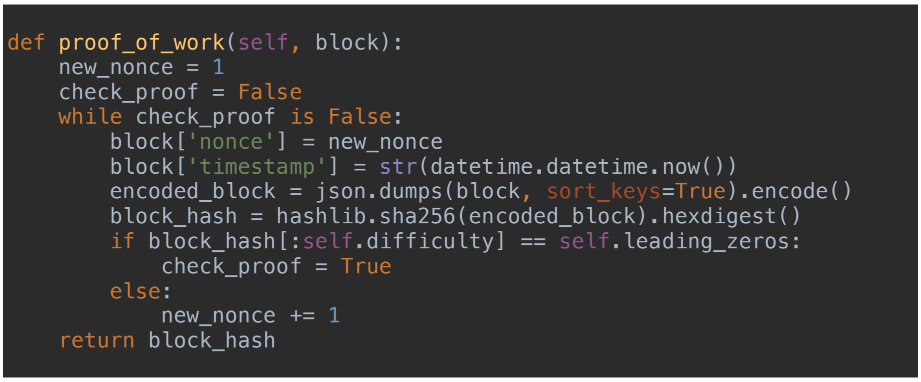
In the `create_block` function, I’ve defined the block with the information above. When this function called, the `index` will be chain size + 1. The `data` will be randomly generated number by ranint() function between 100 and 99999999. This is only for representation of the data, which can be added to a blockchain, like transaction etc. The `timestamp`, `nonce`, and hash will be initially empty, but it will be defined when the nonce calculated by proof_of_work function. The previous_hash, which will get as a parameter by create_block function, will be the hash of the previous block. And this method will return the created block.



```
def create_block(self, previous_hash):
    block = {'index': len(self.chain) + 1,
             'timestamp': '',
             'data': randint(100, 99999999),
             'nonce': 0,
             'previous_hash': previous_hash,
             'hash': ""}
    return block
```

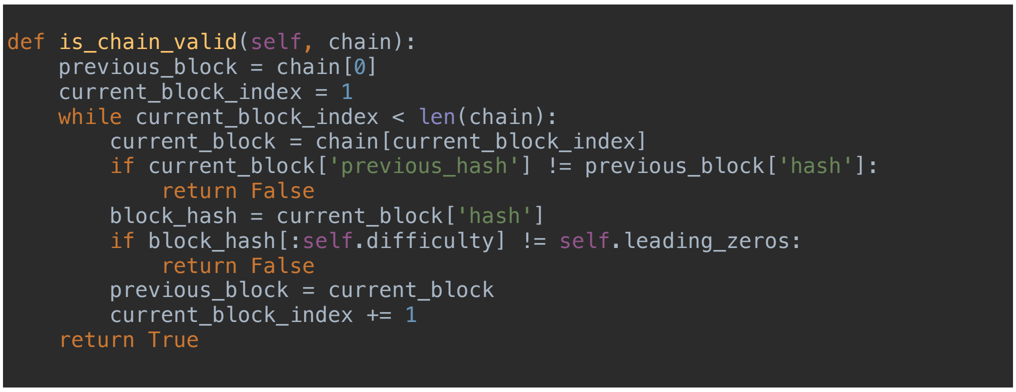
This blockchain basically uses **proof of work** consensus algorithms which encrypted with **sha265**. To be able to mine a block, the nonce of block should be computed as the hash of entire block start with **5 leading zeros**. When it finds the suitable nonce, the block will be added to the chain. This is basically how blockchain technology works.

So, how proof_of_work function works? First of all, new_nonce starts from 1, which increase by 1 in every cycle of while loop until correct nonce is found. And check_proof start as false, and when the correct nonce found, it will be true and the while loop will be break. In every cycle of the while loop, the block(given as parameter) nonce value will be assigned to the new_nonce and the timestamp will be assigned to the current time. After encoding the while block as json, get the sha256 hash of this json value with the ease of hashlib library. Finally, if the block hash's first diffucutly amount of character is all equal to zero, check_proof will be true and while loop will be terminated. Else, the new_nonce will be increased by 1 and all this process starts from beginning. When, it finds the suitable nonce, the found hash will be returned to define as hash of given block.



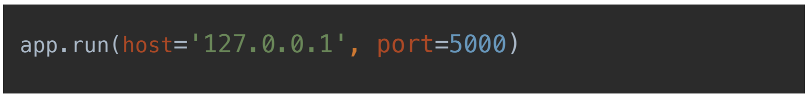
```
def proof_of_work(self, block):
    new_nonce = 1
    check_proof = False
    while check_proof is False:
        block['nonce'] = new_nonce
        block['timestamp'] = str(datetime.datetime.now())
        encoded_block = json.dumps(block, sort_keys=True).encode()
        block_hash = hashlib.sha256(encoded_block).hexdigest()
        if block_hash[:self.difficulty] == self.leading_zeros:
            check_proof = True
        else:
            new_nonce += 1
    return block_hash
```

In the `is_chain_valid` function, it's validating the actual chain's correctness. To do that, it starts from the genesis block which is the first block and move till the end of the chain. Firstly, compares the `current_block`'s `previous_hash` and `previous_block`'s `hash`, and if they are don't match, the method returns false, which means the chain is not valid. Otherwise, keep continue with the comparison of each blocks' hashes which should start with difficulty amount of leading zero. Again, if it don't contain enough leading zeros, the method returns false, otherwise continue with next block's comparisons. At the end, if there is no mistake in the chain, it returns true, which means the chain is valid.



```python
def is_chain_valid(self, chain):
    previous_block = chain[0]
    current_block_index = 1
    while current_block_index < len(chain):
        current_block = chain[current_block_index]
        if current_block['previous_hash'] != previous_block['hash']:
            return False
        block_hash = current_block['hash']
        if block_hash[:self.difficulty] != self.leading_zeros:
            return False
        previous_block = current_block
        current_block_index += 1
    return True
```

To publish my blockchain on Flask, I execute the following code which will run on my local host (`127.0.0.1`) and port `5000`:

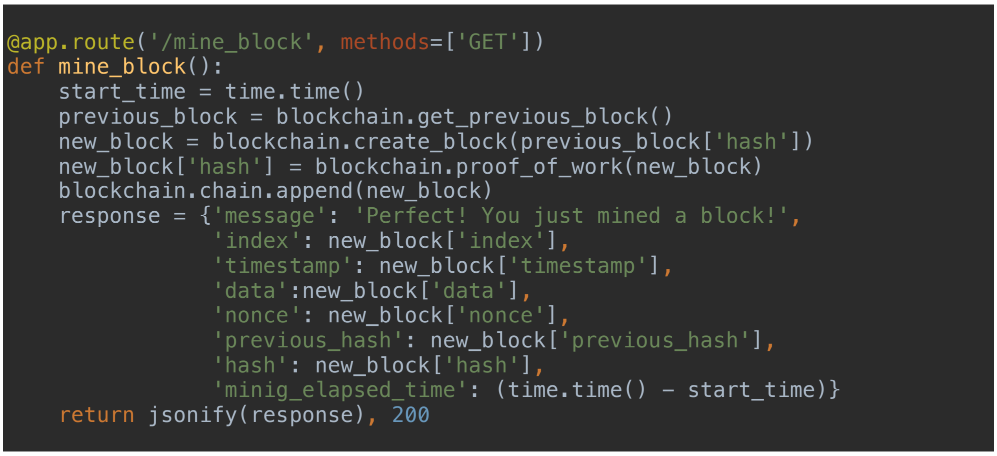


```python
app.run(host='127.0.0.1', port=5000)
```

And, I've defined following routes on this local host:

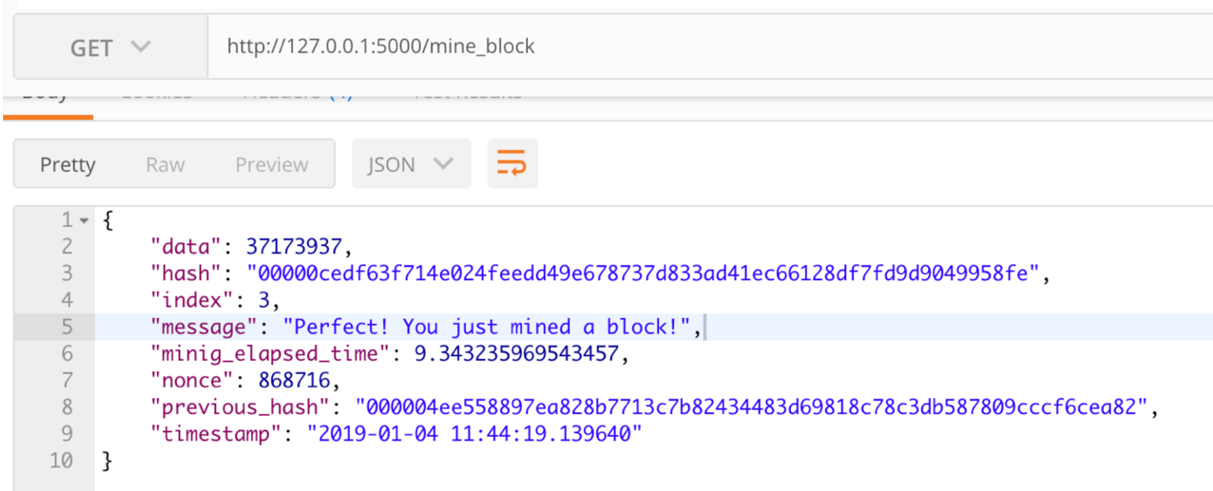
- `localhost/mine_block`

  This route as is evident from its name, basically mine new block to the chain. To do that, firstly, it gets the `previous_block` with the ease of `get_previous_block` function, which returns the last mined block in the chain. Then, it calls the `create_block` function of blockchain class with the `hash` of `previous_block`. Then, it calls the `proof_of_work` function for this newly created block to calculate the suitable `nonce` value. It returns the `hash` of this block which should start with some leading zeros. After the saving this `hash` to the block, it adds this new block to the chain. Finally, it returns a response which include all of the information about newly created block.



```python
@app.route('/mine_block', methods=['GET'])
def mine_block():
    start_time = time.time()
    previous_block = blockchain.get_previous_block()
    new_block = blockchain.create_block(previous_block['hash'])
    new_block['hash'] = blockchain.proof_of_work(new_block)
    blockchain.chain.append(new_block)
    response = {'message': 'Perfect! You just mined a block!',
                'index': new_block['index'],
                'timestamp': new_block['timestamp'],
                'data':new_block['data'],
                'nonce': new_block['nonce'],
                'previous_hash': new_block['previous_hash'],
                'hash': new_block['hash'],
                'minig_elapsed_time': (time.time() - start_time)}
    return jsonify(response), 200
```

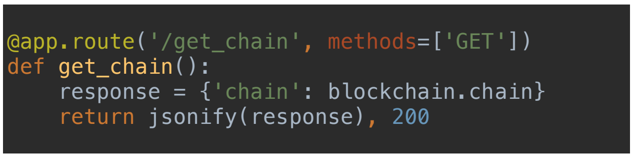
Figure 1 - Postman mine_block request response



```json
{
    "data": 37173937,
    "hash": "00000cedf63f714e024feedd49e678737d833ad41ec66128df7fd9d9049958fe",
    "index": 3,
    "message": "Perfect! You just mined a block!",
    "minig_elapsed_time": 9.343235969543457,
    "nonce": 868716,
    "previous_hash": "000004ee558897ea828b7713c7b82434483d69818c78c3db587809cccf6cea82",
    "timestamp": "2019-01-04 11:44:19.139640"
}
```

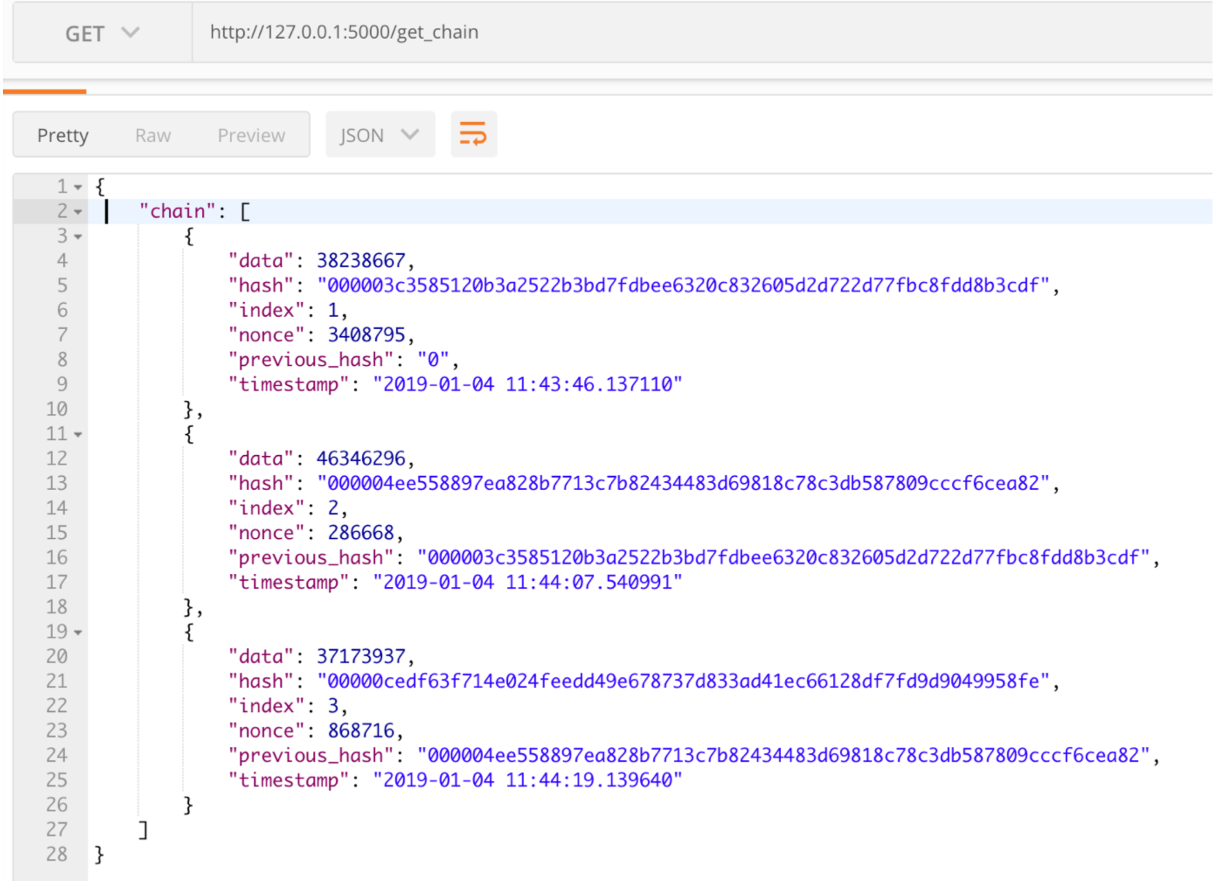
- `localhost/get_chain`

This route, just basically returns the whole chain as response.



```python
@app.route('/get_chain', methods=['GET'])
def get_chain():
    response = {'chain': blockchain.chain}
    return jsonify(response), 200
```

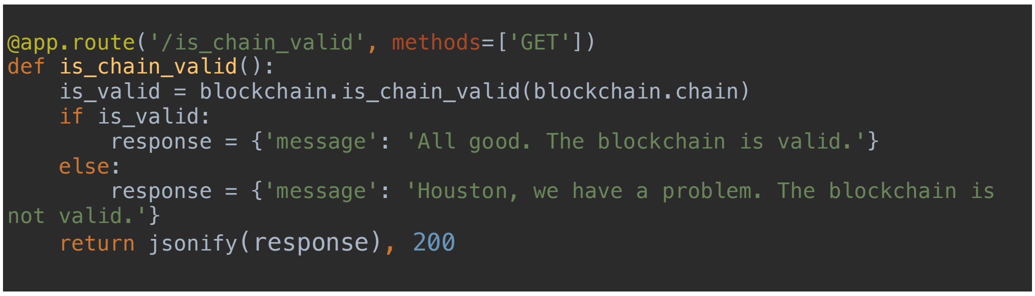
Figure 2 - Postman get_chain request response



```
{
    "chain": [
        {
            "data": 38238667,
            "hash": "000003c3585120b3a2522b3bd7fdbee6320c832605d2d722d77fbc8fdd8b3cdf",
            "index": 1,
            "nonce": 3408795,
            "previous_hash": "0",
            "timestamp": "2019-01-04 11:43:46.137110"
        },
        {
            "data": 46346296,
            "hash": "000004ee558897ea828b7713c7b82434483d69818c78c3db587809cccf6cea82",
            "index": 2,
            "nonce": 286668,
            "previous_hash": "000003c3585120b3a2522b3bd7fdbee6320c832605d2d722d77fbc8fdd8b3cdf",
            "timestamp": "2019-01-04 11:44:07.540991"
        },
        {
            "data": 37173937,
            "hash": "00000cedf63f714e024feedd49e678737d833ad41ec66128df7fd9d9049958fe",
            "index": 3,
            "nonce": 868716,
            "previous_hash": "000004ee558897ea828b7713c7b82434483d69818c78c3db587809cccf6cea82",
            "timestamp": "2019-01-04 11:44:19.139640"
        }
    ]
}
```

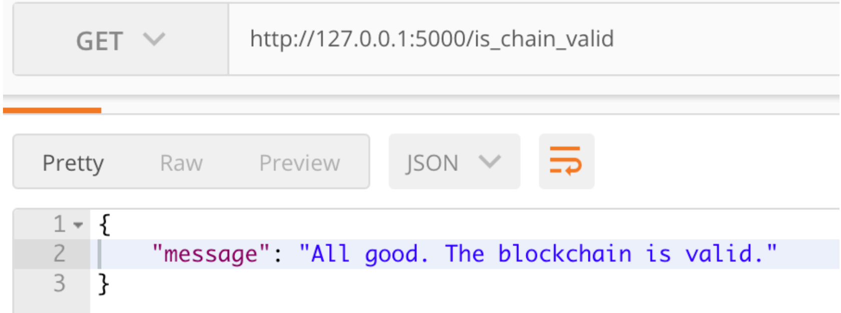
- `Localhost/is_chain_valid`

This route, calls the `is_chain_valid` function, which explained above, of blockchain class. And returns a response according to the validity of blockchain.



```
@app.route('/is_chain_valid', methods=['GET'])
def is_chain_valid():
    is_valid = blockchain.is_chain_valid(blockchain.chain)
    if is_valid:
        response = {'message': 'All good. The blockchain is valid.'}
    else:
        response = {'message': 'Houston, we have a problem. The blockchain is 
not valid.'}
    return jsonify(response), 200
```

Figure 3 - Postman is_chain_valid request response



```
{
    "message": "All good. The blockchain is valid."
}
```

So, you can easily use this blockchain like mining new block, getting whole chain and validating the correctness of the blockchain, with going this routes. To do that, firstly you just need to execute the python code, which will start to publishing the blockchain in the localhost, then you can use your browser or HTTP request, API development applications like Postman more easily.

Cryptocurrency phase

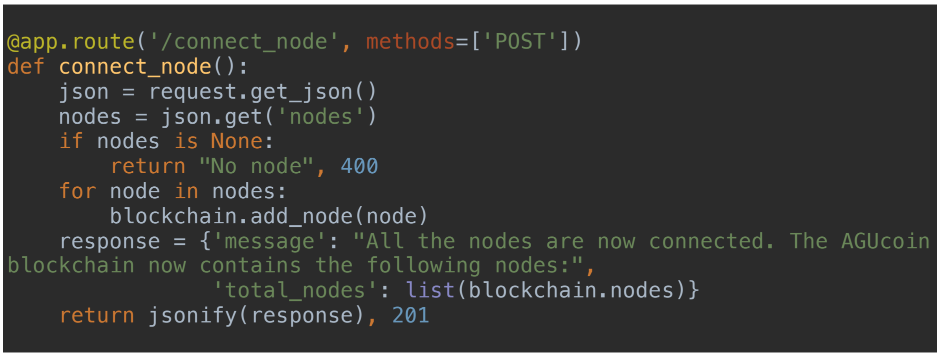
The second phase of the project, I've used this extended version of the blockchain which mentioned above, to build a distributed **cryptocurrency** which called **AGUCoin**. But this time, the **transaction** information between nodes will be included in the blocks, instead of random generated data. In this phase, I just explained only the differences between blockchain phase.

The AGUCoin server will run on my local host with using Flask library of Python, but this time, there will be multiple nodes which execute same software from **different ports**. So, if I would have chance to implement this in to the global network, it will be real distributed and decentralized blockchain network, but for now, I'm just imitating this by different ports. The role of the nodes in second phase is storing the transaction information even more secure because it is distributed network. So, If the **%50+1** of the computation power are consist of trusted nodes, the transaction information will be secure, cannot be added, changed or deleted.

At first, all nodes need to connect to each other, and then, they can make transactions between each other. To do that, I created a route called `connect_node` which should be send as a POST request with the other nodes' addresses in its body. For example, HTTP request for the host node (which runs on `port 5001`), to connect other nodes (which runs on `port 5002 and 5003`) should be like following;

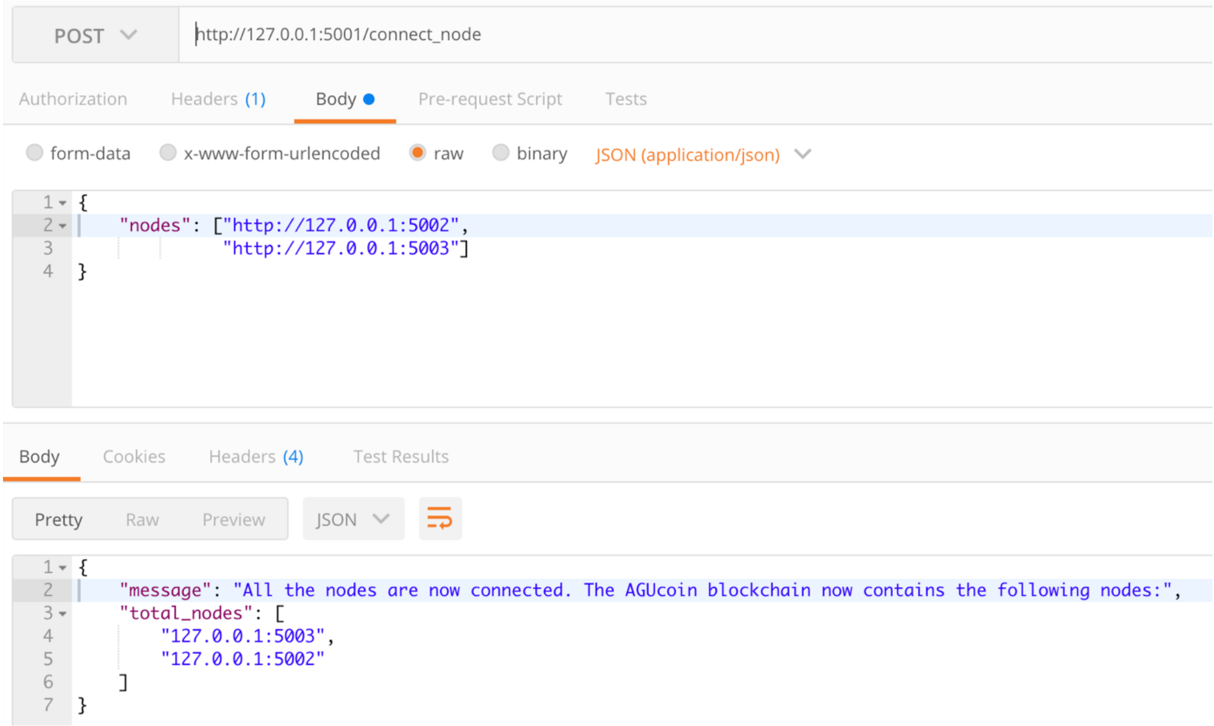
```
POST /connect_node HTTP/1.1
Host: 127.0.0.1:5001
Content-Type: application/json
{
"nodes": ["http://127.0.0.1:5002",
          "http://127.0.0.1:5003"]
}
```

This route will take the given node addresses and puts them to the nodes list of the blockchain of the host node by the ease of `add_node` function. So, when all nodes make this connection, they will be aware of each other.



```python
@app.route('/connect_node', methods=['POST'])
def connect_node():
    json = request.get_json()
    nodes = json.get('nodes')
    if nodes is None:
        return "No node", 400
    for node in nodes:
        blockchain.add_node(node)
    response = {'message': "All the nodes are now connected. The AGUcoin
blockchain now contains the following nodes:",
                'total_nodes': list(blockchain.nodes)}
    return jsonify(response), 201
```

Figure 4 - Postman connect_node request response

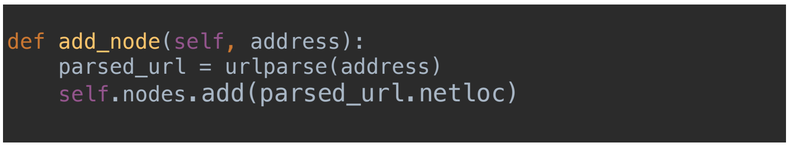


http://127.0.0.1:5001/connect_node

```json
{
    "nodes": ["http://127.0.0.1:5002",
              "http://127.0.0.1:5003"]
}
```

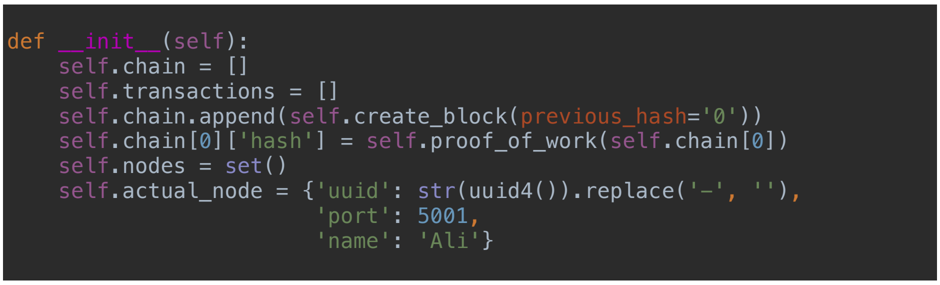
```json
{
    "message": "All the nodes are now connected. The AGUcoin blockchain now contains the following nodes:",
    "total_nodes": [
        "127.0.0.1:5003",
        "127.0.0.1:5002"
    ]
}
```

The `add_node` function just parses the given address's port number and adds it to the node list of the host node.



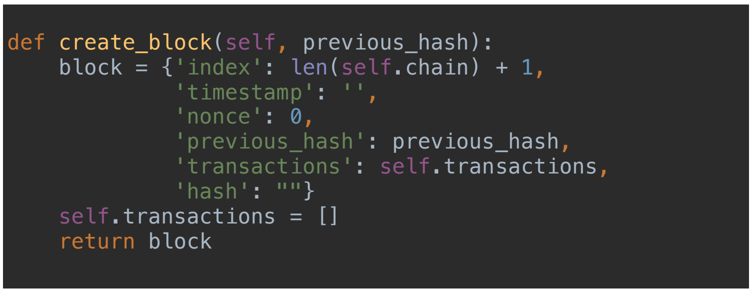
```python
def add_node(self, address):
    parsed_url = urlparse(address)
    self.nodes.add(parsed_url.netloc)
```

In the init function of the blockchain, I added also the transactions array and nodes set which will declared as empty array and set. Also, I just saved the actual_node information. The remaining part as same as the blockchain phase. It just create the genesis block.



```
def __init__(self):
    self.chain = []
    self.transactions = []
    self.chain.append(self.create_block(previous_hash='0'))
    self.chain[0]['hash'] = self.proof_of_work(self.chain[0])
    self.nodes = set()
    self.actual_node = {'uuid': str(uuid4()).replace('-', ''),
                        'port': 5001,
                        'name': 'Ali'}
```

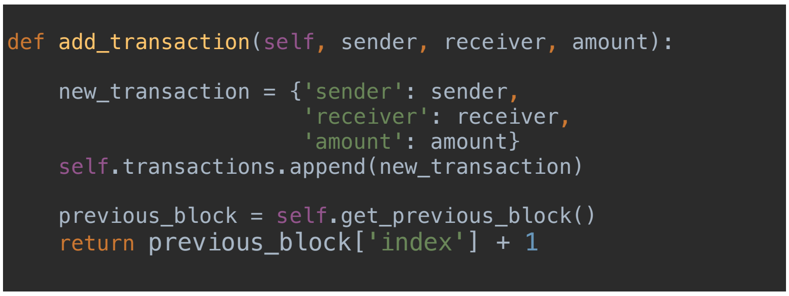
The create_block function, additionally adds the transactions array to the block. And after adding the current transactions in the array to the block, it clear the transactions array.



```
def create_block(self, previous_hash):
    block = {'index': len(self.chain) + 1,
             'timestamp': '',
             'nonce': 0,
             'previous_hash': previous_hash,
             'transactions': self.transactions,
             'hash': ""}
    self.transactions = []
    return block
```

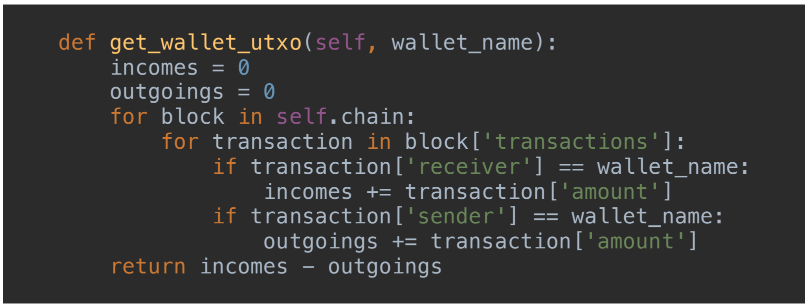
The proof_of_work and is_chain_valid functions work exactly same with the blockchain phase. That's why, I didn't explain them again, codes are exactly same.

To be able to make transaction, first some transaction should be added to the transactions array. So, the add_transation function should be called. This method will get the sender, receiver and transaction amount as parameter and saved them to the transactions array. And it returns the block index which the transactions will added on.



```
def add_transaction(self, sender, receiver, amount):

    new_transaction = {'sender': sender,
                       'receiver': receiver,
                       'amount': amount}
    self.transactions.append(new_transaction)

    previous_block = self.get_previous_block()
    return previous_block['index'] + 1
```

Since AGUCoin is an **UTXO** based cryptocurrency, it should keep track of the incomes and outcomes of the its users. To do that, it compute the wallet's UTXO(Unspent Transaction Output) by traversing all the chain and subtracting all the `outgoings` from the all `incomes` with the ease of `get_wallet_utxo` function.



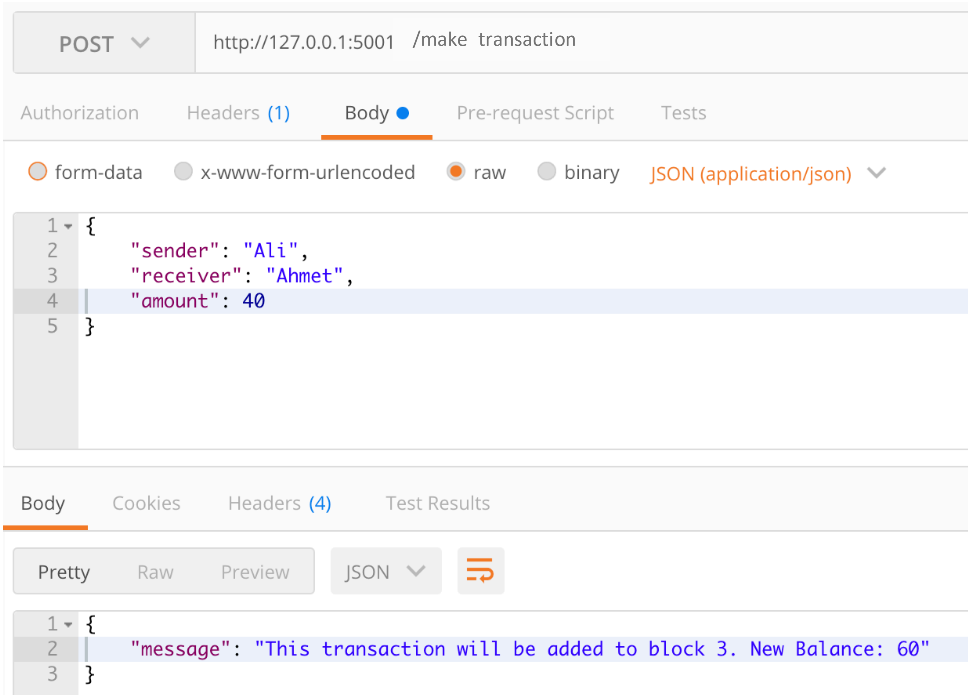
```python
def get_wallet_utxo(self, wallet_name):
    incomes = 0
    outgoings = 0
    for block in self.chain:
        for transaction in block['transactions']:
            if transaction['receiver'] == wallet_name:
                incomes += transaction['amount']
            if transaction['sender'] == wallet_name:
                outgoings += transaction['amount']
    return incomes - outgoings
```

The `make_transaction` route, which is a POST request with transaction data (`sender`, `receiver` and `amount`) in its body, uses the `add_transaction` and `get_wallet_utxo` functions, which explained above.

```
POST /make_transaction HTTP/1.1
Host: 127.0.0.1:5001
Content-Type: application/json
{
        "sender": "Ali",
        "receiver": "Ahmet",
        "amount": 40
}
```

After getting the transaction data, it compute the UTXO of the `sender` by the ease of `get_wallet_utxo` function and save it as `balance`. After that, it checks the `transactions` array of the blockchain, which will include the transactions that are not on a mined block. If there is another transaction which made by same sender, it decrease that amount from the `balance`. Finally it compares the `balance` with the `amount` which want to be sent. If the `balance` is bigger than or equal to the transaction `amount`, the transaction is added to the `transactions` array and `new_balance` of `sender` returns as response. Otherwise, the transaction don't add to the `transactions` array and returns an error with a response which says, the sender don't have enough coin in his/her wallet.

Figure 5 - Postman make_transaction request success response



POST http://127.0.0.1:5001 /make transaction

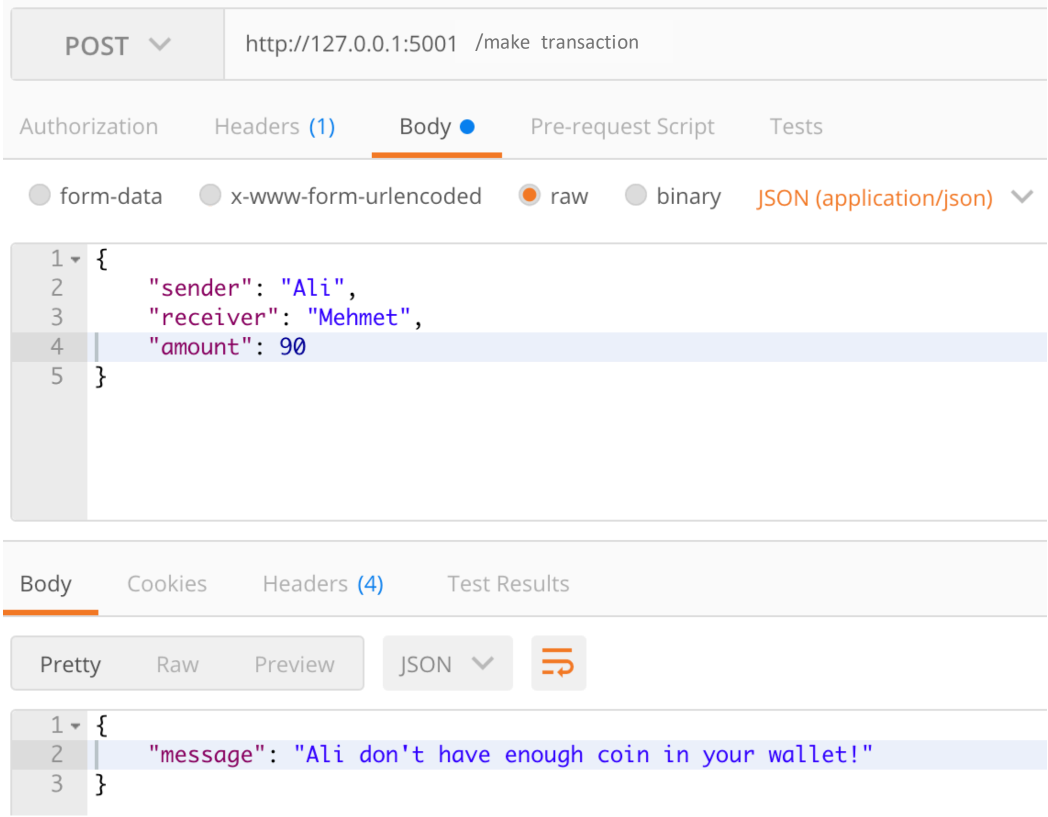
Body: raw, JSON (application/json)

```
{
    "sender": "Ali",
    "receiver": "Ahmet",
    "amount": 40
}
```

Body: Pretty, JSON

```
{
    "message": "This transaction will be added to block 3. New Balance: 60"
}
```

Figure 6 - Postman make_transaction request failure response



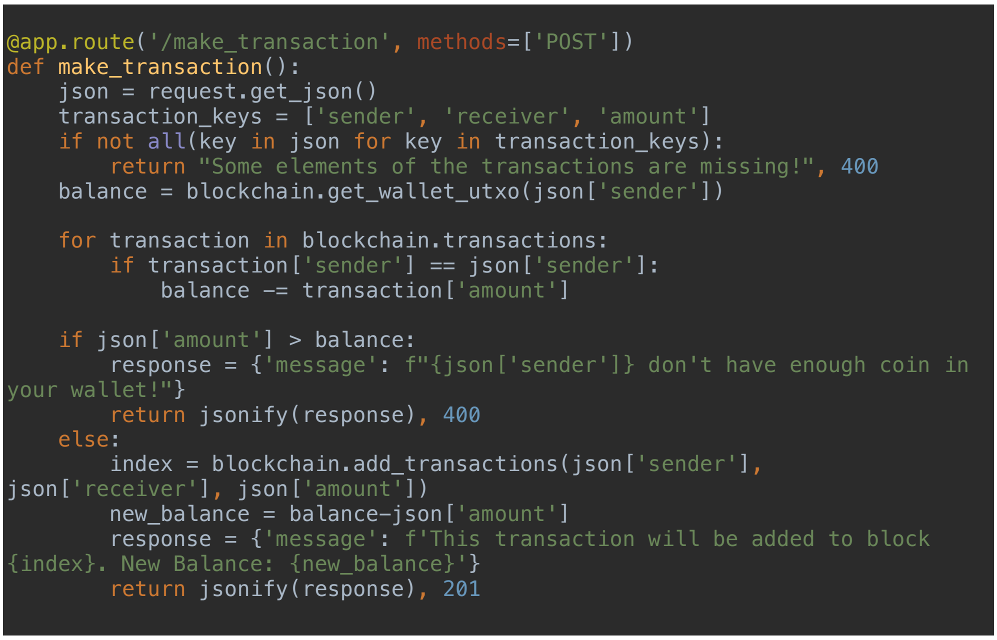
POST http://127.0.0.1:5001 /make transaction

Body: raw, JSON (application/json)

```
{
    "sender": "Ali",
    "receiver": "Mehmet",
    "amount": 90
}
```

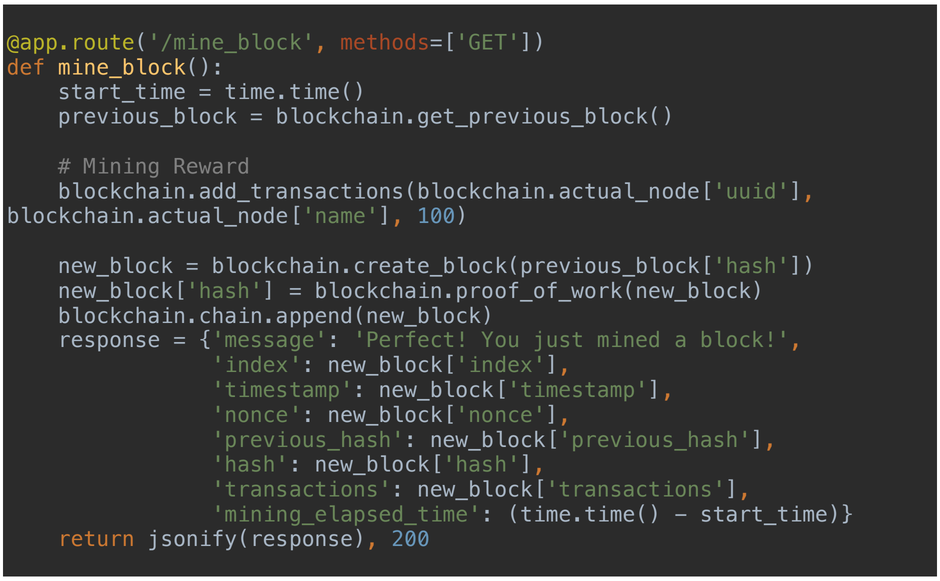
Body: Pretty, JSON

```
{
    "message": "Ali don't have enough coin in your wallet!"
}
```



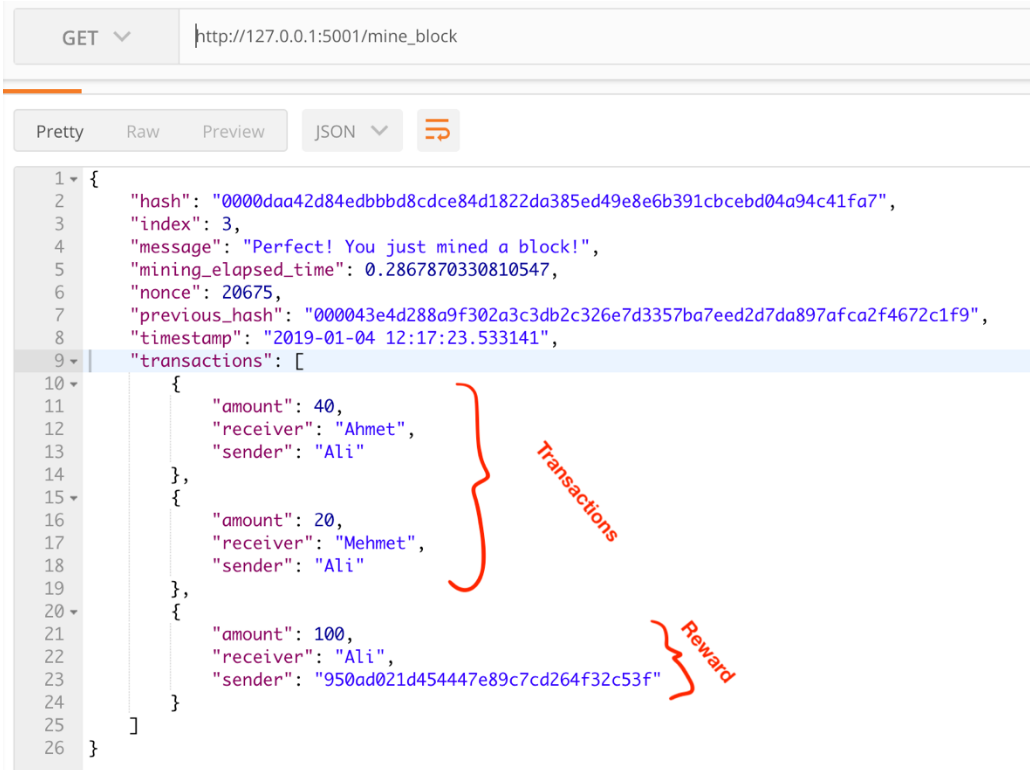
```
@app.route('/make_transaction', methods=['POST'])
def make_transaction():
    json = request.get_json()
    transaction_keys = ['sender', 'receiver', 'amount']
    if not all(key in json for key in transaction_keys):
        return "Some elements of the transactions are missing!", 400
    balance = blockchain.get_wallet_utxo(json['sender'])

    for transaction in blockchain.transactions:
        if transaction['sender'] == json['sender']:
            balance -= transaction['amount']

    if json['amount'] > balance:
        response = {'message': f"{json['sender']} don't have enough coin in 
your wallet!"}
        return jsonify(response), 400
    else:
        index = blockchain.add_transactions(json['sender'], 
json['receiver'], json['amount'])
        new_balance = balance-json['amount']
        response = {'message': f'This transaction will be added to block 
{index}. New Balance: {new_balance}'}
        return jsonify(response), 201
```

But to be able to make transaction, first they need to have some AGUCoin. To have some, they need to gain **reward coins** by mining blocks. That's why I modified the `mine_block` route to give 100 AGUCoin to the miner. To do that, before the `create_block` function called, it calls the `add_transaction` function with the parameters of miner node's `id` as `sender`, miner node's `name` as `receiver` and `100` as `amount`. So, when a new block mined, the node who mined this new block gain the mining reward.



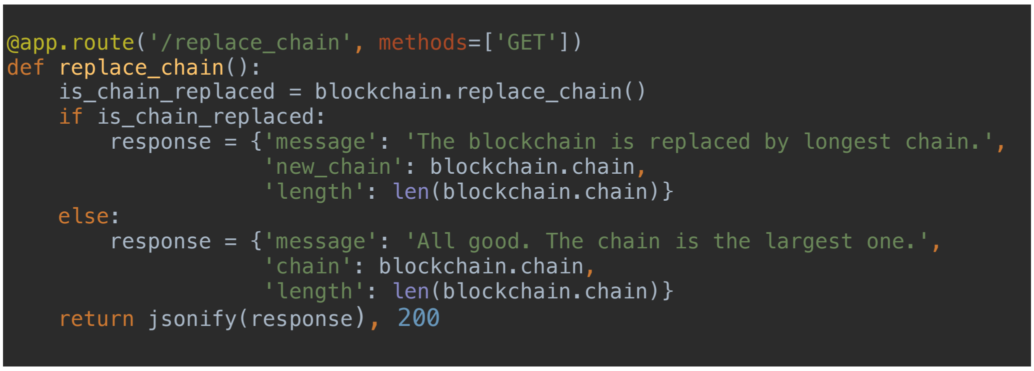
```
@app.route('/mine_block', methods=['GET'])
def mine_block():
    start_time = time.time()
    previous_block = blockchain.get_previous_block()

    # Mining Reward
    blockchain.add_transactions(blockchain.actual_node['uuid'], 
blockchain.actual_node['name'], 100)

    new_block = blockchain.create_block(previous_block['hash'])
    new_block['hash'] = blockchain.proof_of_work(new_block)
    blockchain.chain.append(new_block)
    response = {'message': 'Perfect! You just mined a block!',
                'index': new_block['index'],
                'timestamp': new_block['timestamp'],
                'nonce': new_block['nonce'],
                'previous_hash': new_block['previous_hash'],
                'hash': new_block['hash'],
                'transactions': new_block['transactions'],
                'mining_elapsed_time': (time.time() - start_time)}
    return jsonify(response), 200
```

Figure 7 - Postman mine_block request response



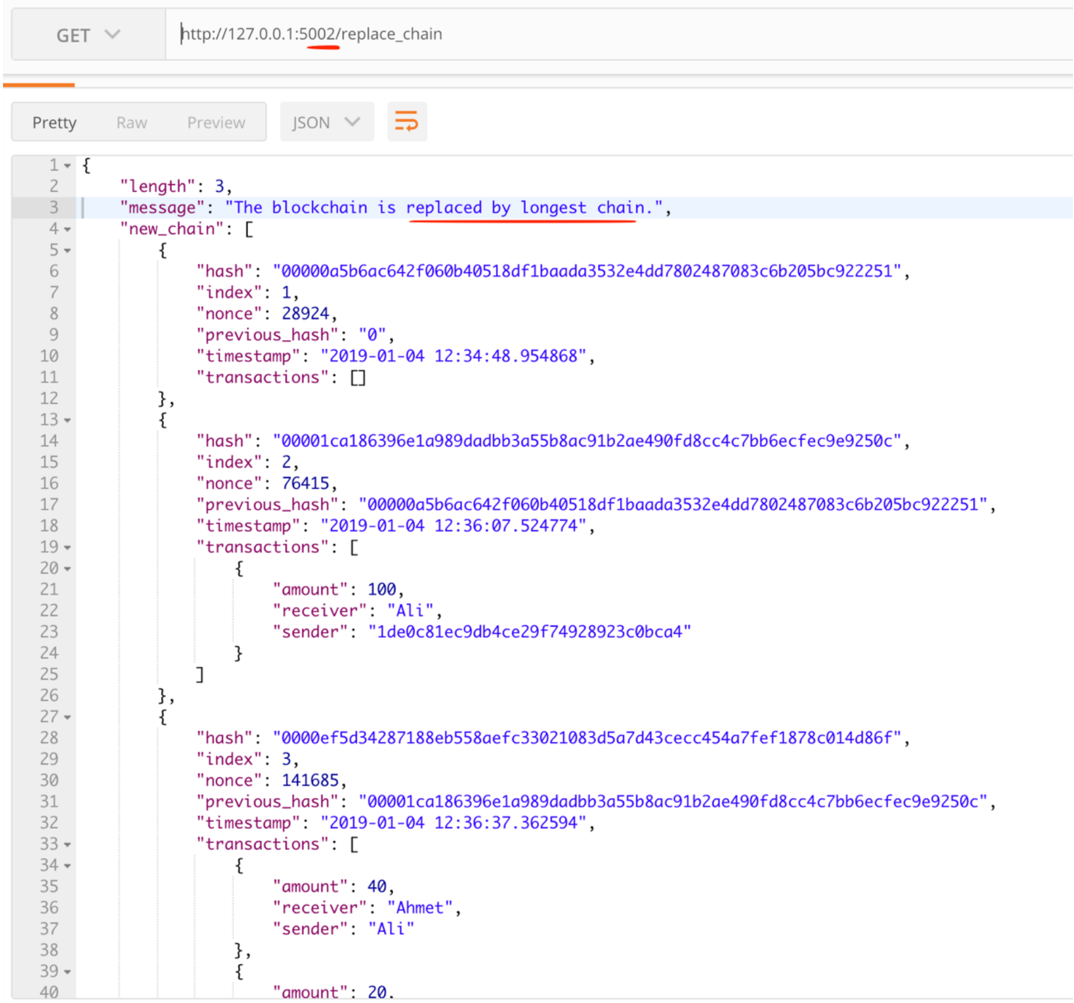
```
{
    "hash": "0000daa42d84edbbbd8cdce84d1822da385ed49e8e6b391cbcebd04a94c41fa7",
    "index": 3,
    "message": "Perfect! You just mined a block!",
    "mining_elapsed_time": 0.2867870330810547,
    "nonce": 20675,
    "previous_hash": "000043e4d288a9f302a3c3db2c326e7d3357ba7eed2d7da897afca2f4672c1f9",
    "timestamp": "2019-01-04 12:17:23.533141",
    "transactions": [
        {
            "amount": 40,
            "receiver": "Ahmet",
            "sender": "Ali"
        },
        {
            "amount": 20,
            "receiver": "Mehmet",
            "sender": "Ali"
        },
        {
            "amount": 100,
            "receiver": "Ali",
            "sender": "950ad021d454447e89c7cd264f32c53f"
        }
    ]
}
```

Also, the due to the AGUCoin server **distributed** among the different nodes, there can be some conflicts. I've solve this conflicts by using the **longest chain** method. So, even a chain forked by 2 different nodes, after a while, the longest chain should be replaced by the other chains in the server. To support this idea, `replace_chain` route, calls the `replace_chain` function. According to its return value, if the chain is replace or not, it returns a response which shows the last situation.



```
@app.route('/replace_chain', methods=['GET'])
def replace_chain():
    is_chain_replaced = blockchain.replace_chain()
    if is_chain_replaced:
        response = {'message': 'The blockchain is replaced by longest chain.',
                    'new_chain': blockchain.chain,
                    'length': len(blockchain.chain)}
    else:
        response = {'message': 'All good. The chain is the largest one.',
                    'chain': blockchain.chain,
                    'length': len(blockchain.chain)}
    return jsonify(response), 200
```

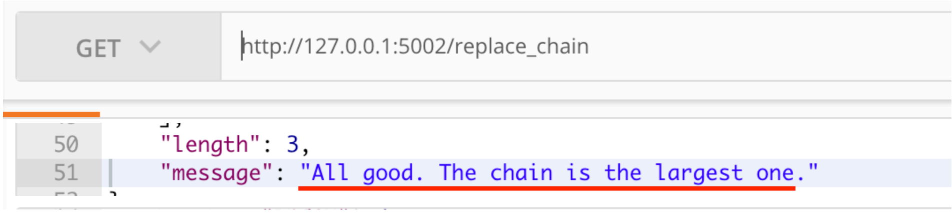
Figure 8 - Postman replace_chain request change response



http://127.0.0.1:5002/replace_chain

```
{
    "length": 3,
    "message": "The blockchain is replaced by longest chain.",
    "new_chain": [
        {
            "hash": "00000a5b6ac642f060b40518df1baada3532e4dd7802487083c6b205bc922251",
            "index": 1,
            "nonce": 28924,
            "previous_hash": "0",
            "timestamp": "2019-01-04 12:34:48.954868",
            "transactions": []
        },
        {
            "hash": "00001ca186396e1a989dadbb3a55b8ac91b2ae490fd8cc4c7bb6ecfec9e9250c",
            "index": 2,
            "nonce": 76415,
            "previous_hash": "00000a5b6ac642f060b40518df1baada3532e4dd7802487083c6b205bc922251",
            "timestamp": "2019-01-04 12:36:07.524774",
            "transactions": [
                {
                    "amount": 100,
                    "receiver": "Ali",
                    "sender": "1de0c81ec9db4ce29f74928923c0bca4"
                }
            ]
        },
        {
            "hash": "0000ef5d34287188eb558aefc33021083d5a7d43cecc454a7fef1878c014d86f",
            "index": 3,
            "nonce": 141685,
            "previous_hash": "00001ca186396e1a989dadbb3a55b8ac91b2ae490fd8cc4c7bb6ecfec9e9250c",
            "timestamp": "2019-01-04 12:36:37.362594",
            "transactions": [
                {
                    "amount": 40,
                    "receiver": "Ahmet",
                    "sender": "Ali"
                },
                {
                    "amount": 20,
```

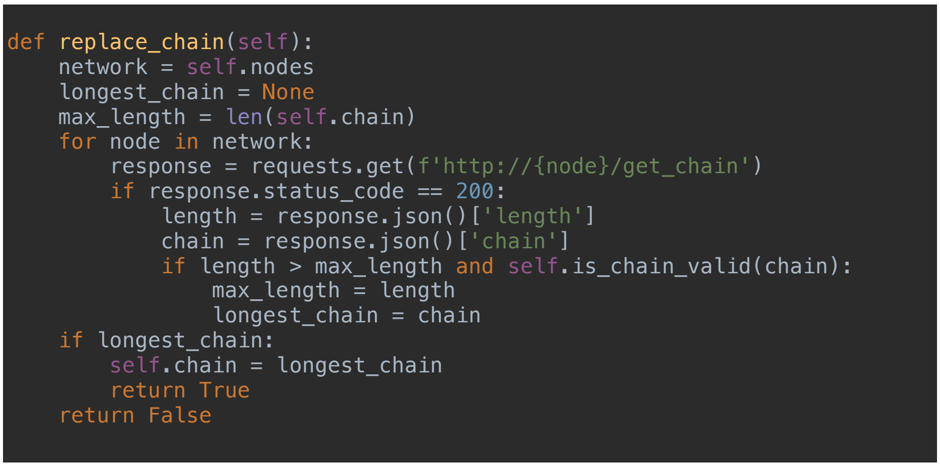
Figure 9 - Postman replace_chain request don't change response



http://127.0.0.1:5002/replace_chain

```
    "length": 3,
    "message": "All good. The chain is the largest one."
```

The replace_chain function makes the HTTP requests to get_chain route, which is same with the blockchain phase, for the whole nodes' addresses in the network, which it connected, to get their whole chains. And it compares the length of the chains of the all nodes and find the longest one. Also it checks the validity of the chains of all nodes by is_chain_valid function. Finally, if the current node's chain is the longest, the function just returns false, which means, there is no change. But otherwise, it changes the current chain with the longest_chain and returns true which means, a replacement happened.



```
def replace_chain(self):
    network = self.nodes
    longest_chain = None
    max_length = len(self.chain)
    for node in network:
        response = requests.get(f'http://{node}/get_chain')
        if response.status_code == 200:
            length = response.json()['length']
            chain = response.json()['chain']
            if length > max_length and self.is_chain_valid(chain):
                max_length = length
                longest_chain = chain
    if longest_chain:
        self.chain = longest_chain
        return True
    return False
```

So, you can easily use this cryptocurrency with just executing the python code, which will start to publishing a node in the localhost. To make it distributed, you need to publish a couple of nodes from different ports. Then, you will be able to make some transactions between these nodes using blockchain technology. Finally, you can use your browser or HTTP request, API development applications like Postman more easily to test AGUCoin.

Probably, this project will not contribute the blockchain studies, but it's really helpful me to understand the basics of blockchain technology. After this project, maybe I can contribute the studies with the perspective which I've gained from this project.

Additionally, you can find the blockchain and the an example node of cryptocurrency phases codes in attached.

Mansur Muaz EKICI

110510030