Build a network to do transactions on blockchain Wallet

The Assignment basically concentrates on implementing a public blockchain from scratch and by building a simple application to leverage it. It includes creation of endpoints for different functions of the blockchain, such as adding a transaction, using the Flask micro-framework, and visual studio code is used to provide packages and environment. It also includes how to build a simple user interface that interacts with the blockchain.
The present goal is to make a website that allows users to share information and the uniqueness is the content is stored in blockchain so it is immutable.

A blockchain, originally block chain, is a growing list of records, called *blocks*, which are linked using cryptography. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data. A distributed data storage consisting of containers (blocks) which are connected. By design, a blockchain is resistant to modification of the data. It is "an open, distributed ledger that can record transactions between two parties efficiently and in a verifiable and permanent way". For use as a distributed ledger, a blockchain is typically managed by a peer-to-peer network collectively adhering to a protocol for inter-node communication and validating new blocks. Once recorded, the data in any given block cannot be altered retroactively without alteration of all subsequent blocks, which requires consensus of the network majority.

AIM:

To build a simple web application that allows users to share information on a blockchain having properties

- History Immutability
- Data Persistence
- No single point of failure

A transaction by a user has the following three data points:

- Sender
- Recipients
- Timestamp

SYSTEM REQUIREMENTS:

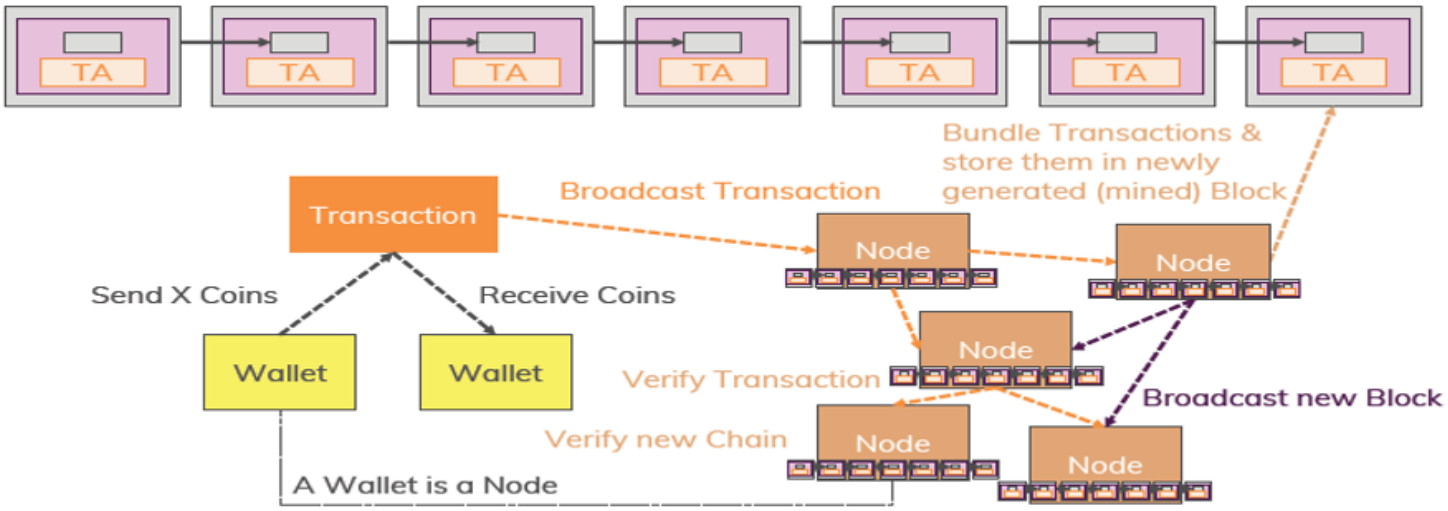
- Python
- Flask Microframe
- JSON format to store data
- Visual Studio Code

ANALYSIS

The project consists of code in different phases:

- Storing data into blocks
- Making blocks immutable
- Chaining blocks
- Mining new blocks
- Share Data, Resolve conflicts
- Wallets
- Analyze and Verify Chain

Nodal structure:
It helps knowing the connections between nodes or block parts



Hashing Structure:

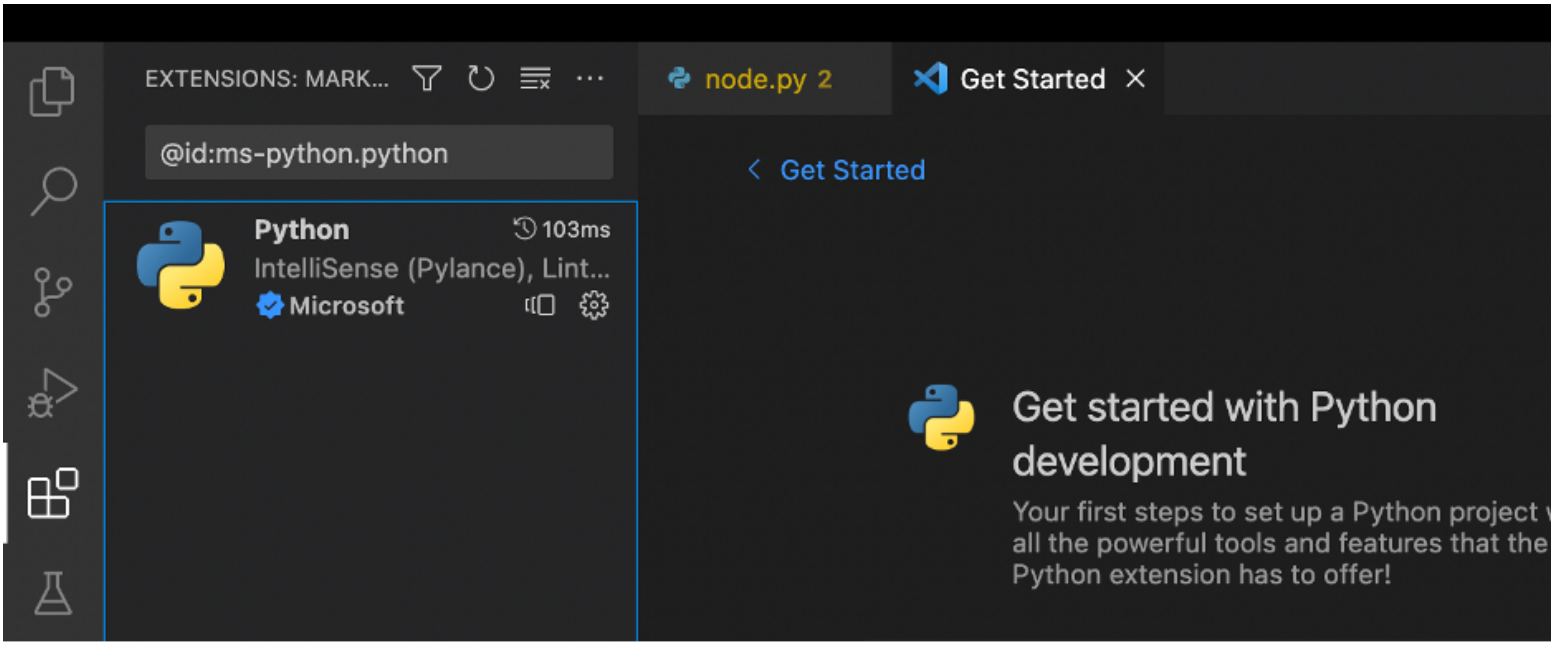
A hash is a function that converts an input of letters and numbers into an encrypted output of a fixed length. A hash is created using an algorithm, and is essential to blockchain management in cryptocurrency.

IMPLEMENTATION
Modules and Code:

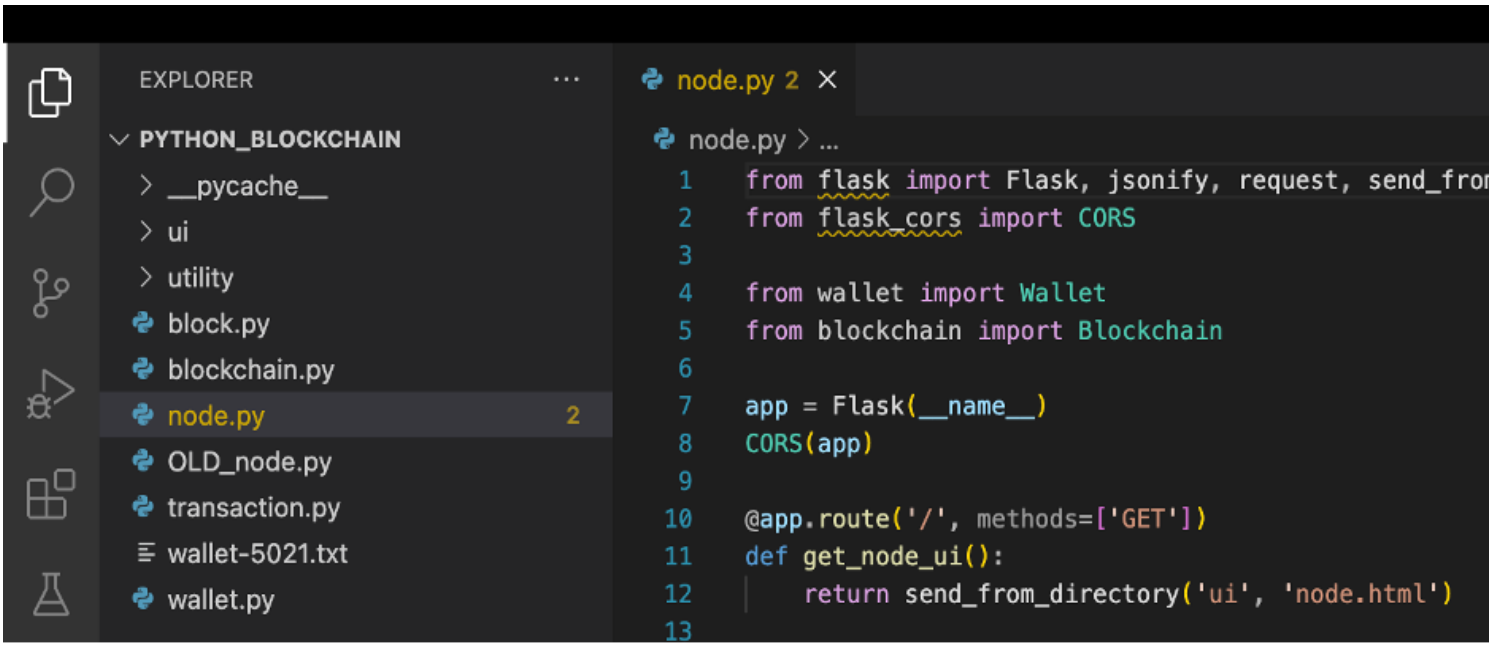
- Block.py
- Blockchain.py
- Node.py
- OLD_node.py
- Transaction.py
- Wallet.py
- Utility
  - Hash_util.py
  - Printable.py
  - Verification.py

Open Visual Studio code:

1) Install python from extension.



2) Download the project from GIT, open folder through visual studio code.



3) curl 'https://bootstrap.pypa.io/get-pip.py' > get-pip.py && sudo python3 get-pip.py

Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. It began as a simple wrapper around Werkzeug and Jinja and has become one of the most popular Python web application frameworks.

Flask offers suggestions, but doesn't enforce any dependencies or project layout. It is up to the developer to choose the tools and libraries they want to use. There are many extensions provided by the community that make adding new functionality easy.

4) python3 -m pip install requests

Requests allows you to send HTTP/1.1 requests extremely easily. There's no need to manually add query strings to your URLs, or to form-encode your PUT & POST data — but nowadays, just use the json method!

5) pip install -U flask-cors

This package has a simple philosophy: when you want to enable CORS, you wish to enable it for all use cases on a domain. This means no mucking around with different allowed headers, methods, etc.

This package exposes a Flask extension which by default enables CORS support on all routes, for all origins and methods. It allows parameterization of all CORS headers on a per-resource level. The package also contains a decorator, for those who prefer this approach.

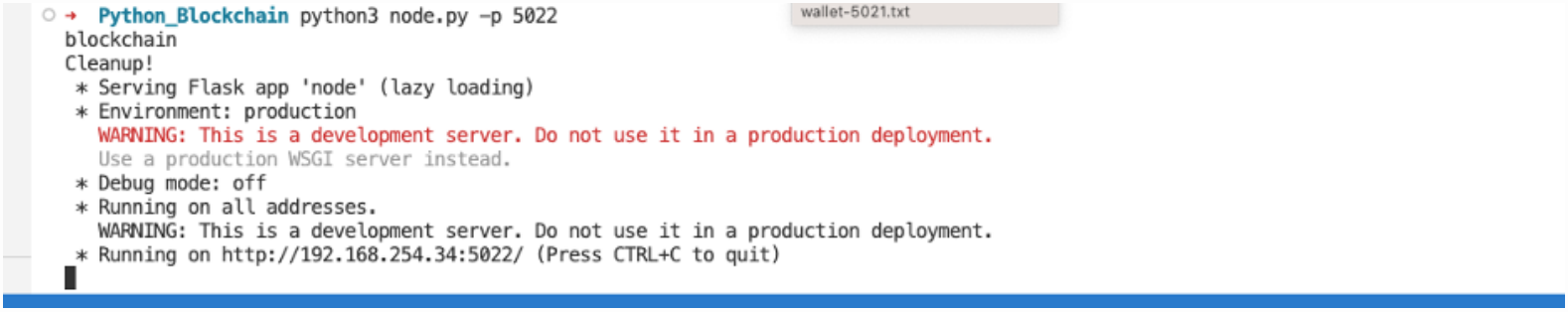
6) pip install pyCrypto

This is a collection of both secure hash functions (such as SHA256 and RIPEMD160), and various encryption algorithms (AES, DES, RSA, ElGamal, etc.). The package is structured to make adding new modules easy. This section is essentially complete, and the software interface will almost certainly not change in an incompatible way in the future; all that remains to be done is to fix any bugs that show up.

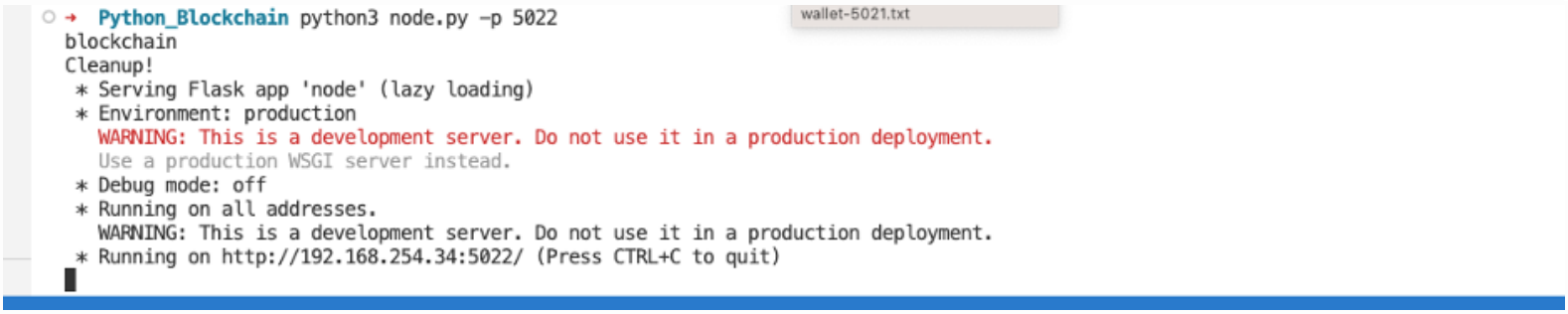
7) pip install -U pycrypto pycryptodome pycryptodomex PyJWT

Command to run the project:

1) Open two terminals
2) python3 node.py -p 5022

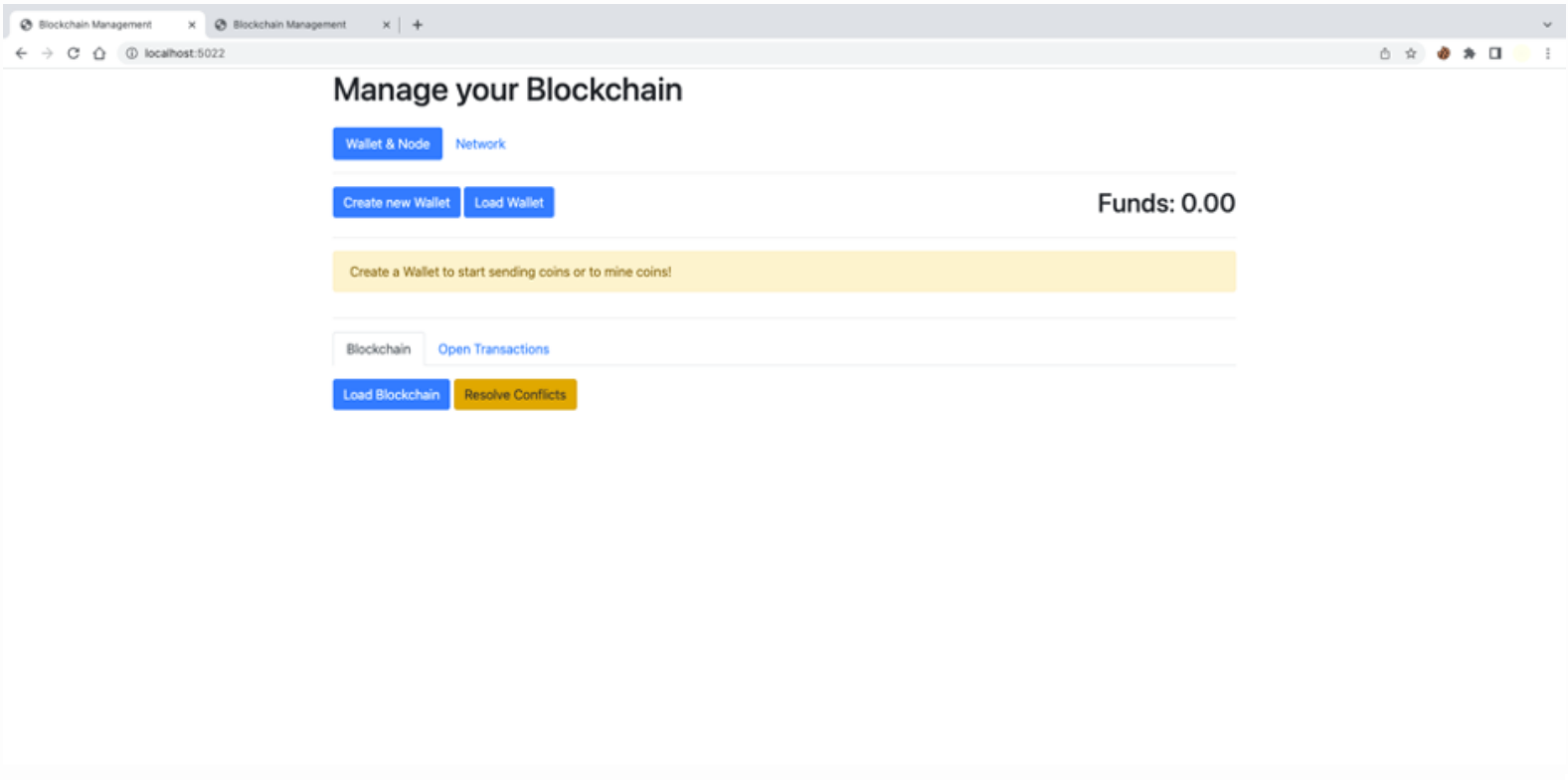


3) python3 node.py -p 5023

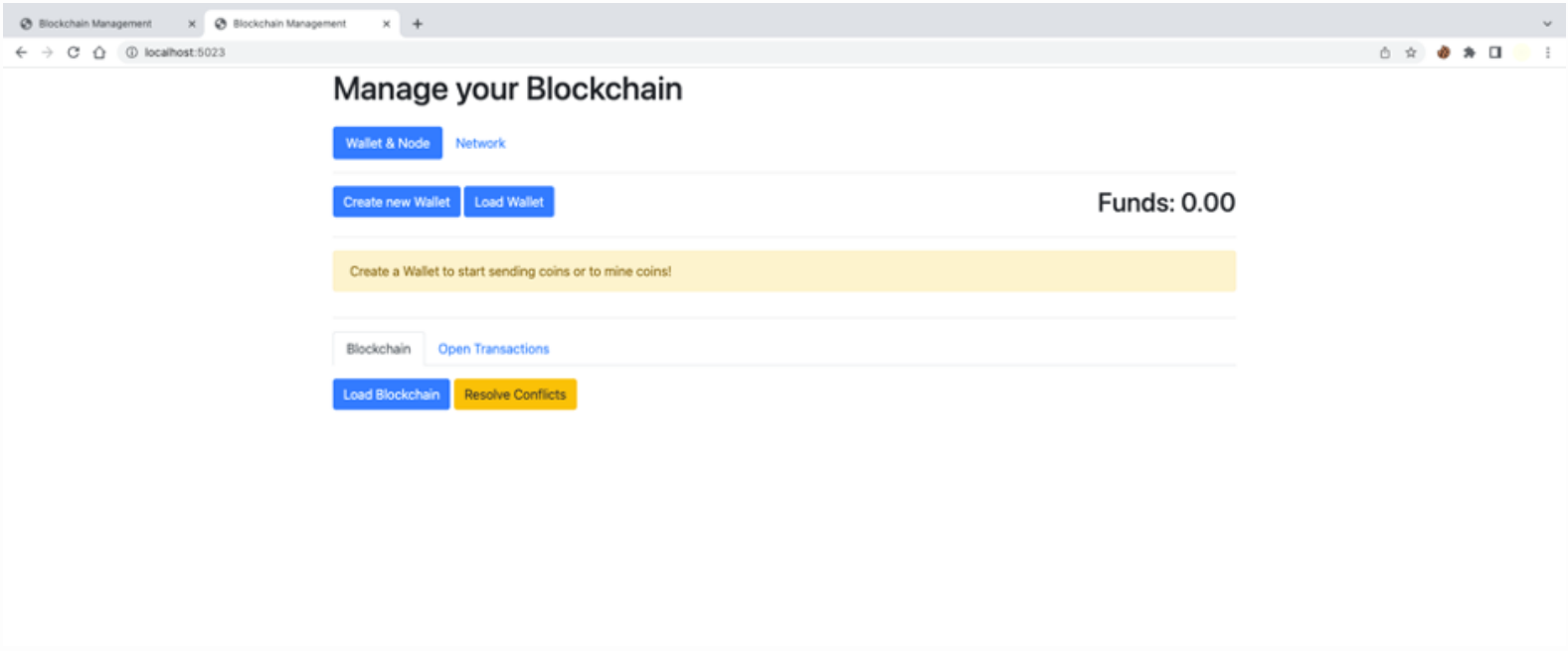


OUTPUT:

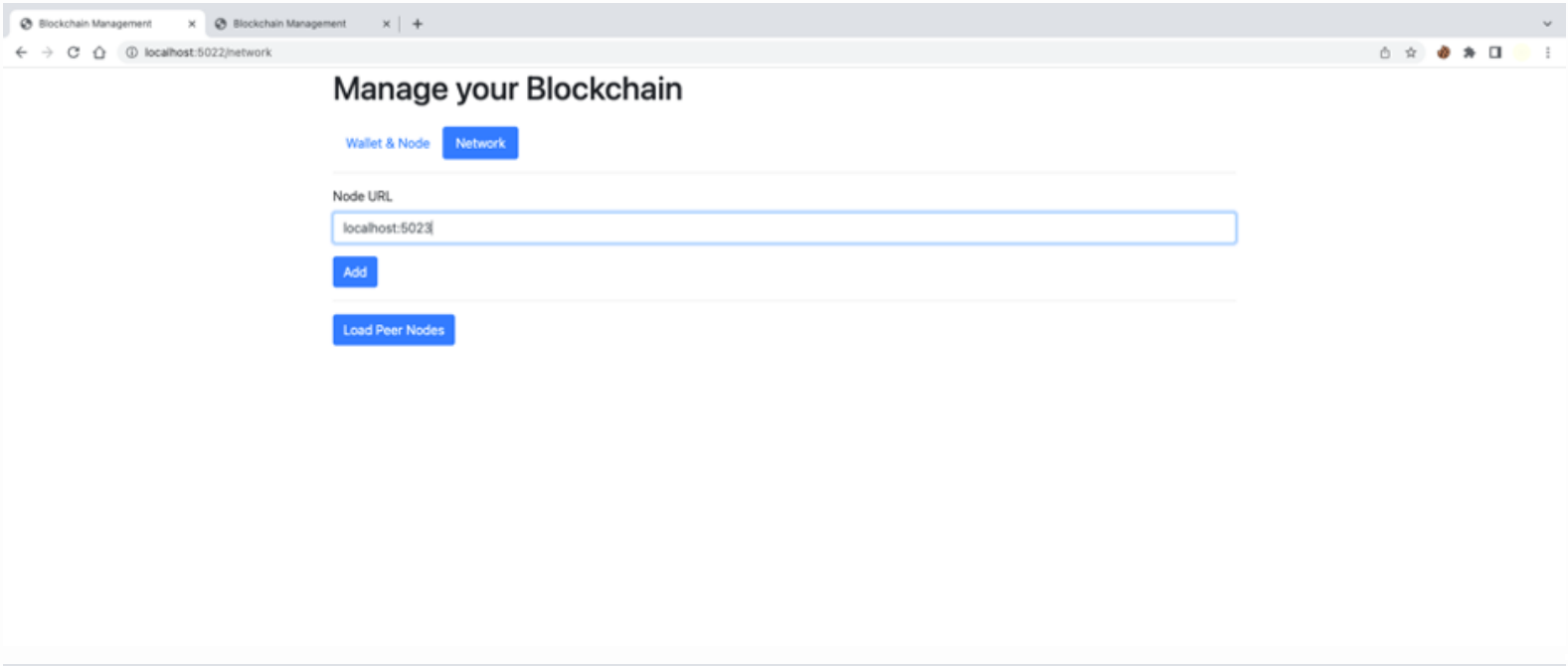
1) open the node http://localhost:5022

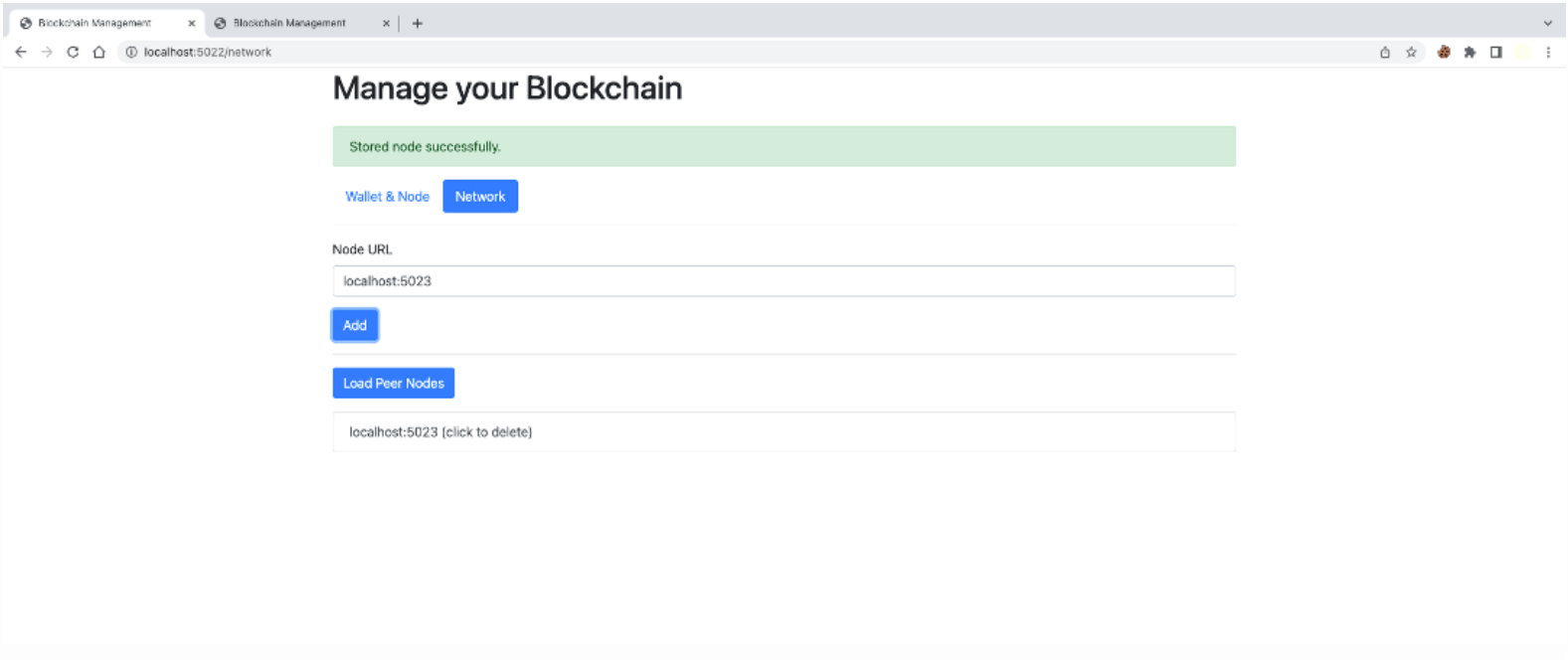


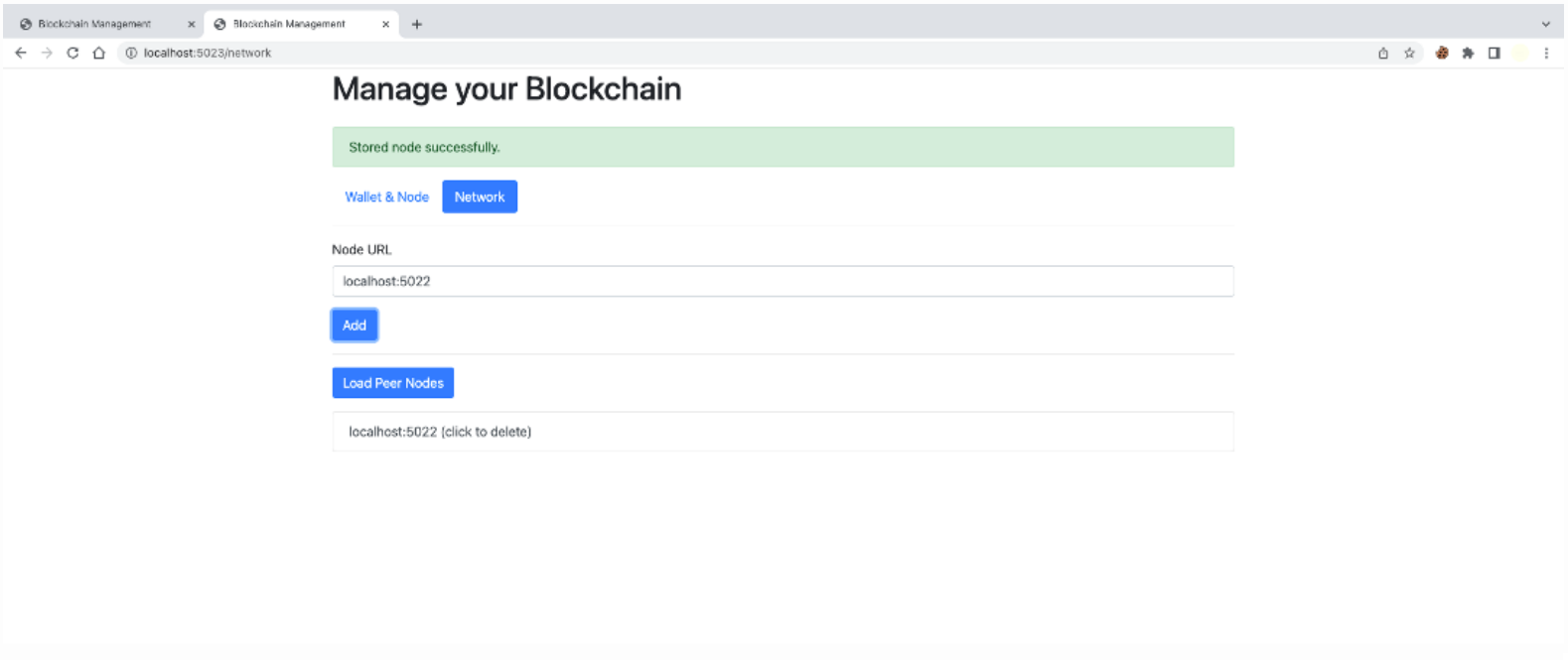
2) open another node http://localhost:5023



3) Click on the Network button to add both nodes to build the network. Add the different node URL in the Node URL textbox.



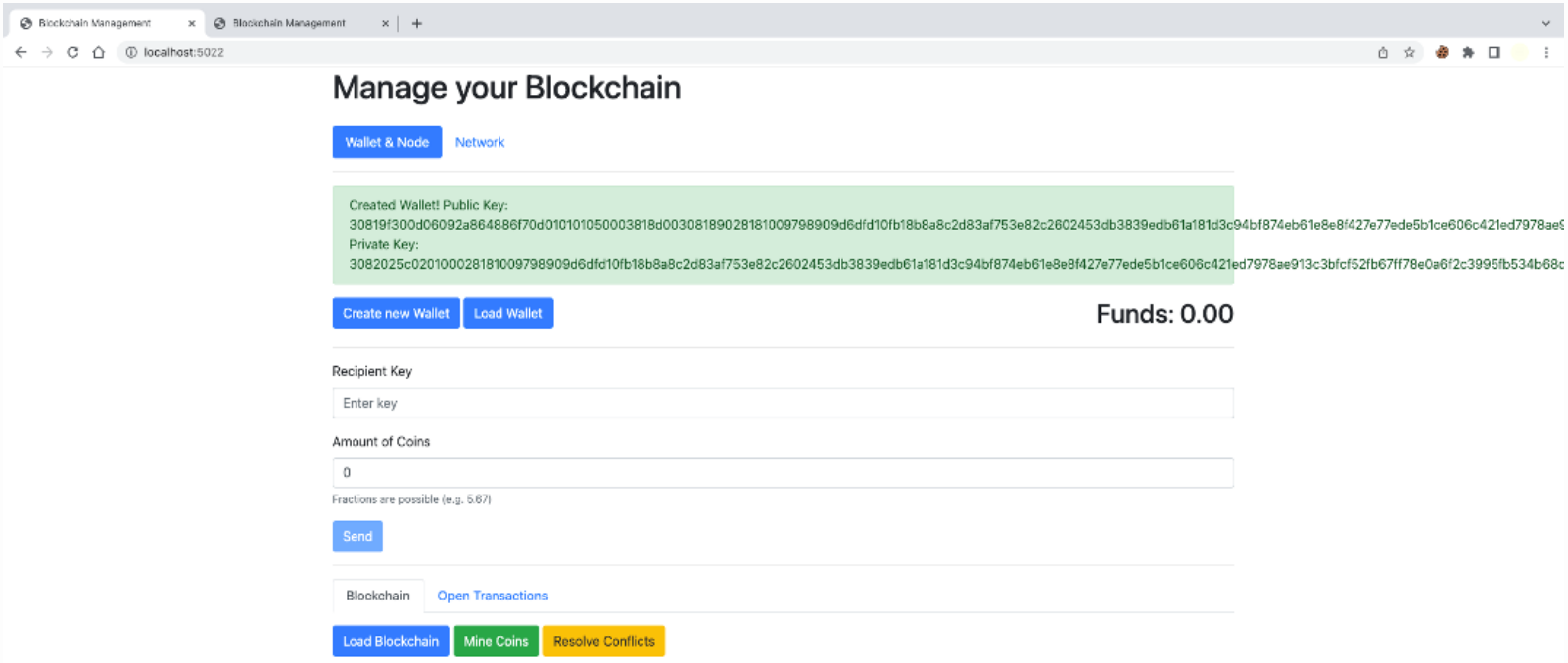




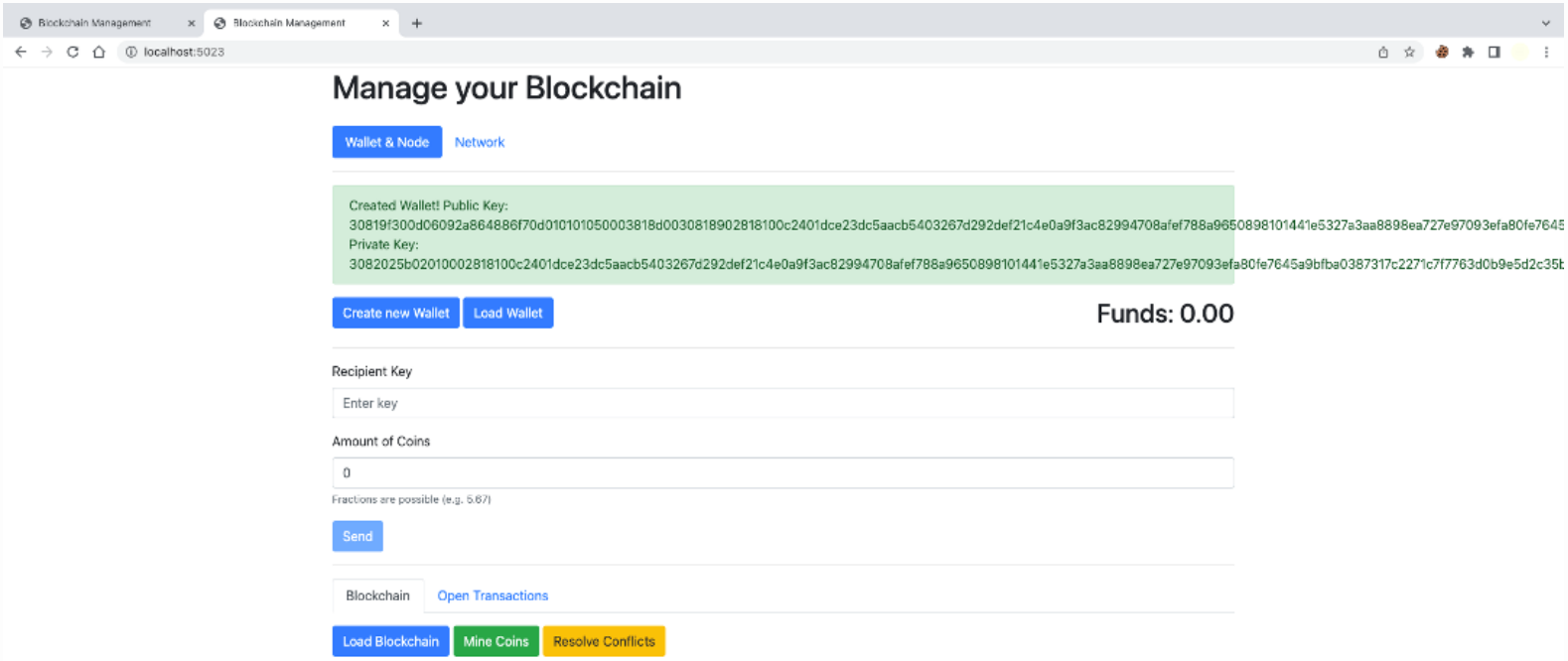
4) Load peer nodes are used to load the nodes.

5) Click on Wallet & Node and then click on Create new wallet button to get public and private keys in both nodes.

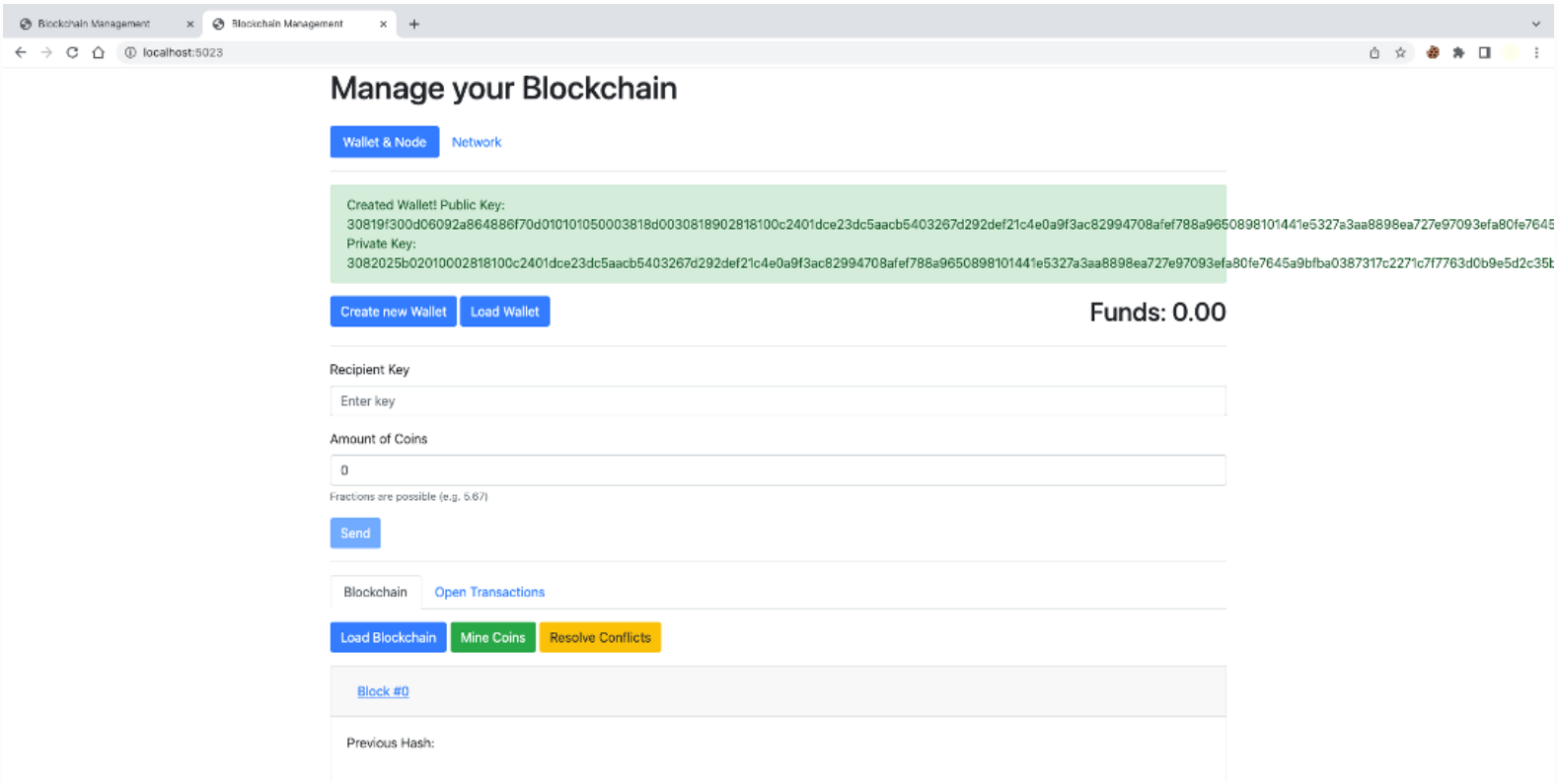
Node : 5022



Node: 5023



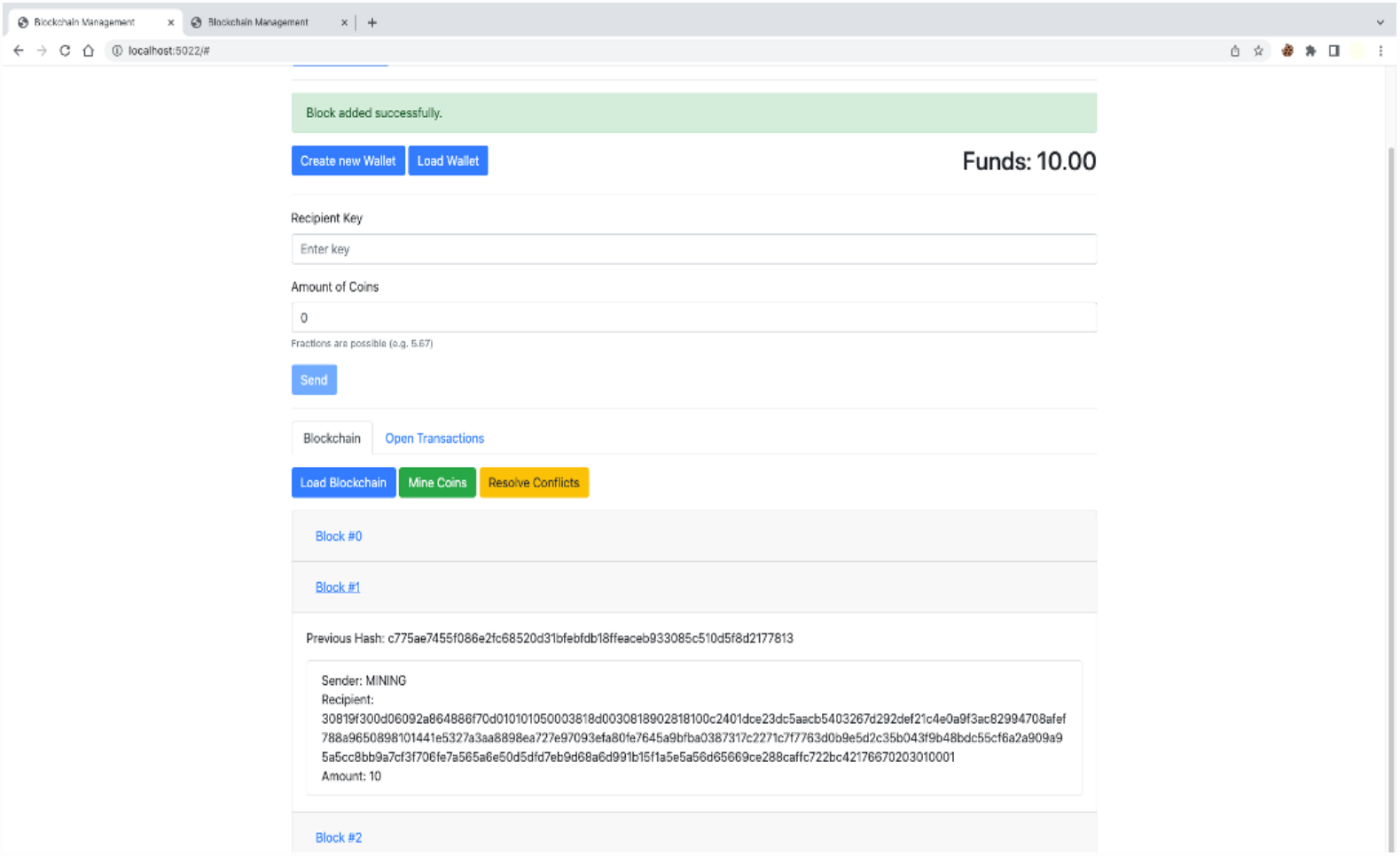
6) In the beginning, the Block having a hash null for both nodes.



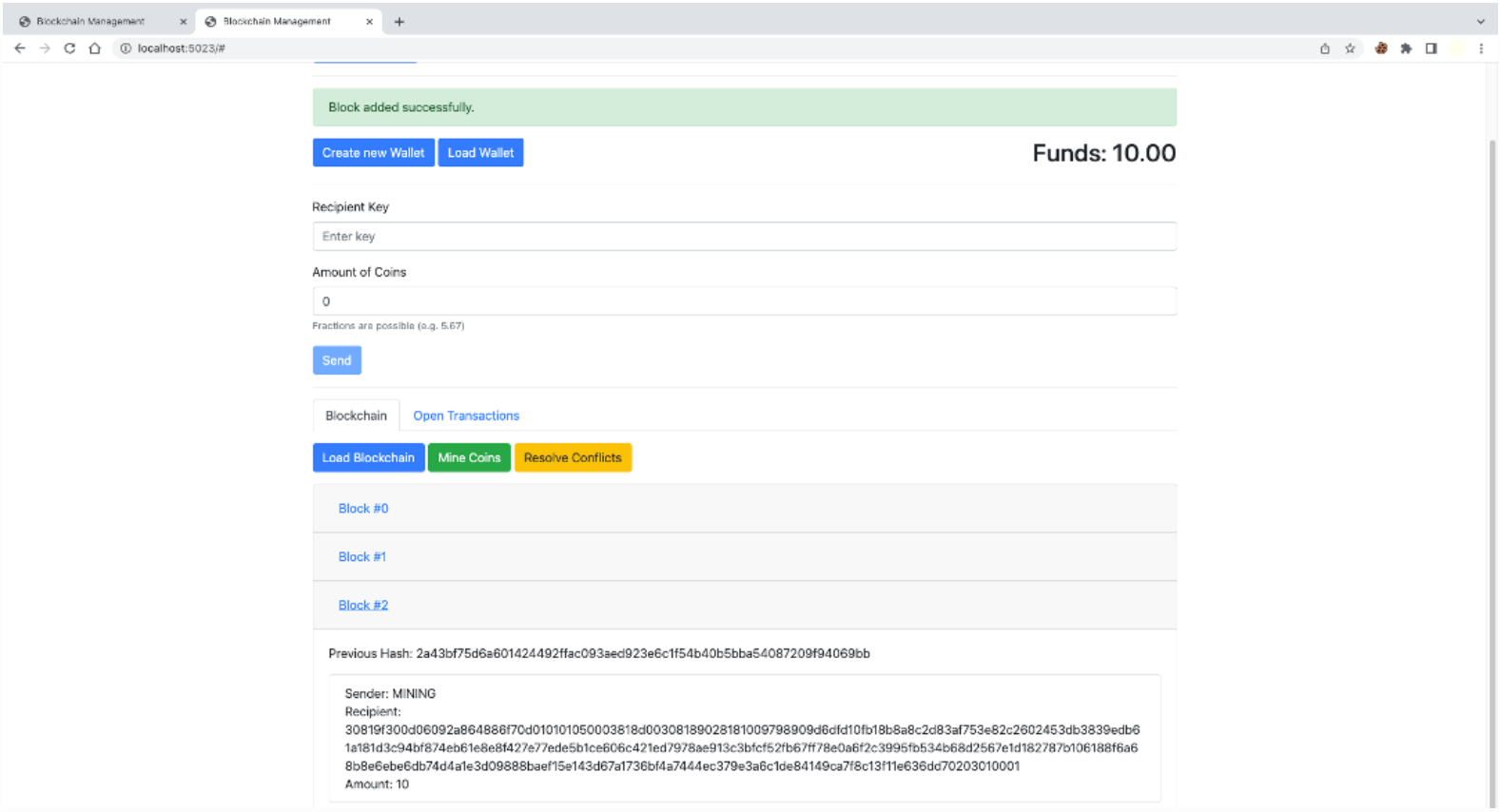
7) By default, By clicking on mine coins for both nodes provides 10 coins. Load Blockchain button is used to load the block details by providing the previous hash in the network. Copy of every block detail will be available in the network nodes.

Node 5022

Block #1

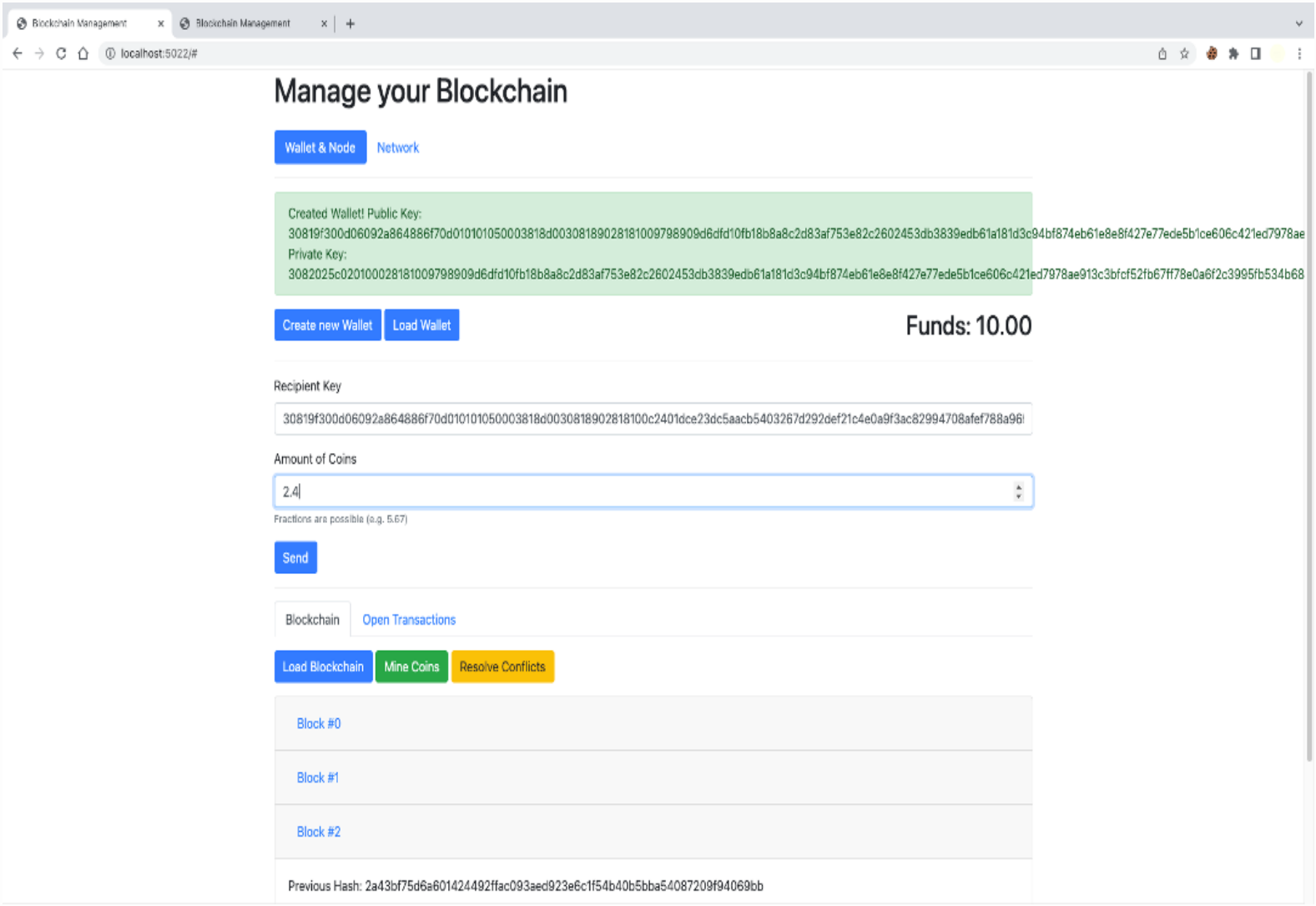


Node 5023
Block #2

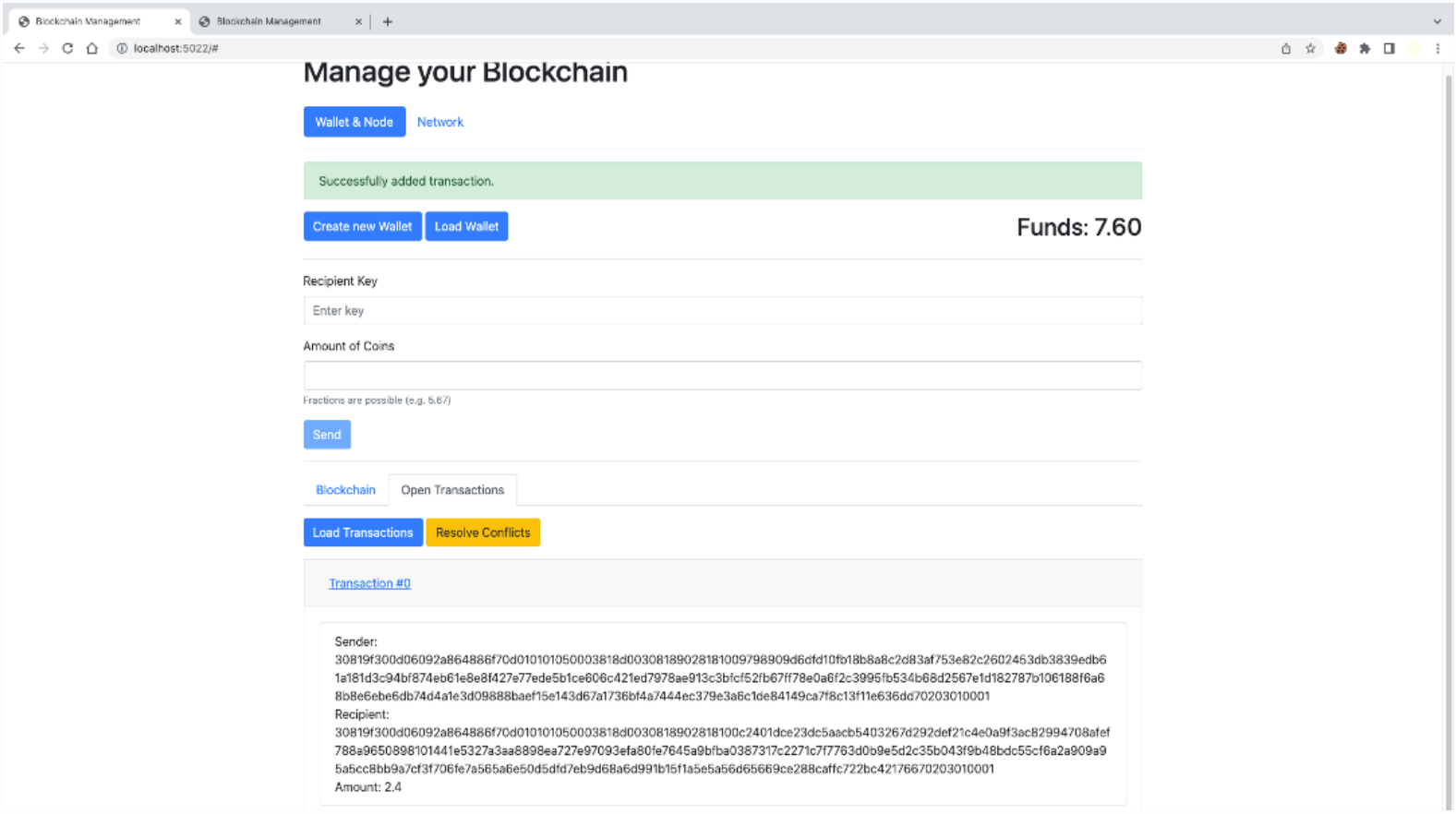


8) Load wallet is used to load the private and public keys.

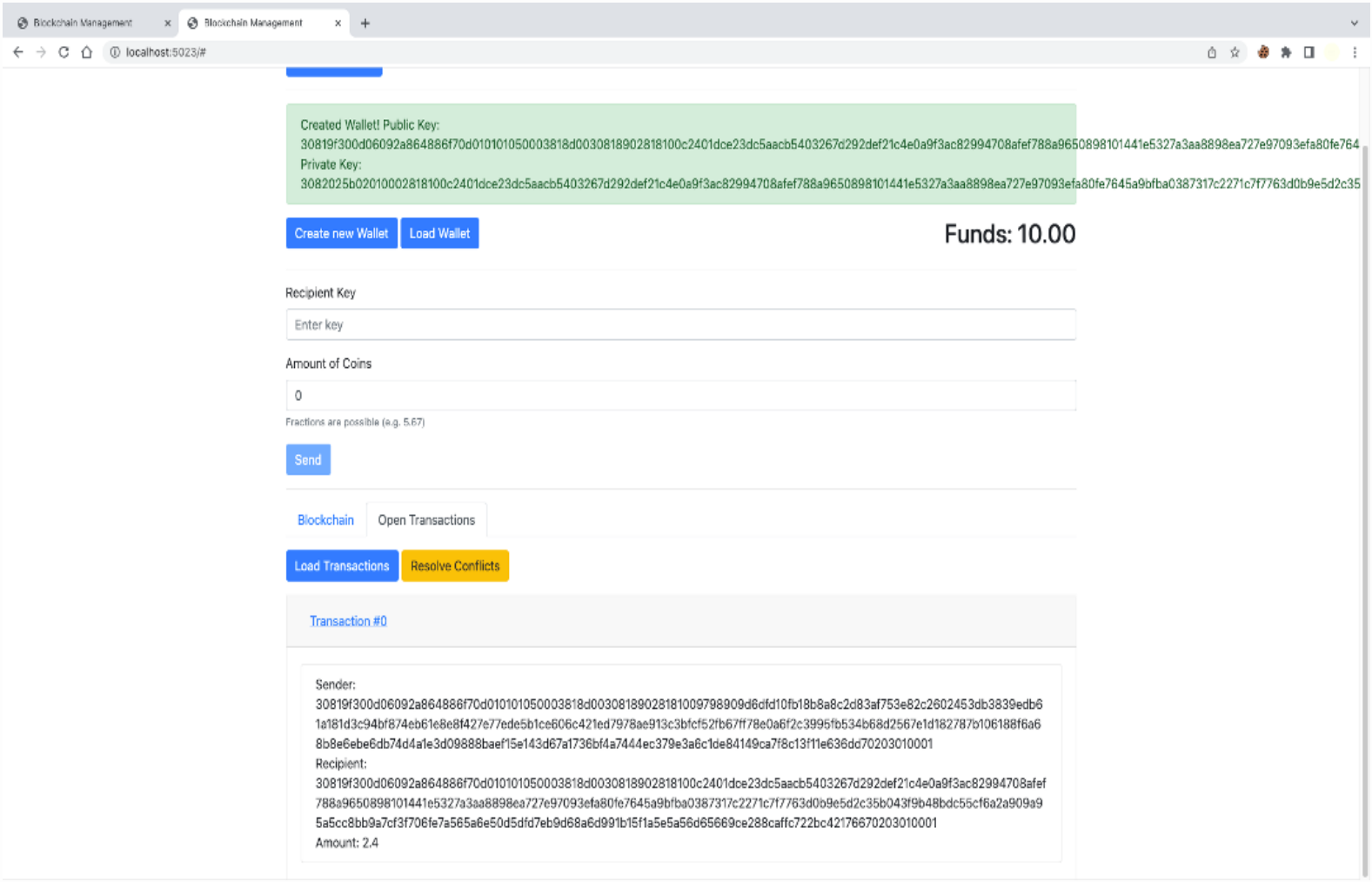
9) Enter the “public key of node 5023” in the Recipient Key text field and give any amount e.g (2.4) of node 5022.



10) Click on the open Transactions and then load Transactions, open transaction#0.

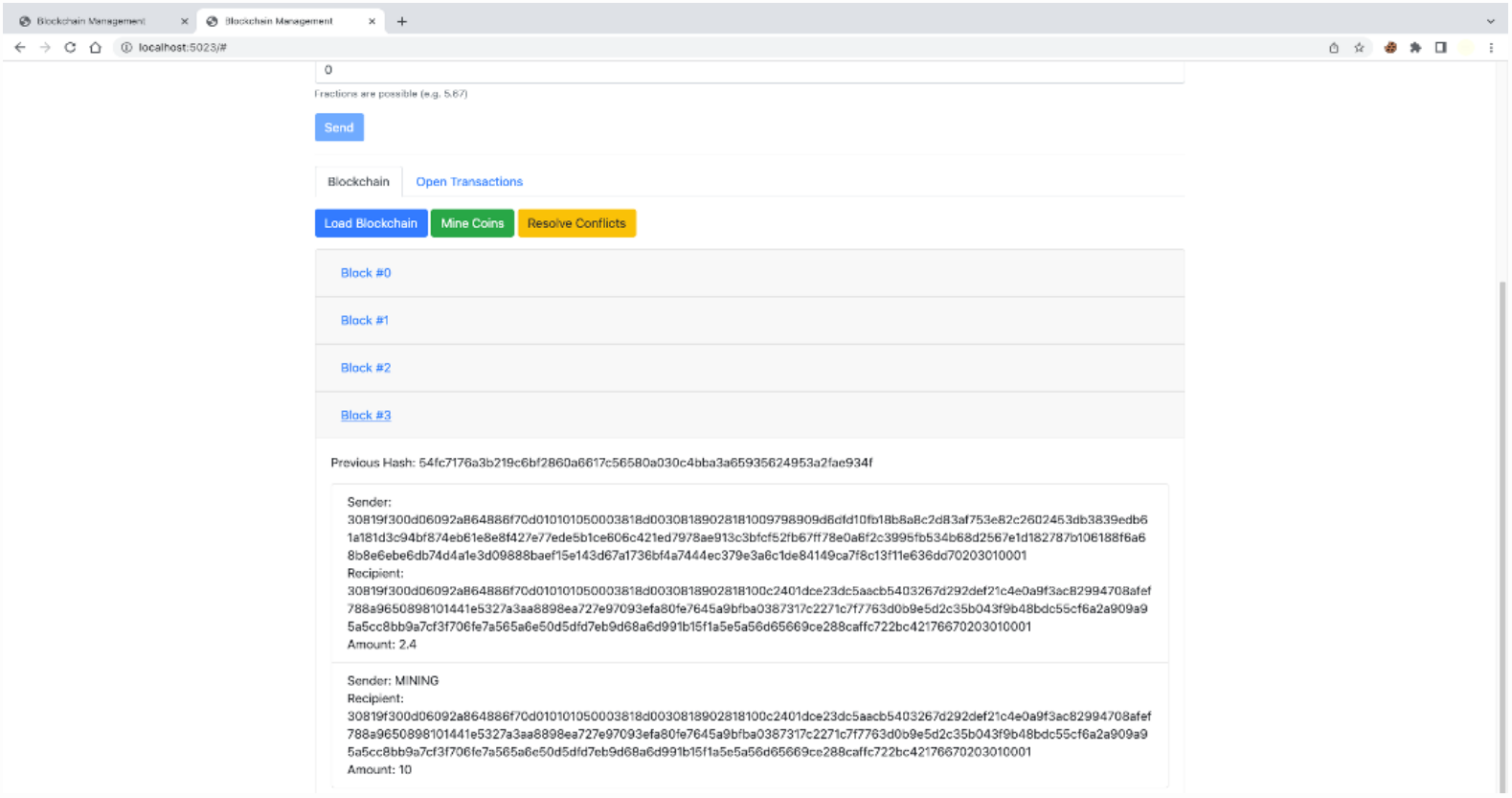


11) Validate the transaction copy in Node 5023.

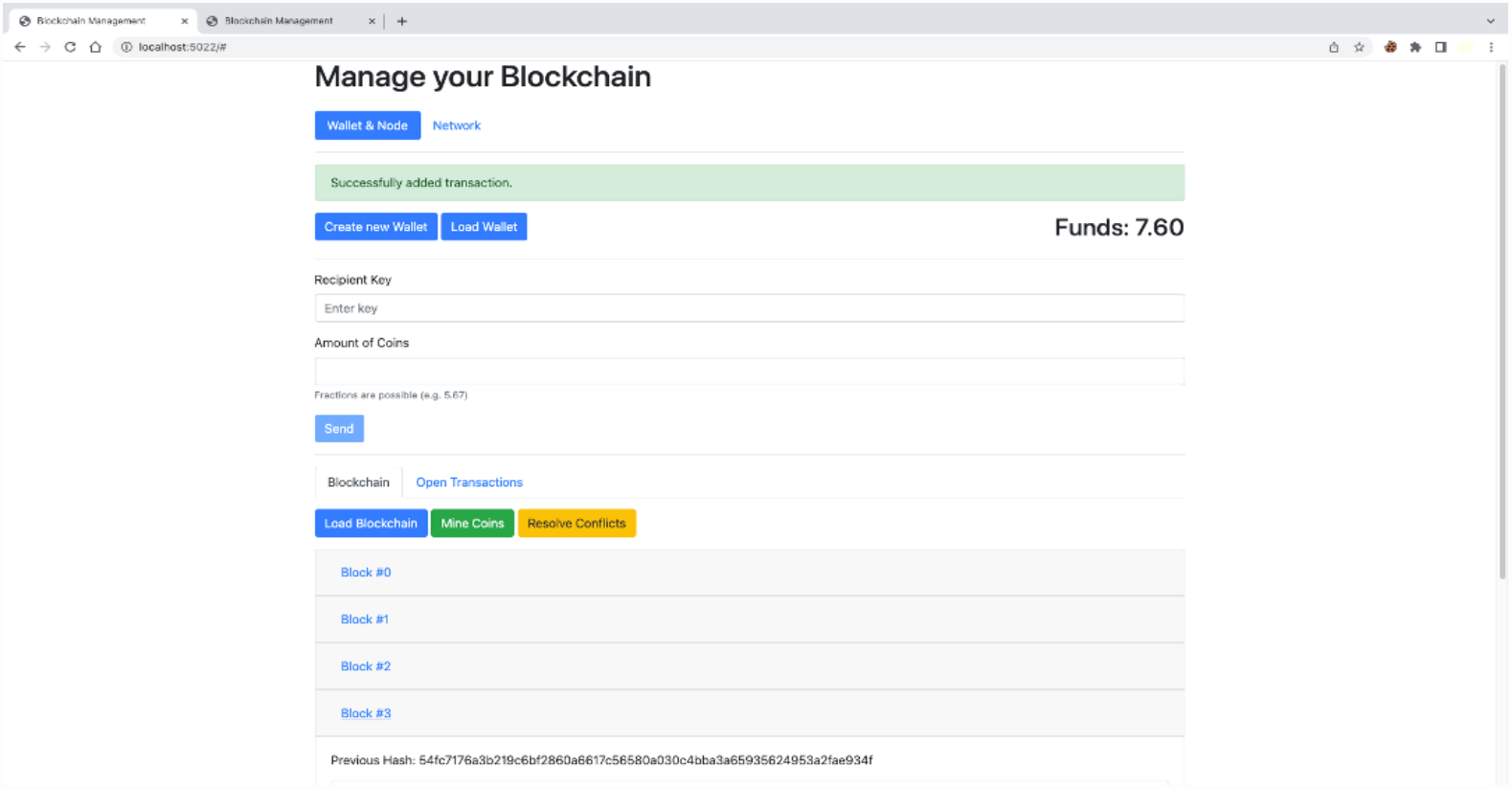


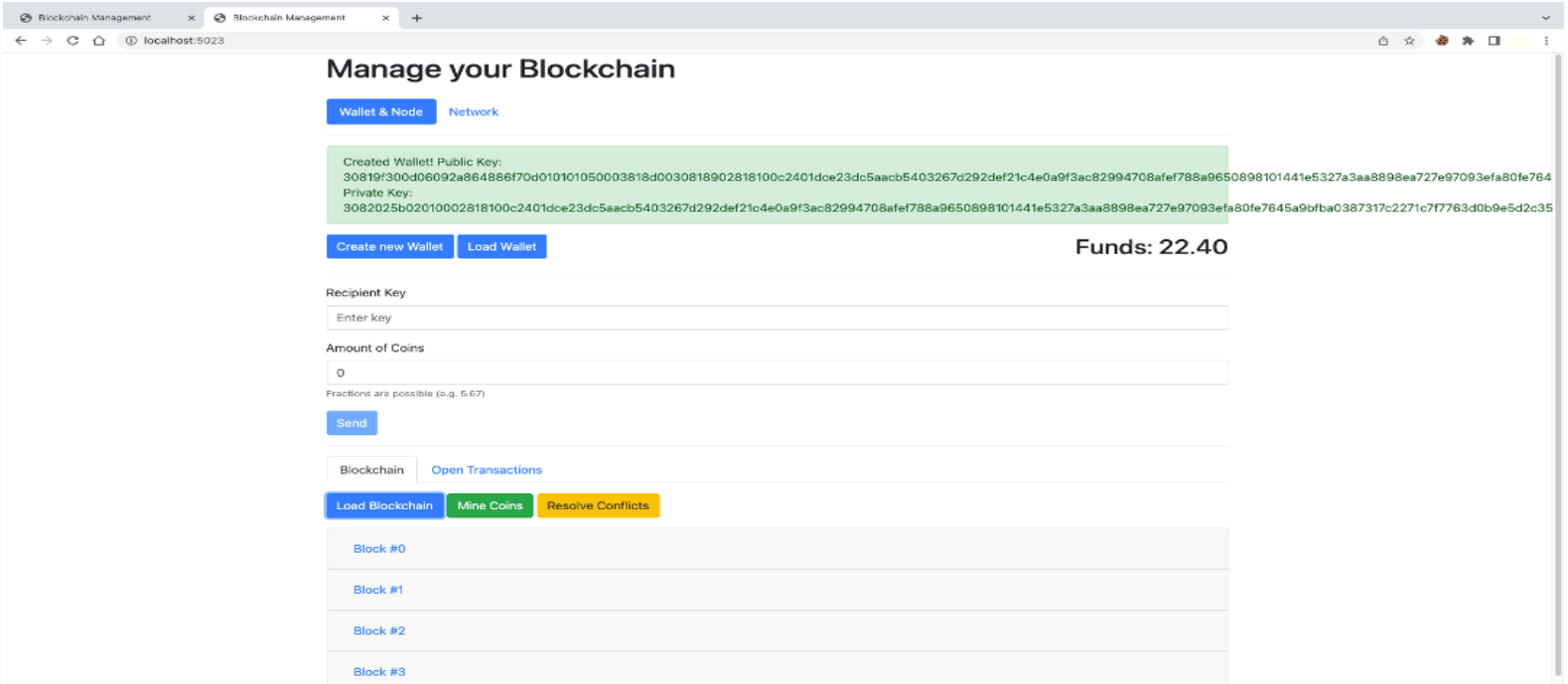
12) By clicking on the Mine coins button, Then the total coin is 10 + 10 + 2.4 = 22.40. Earlier 10 coins, After the Transaction of 2.4 from node 5022, getting 10 by clicking on the mine coins button.

Block #3



13) Resolve conflicts button is used to resolve the conflicts in the blocks and transactions.





14) We can do n times of transactions in the network.

CODE Explanation:

Block.py

```
from time import time

from utility.printable import Printable


class Block(Printable):
    """A single block of our blockchain.

    Attributes:
        :index: The index of this block.
        :previous_hash: The hash of the previous block in the blockchain.
        :timestamp: The timestamp of the block (automatically generated by
        default).
        :transactions: A list of transaction which are included in the block.
        :proof: The proof of work number that yielded this block.
    """

    def __init__(self, index, previous_hash, transactions, proof, time=time()):
        self.index = index
        self.previous_hash = previous_hash
        self.timestamp = time
        self.transactions = transactions
        self.proof = proof
```

OLD_node.py

```
from uuid import uuid4

from blockchain import Blockchain
from utility.verification import Verification
from wallet import Wallet

class Node:
    """The node which runs the local blockchain instance.

    Attributes:
        :id: The id of the node.
        :blockchain: The blockchain which is run by this node.
    """
    def __init__(self):
        # self.id = str(uuid4())
        self.wallet = Wallet()
        self.wallet.create_keys()
        self.blockchain = Blockchain(self.wallet.public_key)

    def get_transaction_value(self):
        """ Returns the input of the user (a new transaction amount) as a float. """
        # Get the user input, transform it from a string to a float and store it in user_input
        tx_recipient = input('Enter the recipient of the transaction: ')
        tx_amount = float(input('Your transaction amount please: '))
        return tx_recipient, tx_amount

    def get_user_choice(self):
        """Prompts the user for its choice and return it."""
        user_input = input('Your choice: ')
        return user_input

    def print_blockchain_elements(self):
        """ Output all blocks of the blockchain. """
        # Output the blockchain list to the console
        for block in self.blockchain.chain:
            print('Outputting Block')
            print(block)
        else:
            print('-' * 20)

    def listen_for_input(self):
```

```
        """Starts the node and waits for user input."""
        waiting_for_input = True

        # A while loop for the user input interface
        # It's a loop that exits once waiting_for_input becomes False or when break is called
        while waiting_for_input:
            print('Please choose')
            print('1: Add a new transaction value')
            print('2: Mine a new block')
            print('3: Output the blockchain blocks')
            print('4: Check transaction validity')
            print('5: Create wallet')
            print('6: Load wallet')
            print('7: Save keys')
            print('q: Quit')
            user_choice = self.get_user_choice()
            if user_choice == '1':
                tx_data = self.get_transaction_value()
                recipient, amount = tx_data
                # Add the transaction amount to the blockchain
                signature = self.wallet.sign_transaction(self.wallet.public_key, recipient, amount)
                            if  self.blockchain.add_transaction(recipient,  self.wallet.public_key,  signature,
amount=amount):
                    print('Added transaction!')
                else:
                    print('Transaction failed!')
                print(self.blockchain.get_open_transactions())
            elif user_choice == '2':
                if not self.blockchain.mine_block():
                    print('Mining failed. Got no wallet?')
            elif user_choice == '3':
                self.print_blockchain_elements()
            elif user_choice == '4':
                                if  Verification.verify_transactions(self.blockchain.get_open_transactions(),
self.blockchain.get_balance):
                    print('All transactions are valid')
                else:
                    print('There are invalid transactions')
            elif user_choice == '5':
                self.wallet.create_keys()
                self.blockchain = Blockchain(self.wallet.public_key)
            elif user_choice == '6':
```

```
                self.wallet.load_keys()
                self.blockchain = Blockchain(self.wallet.public_key)
            elif user_choice == '7':
                self.wallet.save_keys()
            elif user_choice == 'q':
                # This will lead to the loop to exist because it's running condition becomes False
                waiting_for_input = False
            else:
                print('Input was invalid, please pick a value from the list!')
            if not Verification.verify_chain(self.blockchain.chain):
                self.print_blockchain_elements()
                print('Invalid blockchain!')
                # Break out of the loop
                break
            print('Balance of {}: {:6.2f}'.format(self.wallet.public_key, self.blockchain.get_balance()))
        else:
            print('User left!')

        print('Done!')

if __name__ == '__main__':
    node = Node()
    node.listen_for_input()
```

Wallet.py

```
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5
from Crypto.Hash import SHA256
import Crypto.Random
import binascii


class Wallet:
    """Creates, loads and holds private and public keys. Manages transaction
    signing and verification."""

    def __init__(self, node_id):
        self.private_key = None
        self.public_key = None
        self.node_id = node_id

    def create_keys(self):
        """Create a new pair of private and public keys."""
        private_key, public_key = self.generate_keys()
        self.private_key = private_key
        self.public_key = public_key

    def save_keys(self):
        """Saves the keys to a file (wallet.txt)."""
        if self.public_key is not None and self.private_key is not None:
            try:
                with open('wallet-{}.txt'.format(self.node_id), mode='w') as f:
                    f.write(self.public_key)
                    f.write('\n')
                    f.write(self.private_key)
                return True
            except (IOError, IndexError):
                print('Saving wallet failed...')
                return False

    def load_keys(self):
        """Loads the keys from the wallet.txt file into memory."""
        try:
            with open('wallet-{}.txt'.format(self.node_id), mode='r') as f:
                keys = f.readlines()
```

```
                public_key = keys[0][:-1]
                private_key = keys[1]
                self.public_key = public_key
                self.private_key = private_key
            return True
        except (IOError, IndexError):
            print('Loading wallet failed...')
            return False

    def generate_keys(self):
        """Generate a new pair of private and public key."""
        private_key = RSA.generate(1024, Crypto.Random.new().read)
        public_key = private_key.publickey()
        return (
            binascii
            .hexlify(private_key.exportKey(format='DER'))
            .decode('ascii'),
            binascii
            .hexlify(public_key.exportKey(format='DER'))
            .decode('ascii')
        )

    def sign_transaction(self, sender, recipient, amount):
        """Sign a transaction and return the signature.

        Arguments:
            :sender: The sender of the transaction.
            :recipient: The recipient of the transaction.
            :amount: The amount of the transaction.
        """
        signer = PKCS1_v1_5.new(RSA.importKey(
            binascii.unhexlify(self.private_key)))
        h = SHA256.new((str(sender) + str(recipient) +
                        str(amount)).encode('utf8'))
        signature = signer.sign(h)
        return binascii.hexlify(signature).decode('ascii')

    @staticmethod
    def verify_transaction(transaction):
        """Verify the signature of a transaction.

        Arguments:
```

```
            :transaction: The transaction that should be verified.
        """
        public_key = RSA.importKey(binascii.unhexlify(transaction.sender))
        verifier = PKCS1_v1_5.new(public_key)
        h = SHA256.new((str(transaction.sender) + str(transaction.recipient) +
                        str(transaction.amount)).encode('utf8'))
        return verifier.verify(h, binascii.unhexlify(transaction.signature))
```

Node.py

```
from flask import Flask, jsonify, request, send_from_directory
from flask_cors import CORS

from wallet import Wallet
from blockchain import Blockchain

app = Flask(__name__)
CORS(app)

@app.route('/', methods=['GET'])
def get_node_ui():
    return send_from_directory('ui', 'node.html')


@app.route('/network', methods=['GET'])
def get_network_ui():
    return send_from_directory('ui', 'network.html')


@app.route('/wallet', methods=['POST'])
def create_keys():
    wallet.create_keys()
    if wallet.save_keys():
        global blockchain
        blockchain = Blockchain(wallet.public_key, port)
        response = {
            'public_key': wallet.public_key,
            'private_key': wallet.private_key,
            'funds': blockchain.get_balance()
        }
        return jsonify(response), 201
    else:
        response = {
            'message': 'Saving the keys failed.'
        }
        return jsonify(response), 500


@app.route('/wallet', methods=['GET'])
def load_keys():
```

```
    if wallet.load_keys():
        global blockchain
        blockchain = Blockchain(wallet.public_key, port)
        response = {
            'public_key': wallet.public_key,
            'private_key': wallet.private_key,
            'funds': blockchain.get_balance()
        }
        return jsonify(response), 201
    else:
        response = {
            'message': 'Loading the keys failed.'
        }
        return jsonify(response), 500


@app.route('/balance', methods=['GET'])
def get_balance():
    balance = blockchain.get_balance()
    if balance is not None:
        response = {
            'message': 'Fetched balance successfully.',
            'funds': balance
        }
        return jsonify(response), 200
    else:
        response = {
            'messsage': 'Loading balance failed.',
            'wallet_set_up': wallet.public_key is not None
        }
        return jsonify(response), 500


@app.route('/broadcast-transaction', methods=['POST'])
def broadcast_transaction():
    values = request.get_json()
    if not values:
        response = {'message': 'No data found.'}
        return jsonify(response), 400
    required = ['sender', 'recipient', 'amount', 'signature']
    if not all(key in values for key in required):
        response = {'message': 'Some data is missing.'}
```

```
        return jsonify(response), 400
    success = blockchain.add_transaction(
        values['recipient'],
        values['sender'],
        values['signature'],
        values['amount'],
        is_receiving=True)
    if success:
        response = {
            'message': 'Successfully added transaction.',
            'transaction': {
                'sender': values['sender'],
                'recipient': values['recipient'],
                'amount': values['amount'],
                'signature': values['signature']
            }
        }
        return jsonify(response), 201
    else:
        response = {
            'message': 'Creating a transaction failed.'
        }
        return jsonify(response), 500


@app.route('/broadcast-block', methods=['POST'])
def broadcast_block():
    values = request.get_json()
    if not values:
        response = {'message': 'No data found.'}
        return jsonify(response), 400
    if 'block' not in values:
        response = {'message': 'Some data is missing.'}
        return jsonify(response), 400
    block = values['block']
    if block['index'] == blockchain.chain[-1].index + 1:
        if blockchain.add_block(block):
            response = {'message': 'Block added'}
            return jsonify(response), 201
        else:
            response = {'message': 'Block seems invalid.'}
            return jsonify(response), 409
```

```
    elif block['index'] > blockchain.chain[-1].index:
        response = {
            'message': 'Blockchain seems to differ from local blockchain.'}
        blockchain.resolve_conflicts = True
        return jsonify(response), 200
    else:
        response = {
            'message': 'Blockchain seems to be shorter, block not added'}
        return jsonify(response), 409


@app.route('/transaction', methods=['POST'])
def add_transaction():
    if wallet.public_key is None:
        response = {
            'message': 'No wallet set up.'
        }
        return jsonify(response), 400
    values = request.get_json()
    if not values:
        response = {
            'message': 'No data found.'
        }
        return jsonify(response), 400
    required_fields = ['recipient', 'amount']
    if not all(field in values for field in required_fields):
        response = {
            'message': 'Required data is missing.'
        }
        return jsonify(response), 400
    recipient = values['recipient']
    amount = values['amount']
    signature = wallet.sign_transaction(wallet.public_key, recipient, amount)
    success = blockchain.add_transaction(
        recipient, wallet.public_key, signature, amount)
    if success:
        response = {
            'message': 'Successfully added transaction.',
            'transaction': {
                'sender': wallet.public_key,
                'recipient': recipient,
                'amount': amount,
```

```
                'signature': signature
            },
            'funds': blockchain.get_balance()
        }
        return jsonify(response), 201
    else:
        response = {
            'message': 'Creating a transaction failed.'
        }
        return jsonify(response), 500


@app.route('/mine', methods=['POST'])
def mine():
    if blockchain.resolve_conflicts:
        response = {'message': 'Resolve conflicts first, block not added!'}
        return jsonify(response), 409
    block = blockchain.mine_block()
    if block is not None:
        dict_block = block.__dict__.copy()
        dict_block['transactions'] = [
            tx.__dict__ for tx in dict_block['transactions']]
        response = {
            'message': 'Block added successfully.',
            'block': dict_block,
            'funds': blockchain.get_balance()
        }
        return jsonify(response), 201
    else:
        response = {
            'message': 'Adding a block failed.',
            'wallet_set_up': wallet.public_key is not None
        }
        return jsonify(response), 500


@app.route('/resolve-conflicts', methods=['POST'])
def resolve_conflicts():
    replaced = blockchain.resolve()
    if replaced:
        response = {'message': 'Chain was replaced!'}
    else:
```

```
        response = {'message': 'Local chain kept!'}
    return jsonify(response), 200


@app.route('/transactions', methods=['GET'])
def get_open_transaction():
    transactions = blockchain.get_open_transactions()
    dict_transactions = [tx.__dict__ for tx in transactions]
    return jsonify(dict_transactions), 200


@app.route('/chain', methods=['GET'])
def get_chain():
    chain_snapshot = blockchain.chain
    dict_chain = [block.__dict__.copy() for block in chain_snapshot]
    for dict_block in dict_chain:
        dict_block['transactions'] = [
            tx.__dict__ for tx in dict_block['transactions']]
    return jsonify(dict_chain), 200


@app.route('/node', methods=['POST'])
def add_node():
    values = request.get_json()
    if not values:
        response = {
            'message': 'No data attached.'
        }
        return jsonify(response), 400
    if 'node' not in values:
        response = {
            'message': 'No node data found.'
        }
        return jsonify(response), 400
    node = values['node']
    blockchain.add_peer_node(node)
    response = {
        'message': 'Node added successfully.',
        'all_nodes': blockchain.get_peer_nodes()
    }
    return jsonify(response), 201
```

```python
@app.route('/node/<node_url>', methods=['DELETE'])
def remove_node(node_url):
    if node_url == '' or node_url is None:
        response = {
            'message': 'No node found.'
        }
        return jsonify(response), 400
    blockchain.remove_peer_node(node_url)
    response = {
        'message': 'Node removed',
        'all_nodes': blockchain.get_peer_nodes()
    }
    return jsonify(response), 200


@app.route('/nodes', methods=['GET'])
def get_nodes():
    nodes = blockchain.get_peer_nodes()
    response = {
        'all_nodes': nodes
    }
    return jsonify(response), 200


if __name__ == '__main__':
    from argparse import ArgumentParser
    parser = ArgumentParser()
    parser.add_argument('-p', '--port', type=int, default=5000)
    args = parser.parse_args()
    port = args.port
    wallet = Wallet(port)
    blockchain = Blockchain(wallet.public_key, port)
    app.run(host='0.0.0.0', port=port)
```

Transaction.py

```
from collections import OrderedDict

from utility.printable import Printable


class Transaction(Printable):
    """A transaction which can be added to a block in the blockchain.

    Attributes:
        :sender: The sender of the coins.
        :recipient: The recipient of the coins.
        :signature: The signature of the transaction.
        :amount: The amount of coins sent.
    """

    def __init__(self, sender, recipient, signature, amount):
        self.sender = sender
        self.recipient = recipient
        self.amount = amount
        self.signature = signature

    def to_ordered_dict(self):
        """Converts this transaction into a (hashable) OrderedDict."""
        return OrderedDict([('sender', self.sender),
                            ('recipient', self.recipient),
                            ('amount', self.amount)])
```