

## HW2 Q2 Documentation

Jing Jiang & Zheng Liu

a)

```
9
10 class Blockchain:
11     def __init__(self):
12         self.current_tx = []
13         self.chain = []
14         self.nodes = set()
15
16         self.new_genesis_block()
17
18     def new_genesis_block(self):
19         """
20         """
21         first_hash = "0000000000000000000000000000000000000000000000000000000000000000"
22         first_time = time()
23
24         block = {
25             'index': len(self.chain) + 1,
26             'timestamp': first_time,
27             'transactions': self.current_tx,
28             'nonce': self.proof_of_work(first_hash, first_time),
29             'prev_hash': first_hash,
30         }
31         self.current_tx = []
32
33         self.chain.append(block)
34         return block
35
```

In the screenshot above, I created a Blockchain class. Inside the constructor, 3 properties are defined. The *current\_tx* is the transactions pending to be added to the next block. The chain stores the data structure of the blockchain. And nodes is related to the consensus algorithms. Then, I initialize it with a method *new\_genesis\_block*. Unlike the tutorial, I didn't use the *new\_block* method, because the author of the tutorial assigned a nonce instead of mining to obtain one, which will cause the initial block having a unverified hash.

```

36
37 def new_block(self, nonce, prev_hash, timestamp):
38     """
39     """
40     block = {
41         'index': len(self.chain) + 1,
42         'timestamp': timestamp,
43         'transactions': self.current_tx,
44         'nonce': nonce or self.proof_of_work(prev_hash, timestamp),
45         'prev_hash': prev_hash or self.hash(self.chain[-1]),
46     }
47     self.current_tx = []
48
49     self.chain.append(block)
50     return block
51
52 def new_transaction(self, recipient, sender, amount):
53     """
54     """
55     self.current_tx.append({
56         'recipient': recipient,
57         'sender': sender,
58         'amount': amount,
59     })
60
61     return self.last_block['index'] + 1
62

```

In the *new\_block* method, a dictionary called block is defined and its nonce is mined to make the block verified. The *current\_tx* will be reset, meaning that all transactions are added to the new block. And then the block is appended to the chain.

In the *new\_transaction* method, a dictionary with keys: recipient, sender and amount is appended to the transaction list. The method will return the index of the block which this transaction will be added to.

```

63     @property
64     def last_block(self):
65         if len(self.chain)>0:
66             return self.chain[-1]
67         else:
68             return None
69
70     @staticmethod
71     def hash(block):
72         """
73         """
74         block_string = json.dumps(block, sort_keys=True).encode()
75         return hashlib.sha256(block_string).hexdigest()
76
77     def proof_of_work(self, last_hash, timestamp):
78         """
79         """
80         if self.last_block is not None:
81             current_index = self.last_block['index'] + 1
82         else:
83             current_index = 1
84
85         nonce = 0
86         while self.valid_proof(current_index, timestamp, self.current_tx, nonce, last_hash) is False:
87             nonce += 1
88
89         return nonce
90
91     @staticmethod
92     def valid_proof(index, timestamp, transactions, nonce, last_hash):
93         """
94         """
95         block_to_verify = {
96             'index': index,
97             'timestamp': timestamp,
98             'transactions': transactions,
99             'nonce': nonce,
100             'prev_hash': last_hash,
101         }
102
103         guess_string = json.dumps(block_to_verify, sort_keys=True).encode()
104         guess_hash = hashlib.sha256(guess_string).hexdigest()
105         return guess_hash[:4] == "0000"
106

```

As shown above, the *last\_block* returns the last block of the chain if the chain is not empty. The hash method uses the json library and the hashlib library to provide the hash code of a block.

The *proof\_of\_work* method and the *valid\_proof* method are related to the verification of a block. The while loop in the *proof\_of\_work* method tries to pass a integer as a nonce to the *valid\_proof* method to see if this nonce can make the block verified. If *valid\_proof* returns true, the loop ends and the nonce is returned.

Inside the *valid\_proof* method, every time a dictionary called *block\_to\_verify* is created. This is basically a block pending to be verified. Every time this block is JSON serialized and hashed. If the hash code has 4 preceding zeros, this method returns true, indicating that this block is verified.

b)

In the example, the author simplified the hash puzzle of the proof of work greatly by only passing the nonce of the previous block to the method to calculate the hash. He concatenated the previous nonce (what he called proof) with the current nonce and hashed them until the hash got four preceding zeros. However, what we discussed in class is that the hash of the block should be verified with 4 zeros, not a concatenated string. So I implemented my own method to replace the simplified method, as shown and explain in part a.

c)

```
179 @app.route('/mine', methods=['GET'])
180 def mine():
181     timestp = time()
182     blockchain.new_transaction(
183         recipient=node_id,
184         sender="coinbase",
185         amount=100,
186     )
187     last_block = blockchain.last_block
188
189     prev_hash = blockchain.hash(last_block)
190
191     nonce = blockchain.proof_of_work(prev_hash, timestp)
192
193     block = blockchain.new_block(nonce, prev_hash, timestp)
194
195     response = {
196         'message': "New Block Forged",
197         'index': block['index'],
198         'transactions': block['transactions'],
199         'nonce': block['nonce'],
200         'prev_hash': block['prev_hash'],
201         'block_hash': blockchain.hash(block)
202     }
203
204     return jsonify(response), 200
205
```

The first method(endpoint) is `/mine`. This method allows me to mine the new block. Every time this method is called, a block reward transaction is added. The sender of this transaction is `"coinbase"`, and the receiver is whoever who mined the block. Then the nonce is find and a new block is created. A response showing the information of this block is returned.

```

207 @app.route('/transactions/new', methods=['POST'])
208 def new_transaction():
209     values = request.get_json()
210
211     required = ['recipient', 'sender', 'amount']
212
213     if not all(k in values for k in required):
214         return 'Not enough values', 400
215
216     ind = blockchain.new_transaction(values['recipient'], values['sender'], values['amount'])
217
218     response = {'message': f'The transaction will be added to Block {ind}'}
219
220     return jsonify(response), 201
221

```

The endpoint above is `/transactions/new`. This method creates new transactions. Unlike `/mine`, this method is a POST request, in which information are sent from local to server. The keywords `'recipient'`, `'sender'` and `'amount'` are specified and a response is returned if this works correctly.

```

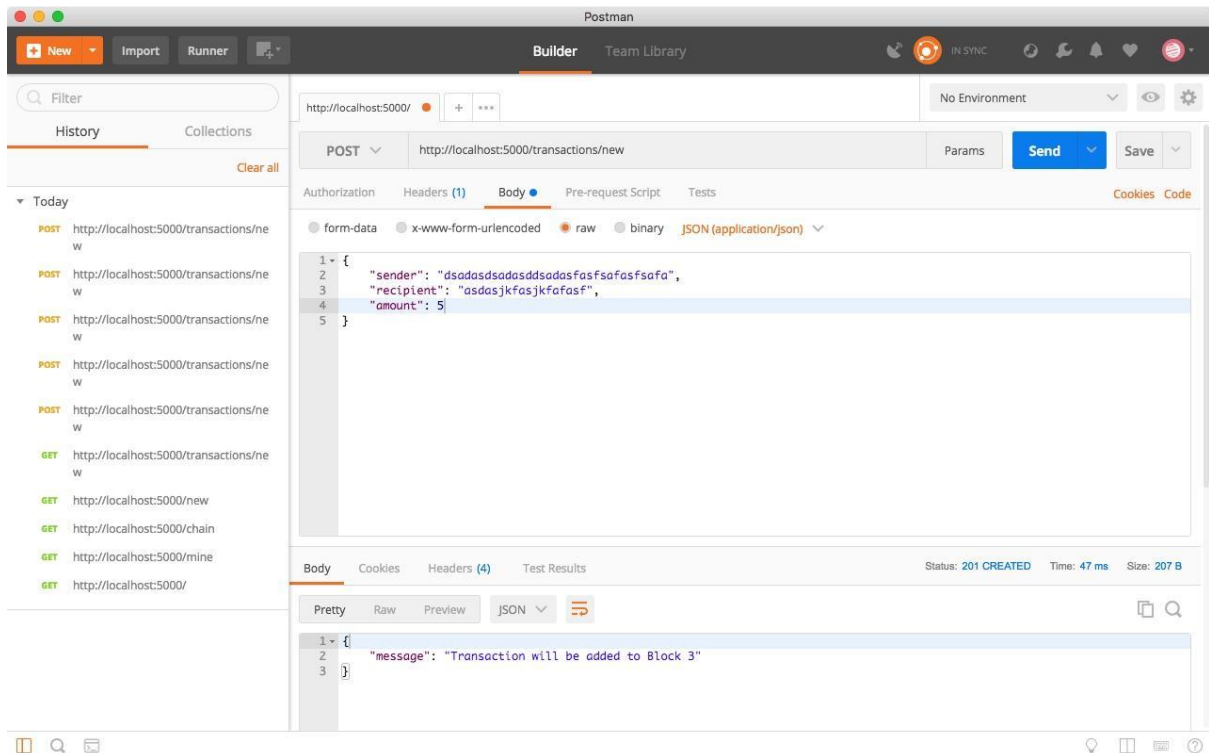
223 @app.route('/chain', methods=['GET'])
224 def full_chain():
225     response = {
226         'Blockchain': blockchain.chain,
227         'length of the chain': len(blockchain.chain),
228     }
229
230     return jsonify(response), 200
231

```

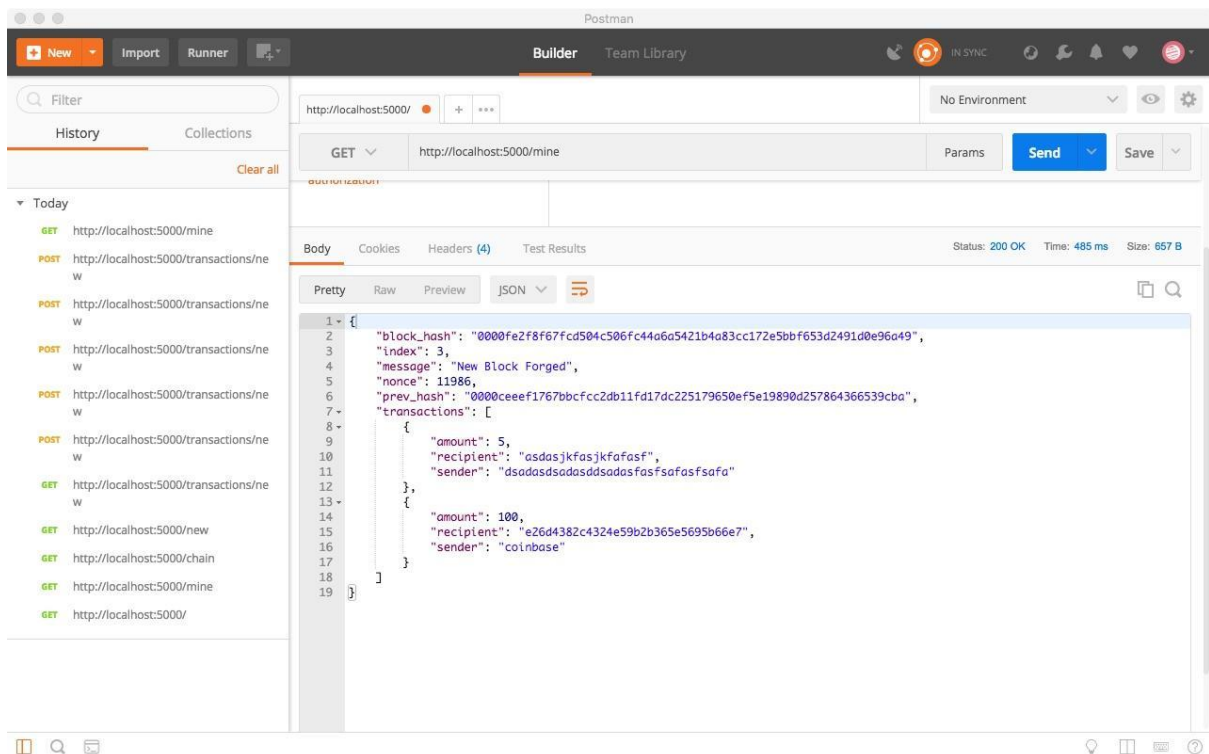
This endpoint above is `/chain`, which returns a response showing the entire blockchain and the length of the blockchain.

Endpoints are like places on a server. People can access it by adding endpoints after the server address and it retrieves information if selecting “Get” and add information from local if selecting “Post”. When people access the endpoints, there could be just returned strings and could also trigger operations. Generally, endpoints are address/directions pointed to some kind of information.

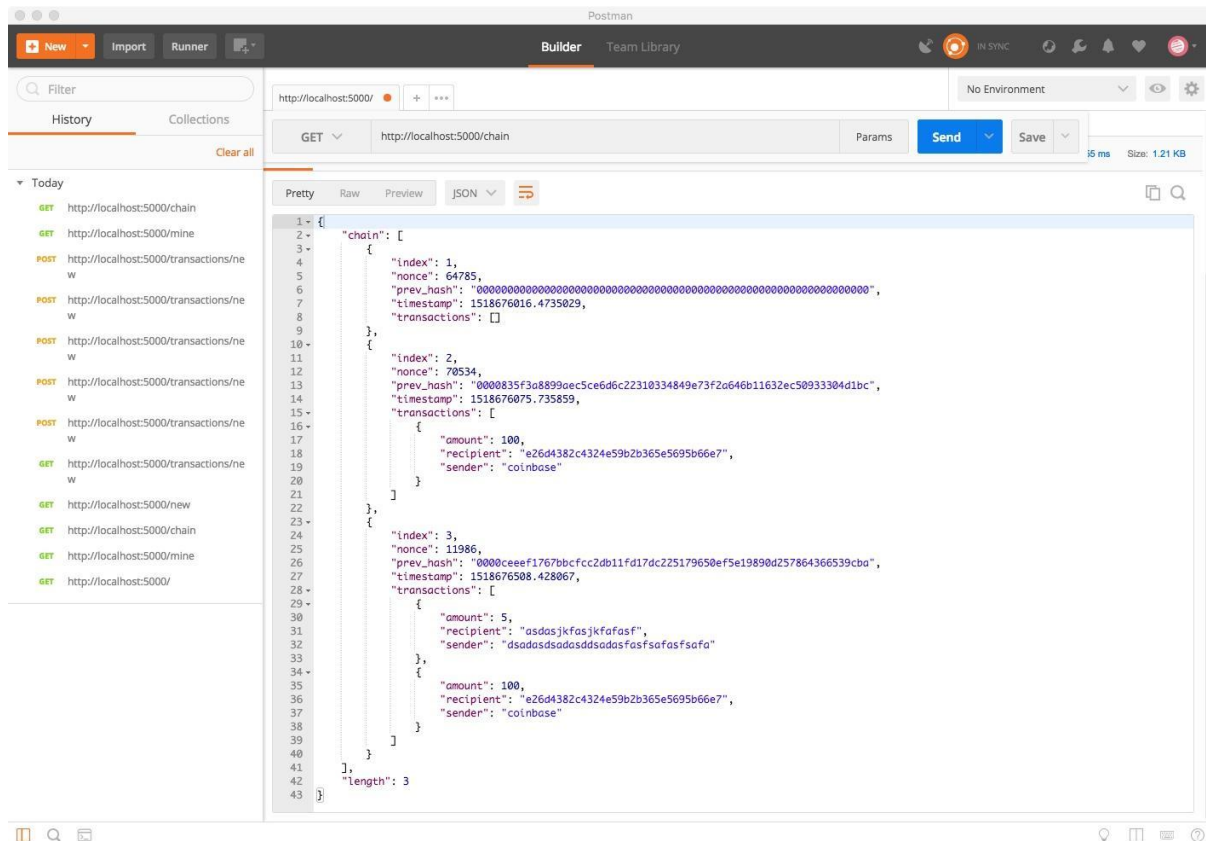
d) Endpoint *transactions/new* creates a transaction the specifies the “sender”, “recipient”, etc and assigns it to the new block that will be mined the next.



And the next miner will mine (using */mine*) the block with the newly added transaction.



The new chain is formed and can be retrieved by `/chain`.



e) When the program tries to detect conflicts and solve them, the following chunk of code checks whether the chain at any one of the neighbors is longer and valid. If so, that chain will be taken as the valid chain and replace the current one of the host. Therefore the conflicts are solved.

```
for node in neighbors:
    rp = requests.get(f'http://{node}/chain')

    if rp.status_code is 200:
        #validation
        length = rp.json()['length']
        chain = rp.json()['chain']

        if length > max_length and self.valid_chain(chain):
            max_length = length
            new_chain = chain
```

The code above calls `valid_chain()` to validate the chains at neighbor nodes. The most important part that analyzes the chain is the following:

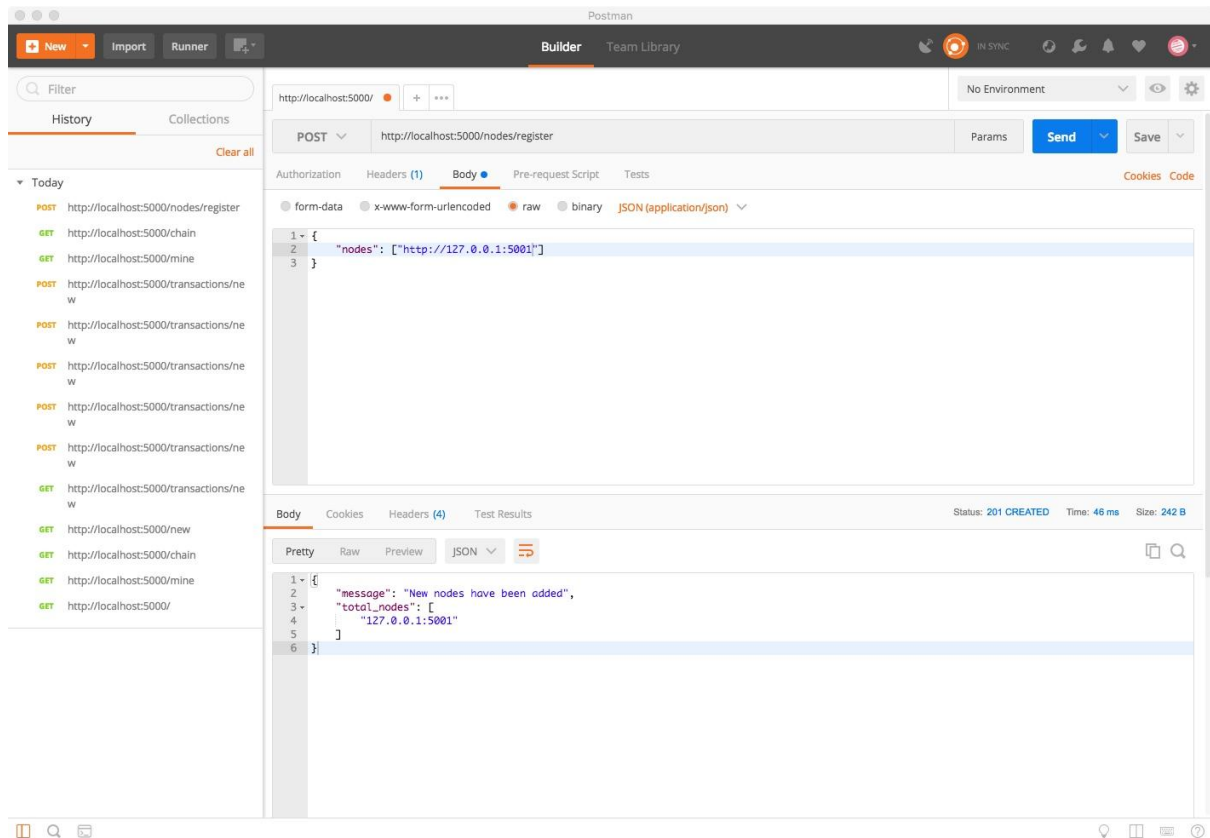
```
if block['prev_hash'] != self.hash(last_block):
    return False

if not self.valid_proof(block['index'], block['timestamp'], block['transactions'], block['nonce'], block['prev_hash']):
    return False
```



The first “if” checks if the block belongs to the chain by checking the hash of the previous block and being checked by the next block. The second “if” checks whether the proof of work is valid by checking the if the hash meets the requirements. If the chain passes two “if”’s, it is to be proved valid. Following is an example of the use of */resolve* (*resolve\_conflicts()*).

Initially port:5001 is registered by port:5000.





Port:5001 then mines 5 times so it now has 6 blocks in its chain while port:5000 has only one(the genesis block). There is a conflict. Port:5001 has the authoritative chain.

The screenshot shows the Postman interface with a GET request to `http://localhost:5001/nodes/resolve`. The response is a JSON array of 6 blocks, each containing an index, nonce, previous hash, timestamp, and a list of transactions. The transactions include an amount of 100 and a recipient address. The final message in the response is "Our chain is authoritative".

```
1  {
2    "index": 4,
3    "nonce": 100820,
4    "prev_hash": "000035766b243a1422bfb8df1e62ddb57226392db3cb18577ea81a7b7b6e198",
5    "timestamp": 1518686499.620955,
6    "transactions": [
7      {
8        "amount": 100,
9        "recipient": "b7263e7675b64908ad27524a893579ce",
10       "sender": "coinbase"
11     }
12   ],
13   "index": 5,
14   "nonce": 168712,
15   "prev_hash": "00009bb8476b001c3fdba2dc6b8d2f4dd31b9d62d5fa18bc80e884e9c0204b60",
16   "timestamp": 1518686530.532413,
17   "transactions": [
18     {
19       "amount": 100,
20       "recipient": "b13209864f7a4e20ae59a1bd8113469",
21       "sender": "coinbase"
22     }
23   ],
24   "index": 6,
25   "nonce": 26132,
26   "prev_hash": "0000dfdb35015ef6e09348c007ba7a92890b248f2e956a04719ec8ea0c42e96c",
27   "timestamp": 1518686735.3580391,
28   "transactions": [
29     {
30       "amount": 100,
31       "recipient": "b13209864f7a4e20ae59a1bd8113469",
32       "sender": "coinbase"
33     }
34   ]
35 },
36 {
37   "message": "Our chain is authoritative"
38 }
```

And the chain at port:5000 is replaced.

The screenshot shows the Postman interface with a GET request to `http://localhost:5000/nodes/resolve`. The response is a JSON object with a message "Our chain was replaced" and a new chain of 4 blocks. The first block is the genesis block, and the subsequent blocks contain transactions with an amount of 100 and a recipient address.

```
1  {
2    "message": "Our chain was replaced",
3    "new_chain": [
4      {
5        "index": 1,
6        "nonce": 167515,
7        "prev_hash": "0000000000000000000000000000000000000000000000000000000000000000",
8        "timestamp": 1518686424.029719,
9        "transactions": []
10     },
11     {
12       "index": 2,
13       "nonce": 9002,
14       "prev_hash": "00005c96c3be5c20eb4e6578b16549b2d148e44176227928957a2c7f9d4ae74f",
15       "timestamp": 1518686495.568933,
16       "transactions": [
17         {
18           "amount": 100,
19           "recipient": "b7263e7675b64908ad27524a893579ce",
20           "sender": "coinbase"
21         }
22       ]
23     },
24     {
25       "index": 3,
26       "nonce": 86436,
27       "prev_hash": "00000d8aa98b3b10db660aac39b08beae6c739330196d0d0448c63c359810c7b",
28       "timestamp": 1518686496.602775,
29       "transactions": [
30         {
31           "amount": 100,
32           "recipient": "b7263e7675b64908ad27524a893579ce",
33           "sender": "coinbase"
34         }
35       ]
36     },
37     {
38       "index": 4,
39       "nonce": 100820,
40       "prev_hash": "000035766b243a1422bfb8df1e62ddb57226392db3cb18577ea81a7b7b6e198",
41       "timestamp": 1518686499.620955,
42       "transactions": [
43         {
44           "amount": 100,
45           "recipient": "b7263e7675b64908ad27524a893579ce",
46           "sender": "coinbase"
47         }
48       ]
49     }
50   ]
51 }
```

**Roles:**

We discussed this question and read the tutorial together.

Jing is mainly in charge of part a, b and c

Zheng is mainly in charge of part d and e.

We wrote the documentation together.