

## **Tutorial – Building a Blockchain**

### **Representing a blockchain**

The tutorial represents the blockchain as a class. The class contains the methods that can be used on the class to accomplish the various tasks required of the chain. The methods described are:

#### *1. Initialisation*

Initialisation entails creating the chain. It creates two empty lists – one to store the blockchain, the other to store transactions which are used to create blocks.

#### *2. New Block Creation*

The `.new_block` method creates a new block, and then adds it to the end of the blockchain. The transactions will be compiled from the list, added to the block, and then the block will be added to the chain, with all the required properties.

#### *3. New Transaction Creation*

A `new_transaction` with all the required information is created and added to the list of transactions.

Using these methods, the blockchain can be created and subsequently manipulated as required. Given that each method is explored in detail, for now, a simple placeholder definition of the class is sufficient, as seen below:

```
class Blockchain(object):
    def __init__(self):
        self.chain = []
        self.current_transactions = []

    def new_block(self):
        # Creates a new Block and adds it to the chain
        pass

    def new_transaction(self):
        # Adds a new transaction to the list of transactions
        pass

    @staticmethod
    def hash(block):
        pass

    @property
    def last_block(self):
        # Returns the last Block in the chain
        pass
```

The final part of the placeholder is a static method which computes the hash of the block, by using sha256 as a hash function, and a property which allows the user to see the last block in the chain. This is useful functionality that is applied later.

## **Structure of a Block**

Every block comprises the following parts:

1. Index – the block identification index.
2. Timestamp – the Unix timestamp of the block's creation/verification
3. Transaction list
4. “Proof” – proof of work, discussed later
5. Hash pointer to the previous block

```
block = {
  'index': 1,
  'timestamp': 1506057125.900785,
  'transactions': [
    {
      'sender': "8527147fe1f5426f9dd545de4b27ee00",
      'recipient': "a77f5cdfa2934df3954a5c7c7da5df1f",
      'amount': 5,
    }
  ],
  'proof': 324984774000,
  'previous_hash': "2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824"
```

The transactions are added to the block using a combination of the `.new_block` and the `.new_transaction` methods. By encoding the `previous_hash` into the block, the previous block is “set in stone”. The idea is that the hash is computed using the static method `hash`, which uses *all* the data in the block. If a block is edited or modified in any way, the hash algorithm will not match, and a subsequent rehash of the edited block will result in a new hash value. However, since the original hash was encoded in the new block, it will be apparent that the edited block has been tampered with, and all the subsequent blocks will contain incorrect hashes<sup>1</sup>.

## **Creating a New Transaction**

The new transaction is one of the “helper methods” in the tutorial. It accomplishes three tasks:

1. It creates a transaction with sender, recipient and amount fields.
2. It adds this transaction to a list of transactions which will make up the next block
3. It then returns the index of the block that will contain the transaction being added – which is the *next block that will be mined*.

---

<sup>1</sup> Conceivably, one could go and rehash *the entire blockchain beyond the tampered block*, but this would be cryptographically difficult. While it may be possible in the case of this simple example, it would not work in a stable system like Bitcoin, which depends on verification of the chain.

```

def new_transaction(self, sender, recipient, amount):
    """
    Creates a new transaction to go into the next mined Block
    :param sender: <str> Address of the Sender
    :param recipient: <str> Address of the Recipient
    :param amount: <int> Amount
    :return: <int> The index of the Block that will hold this transaction
    """

    self.current_transactions.append({
        'sender': sender,
        'recipient': recipient,
        'amount': amount,
    })

    return self.last_block['index'] + 1

```

current\_transactions is the list of transactions that will comprise the next block.

## **Creating a New Block**

Creating a new block has various facets to it. It needs to initialise a block, index it, link it to the predecessor, and add the transactions.

```

def new_block(self, proof, previous_hash=None):
    """
    Create a new Block in the Blockchain
    :param proof: <int> The proof given by the Proof of Work algorithm
    :param previous_hash: (Optional) <str> Hash of previous Block
    :return: <dict> New Block
    """

    block = {
        'index': len(self.chain) + 1,
        'timestamp': time(),
        'transactions': self.current_transactions,
        'proof': proof,
        'previous_hash': previous_hash or self.hash(self.chain[-1]),
    }

    # Reset the current list of transactions
    self.current_transactions = []

    self.chain.append(block)
    return block

```

## **Proof of Work**

The example in the tutorial uses a much-simplified model. While the `valid_proof` static method requires that the first four digits of the hash be 0 – the same as the example in class (the live blockchain at [blockchain.mit.edu](https://blockchain.mit.edu)), the difference lies in the `proof_of_work` algorithm. The proof of work in the tutorial calculates the proof based on the proof of the last block only.

```
def proof_of_work(self, last_proof):
    """
    Simple Proof of Work Algorithm:
    - Find a number p' such that hash(pp') contains leading 4 zeroes, where p is the
      previous p'
    - p is the previous proof, and p' is the new proof
    :param last_proof: <int>
    :return: <int>
    """

    proof = 0
    while self.valid_proof(last_proof, proof) is False:
        proof += 1

    return proof

    @staticmethod
    def valid_proof(last_proof, proof):
        """
        Validates the Proof: Does hash(last_proof, proof) contain 4 leading zeroes?
        :param last_proof: <int> Previous Proof
        :param proof: <int> Current Proof
        :return: <bool> True if correct, False if not.
        """

        guess = f'{last_proof}{proof}'.encode()
        guess_hash = hashlib.sha256(guess).hexdigest()
        return guess_hash[:4] == "0000"
```

Essentially, the algorithm hashes together the proof of the last block together with the proof of the current block, and then checks whether the first four digits of the resulting hash are 0. If they are, then the proof is returned, if not, the proof of the current block is incremented by 1, and so on, until the condition is met.

The MIT website, however, hashes together the data in the block, the *hash of the previous block*, and the proof (which is referred to as ‘nonce’), and then checks the resulting hash to see whether the first four digits are 0 or not.

## **Python Flask Framework**

`/transactions/new`

Creates a new transaction and adds to the next block to be mined. Essentially this endpoint calls the `new_transaction` method in python. It is a POST method, in which the transaction is posted to the client.

`/mine`

Mines the next block at the given node. Essentially calculates the proof brute force such that a hash condition is met. This is a GET method – it receives information from the client.

`/chain`

Returns the chain as it currently stands on the node in question. This is also a GET method.

### *Endpoints:*

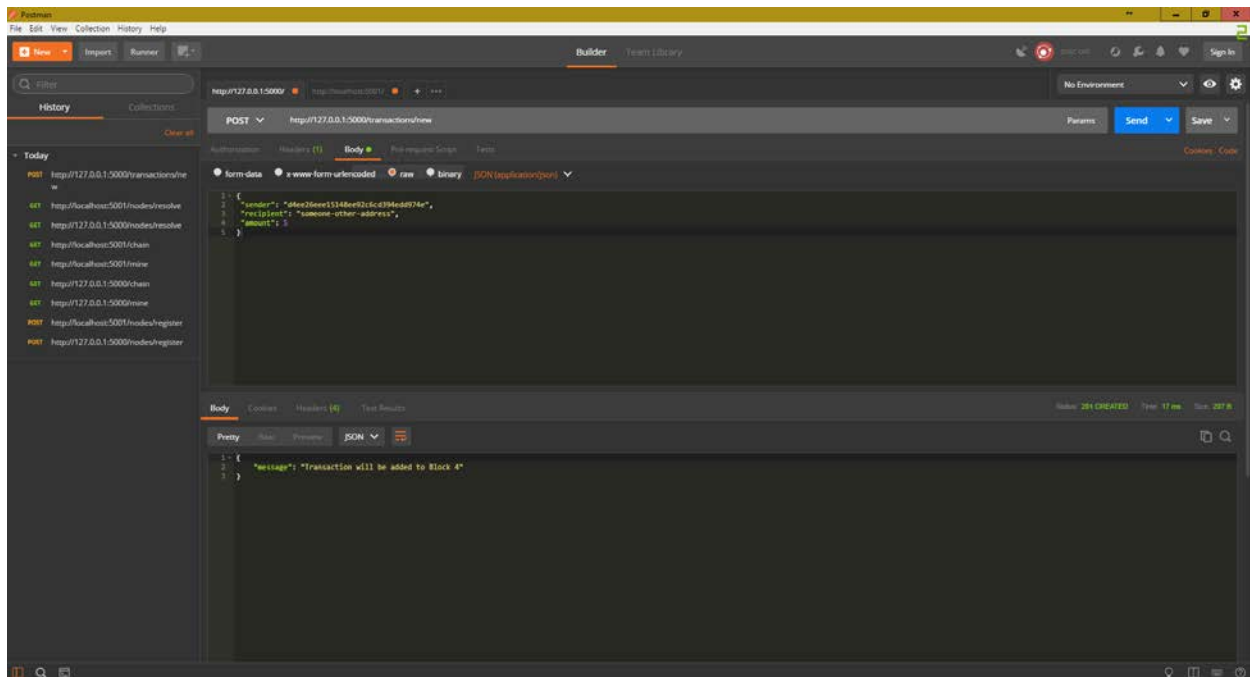
Endpoints are the unique urls which are used as pointers. Here, the endpoints point to Python functions, which are requested by the HTTP client (Postman). So, when the method `/mine` is called, it acts as a GET request from the endpoint, receives the function and executes it to mine the block.

## **Implementation**

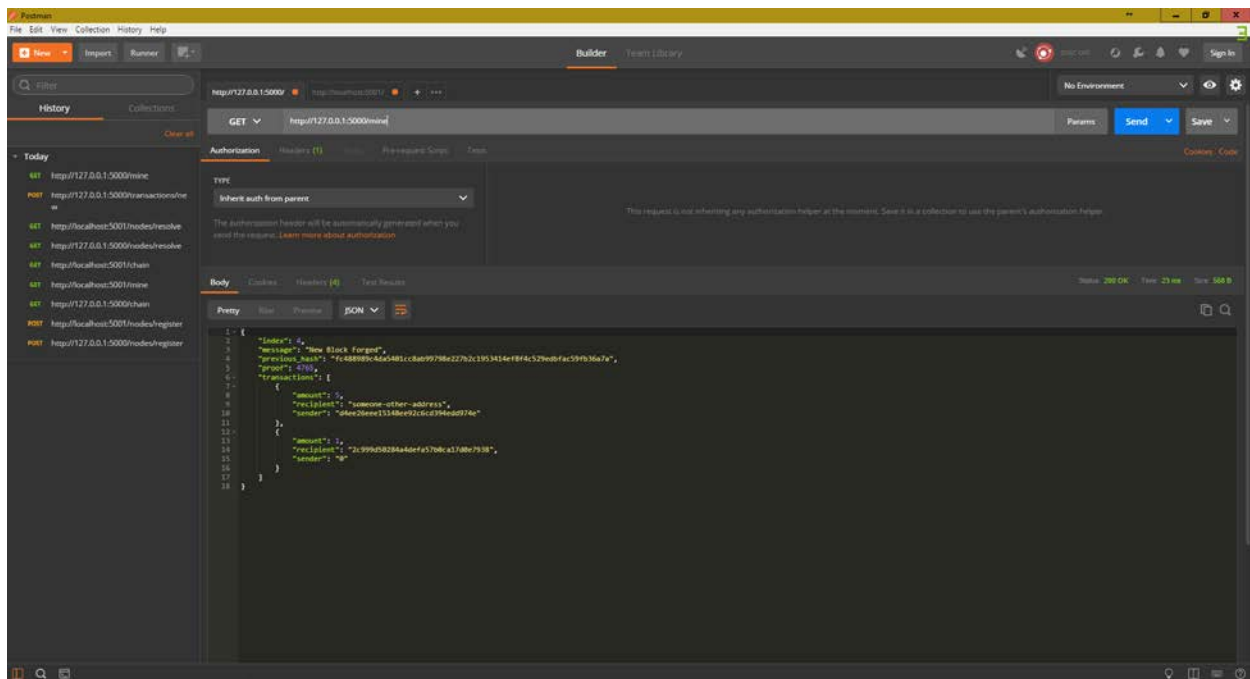
The implementation for the next sections – using Postman to GET and POST requests to the chain, was done by initialising the node on a computer using Spyder from the Anaconda Suite. Spyder allows for multiple consoles, which allowed for simultaneous execution of two separate files – simulating two nodes on ports 5000 and 5001 of the `localhost`.

It is to be noted that this entire process, which combined Spyder and Postman could be replaced by a simple terminal prompt and `cURL`. Similarly, the nodes could be extended to any number of machines, provided that they are on the same network.

## Posting a Transaction



## Mining a Block



## Calling for the entire blockchain

The screenshot shows the Postman API client interface. The top bar includes the Postman logo, a search bar, and tabs for 'New', 'Import', 'Runner', and 'Builder'. The 'Builder' tab is active, showing a GET request to the endpoint `http://127.0.0.1:5000/chain`. The response is displayed in the 'Pretty' view, showing a JSON array of four blockchain blocks. The left sidebar shows a 'History' tab with a list of recent requests.

**Request:**

```
GET http://127.0.0.1:5000/chain
```

**Response (JSON):**

```
{
  "chain": [
    {
      "index": 1,
      "previous_hash": "1",
      "timestamp": 1518788777.5438347,
      "transactions": [
        {
          "amount": 1,
          "recipient": "3c9995d8846defa579dc17dbcf336",
          "sender": "g"
        }
      ]
    },
    {
      "index": 2,
      "previous_hash": "2b75c3c21264805bcbfa2224fcd9ba723db8fffa07e467c6d1af924edf256",
      "timestamp": 1518788777.5438347,
      "transactions": [
        {
          "amount": 1,
          "recipient": "3c9995d8846defa579dc17dbcf336",
          "sender": "g"
        }
      ]
    },
    {
      "index": 3,
      "previous_hash": "80d9780fc51ba992aa9f4bcafc83cbcd54f6dd8c713cd58f913acdef336",
      "timestamp": 1518788777.5438347,
      "transactions": [
        {
          "amount": 1,
          "recipient": "3c9995d8846defa579dc17dbcf336",
          "sender": "g"
        }
      ]
    },
    {
      "index": 4,
      "previous_hash": "fc48808b4da5481ccda99786c227b3c1953434ef8f4c529edfac59fb36a3",
      "timestamp": 1518788777.5438347,
      "transactions": [
        {
          "amount": 1,
          "recipient": "someone-else-address",
          "sender": "3c9995d8846defa579dc17dbcf336"
        },
        {
          "amount": 1,
          "recipient": "3c9995d8846defa579dc17dbcf336",
          "sender": "g"
        }
      ]
    }
  ],
  "length": 4
}
```

## Consensus

The algorithm is incredibly basic in that it only searches for one parameter, and based on a binary result, proceeds with resolution of conflicts. The method is `resolve_conflicts`:

```
def resolve_conflicts(self):
    """
    This is our consensus algorithm, it resolves conflicts
    by replacing our chain with the longest one in the network.

    :return: True if our chain was replaced, False if not
    """
    # pdb.set_trace()
    neighbours = self.nodes
    new_chain = None

    # We're only looking for chains longer than ours
    max_length = len(self.chain)

    # Grab and verify the chains from all the nodes in our network
    for node in neighbours:
        response = requests.get(f'http://{node}/chain')

        if response.status_code == 200:
            length = response.json()['length']
            chain = response.json()['chain']

            # Check if the length is longer and the chain is valid
            if length > max_length and self.valid_chain(chain):
                max_length = length
                new_chain = chain

    # Replace our chain if we discovered a new, valid chain longer than ours
    if new_chain:
        self.chain = new_chain
        return True

    return False
```

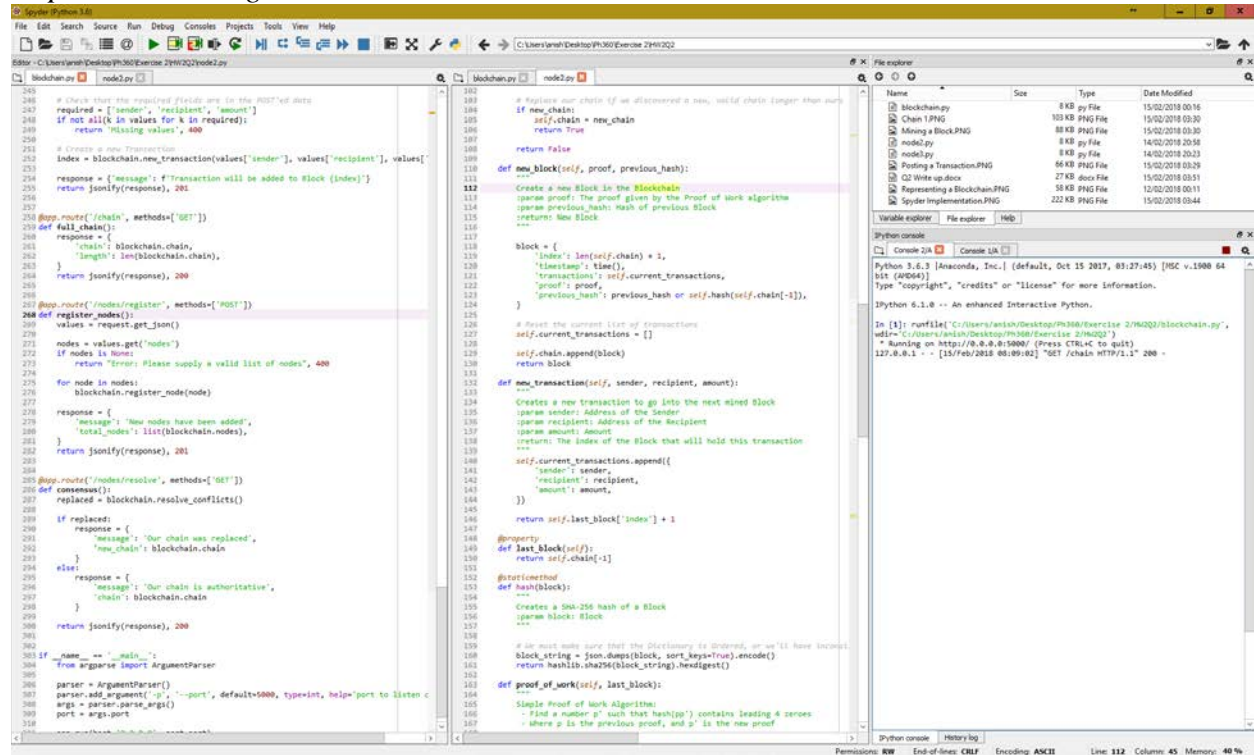
The method first finds the length of the chain at the node where the method is called. It then compares the length of the chain of the nodes that are its neighbours. After catching the longest chain, it checks validity and returns true or false based on the outcome.

**NOTE:** It was found that there was an issue in the `valid_chain` method, where the original code called for the `previous_hash` of the `last_block`, instead of the `current block`. This is currently an open issue on the GitHub ([Issue #66](#)) and by editing `last_block` to `block`, the `resolve_conflicts` method did not resolve incorrectly, defaulting to authoritative no matter the length of the chain. We used a `pdb` trace to verify this issue independently.

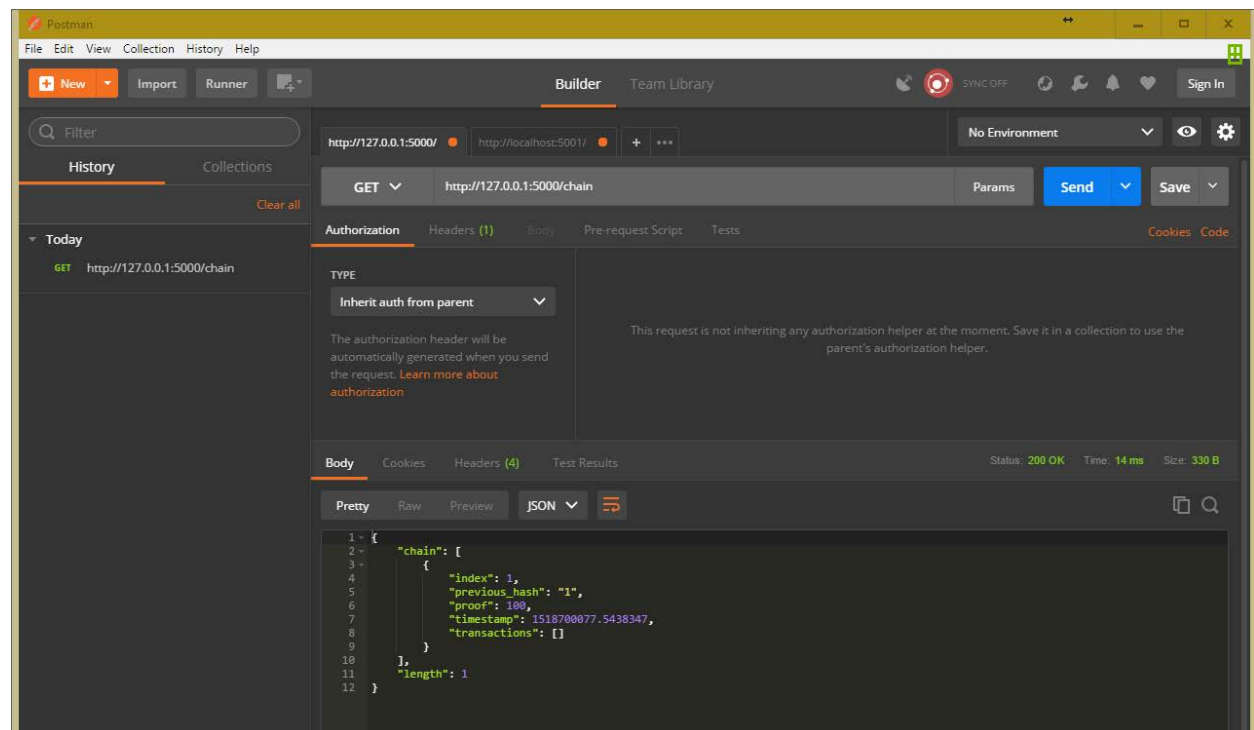


## Consensus, as seen in code

### Step 1 – Initialising both nodes

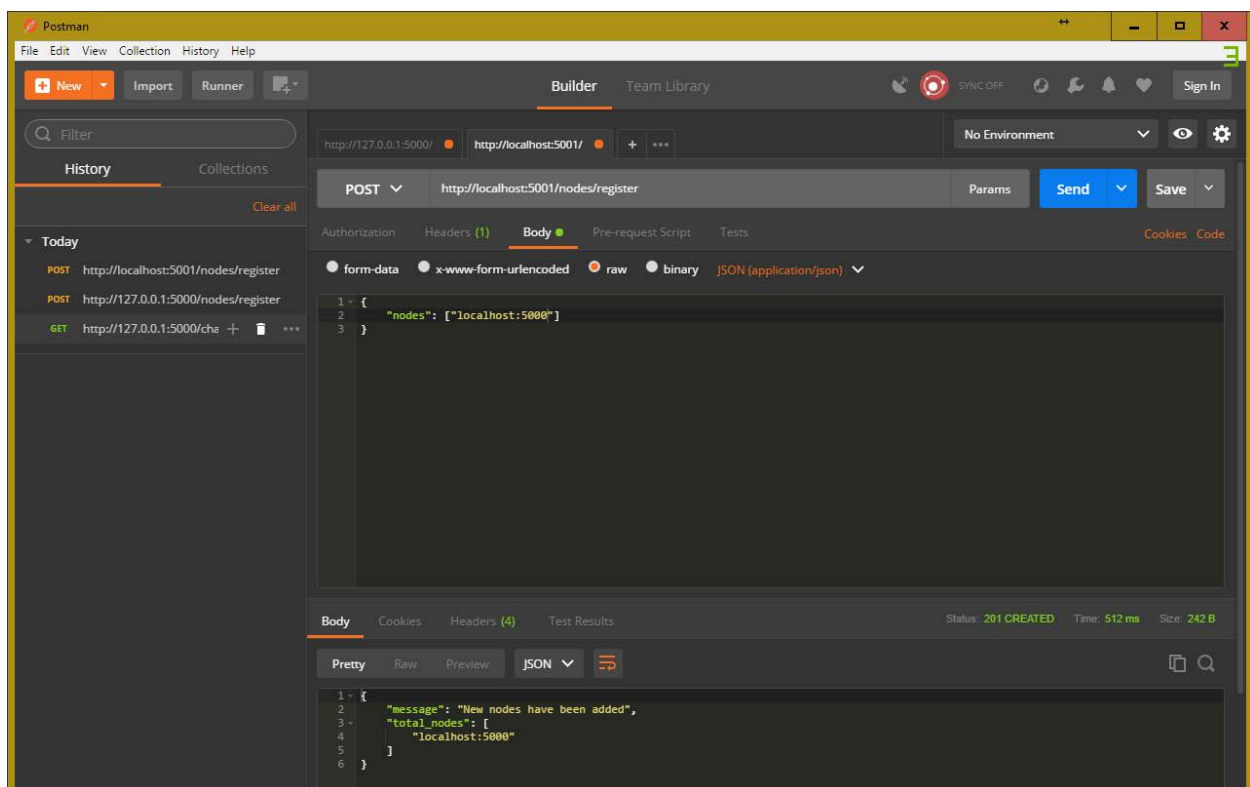
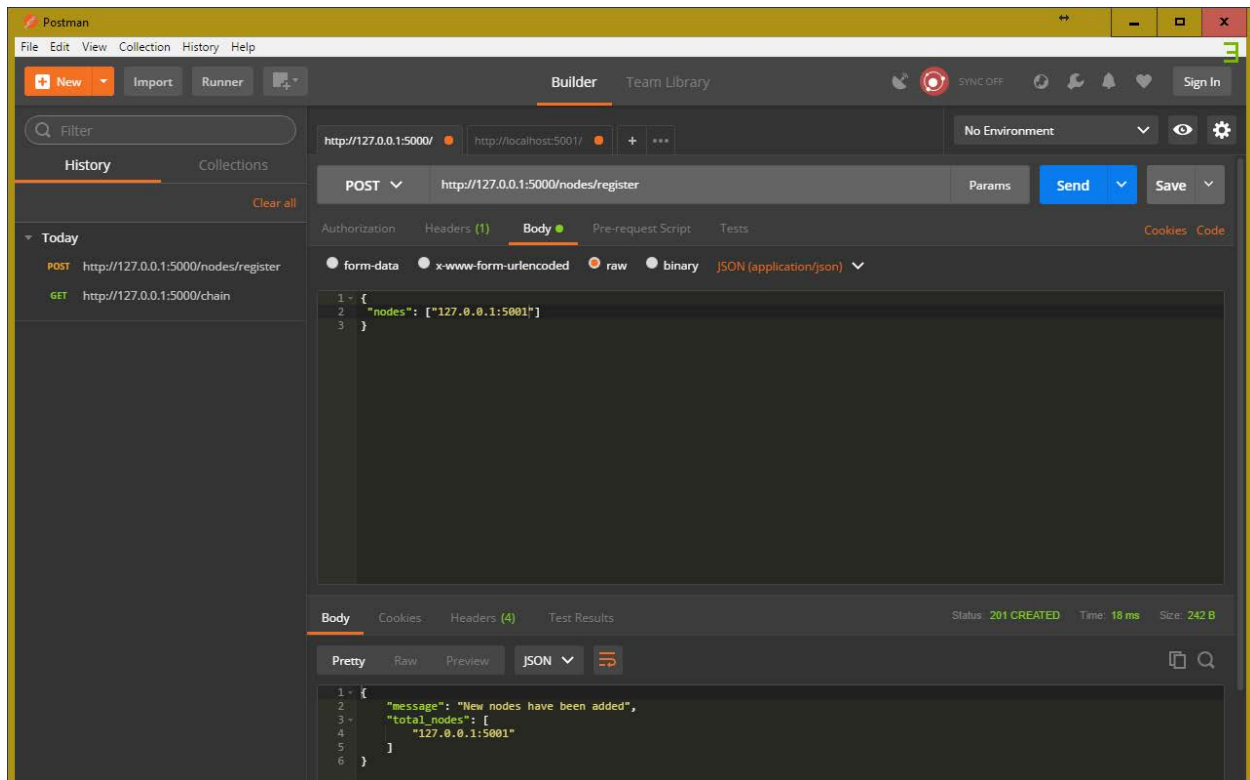


```
245 # Check that the required fields are in the POST'ed data
246 required = ['sender', 'recipient', 'amount']
247 if not all(k in values for k in required):
248     return "Missing values", 400
249
250 # Create a new transaction
251 index = blockchain.new_transaction(values['sender'], values['recipient'], values['amount'])
252 response = {'message': f'Transaction will be added to block {index}'}
253 return jsonify(response), 201
254
255 @app.route('/chain', methods=['GET'])
256 def get_chain():
257     response = {
258         'chain': blockchain.chain,
259         'length': len(blockchain.chain),
260     }
261     return jsonify(response), 200
262
263 @app.route('/nodes/register', methods=['POST'])
264 def register_nodes():
265     values = request.get_json()
266     if nodes is None:
267         return "Error: Please supply a valid list of nodes", 400
268     for node in nodes:
269         blockchain.register_node(node)
270     response = {
271         'message': 'New nodes have been added',
272         'total_nodes': len(blockchain.nodes),
273     }
274     return jsonify(response), 201
275
276 @app.route('/nodes/resolve', methods=['GET'])
277 def consensus():
278     replaced = blockchain.resolve_conflicts()
279     if replaced:
280         response = {
281             'message': 'Our chain was replaced',
282             'new_chain': blockchain.chain,
283         }
284     else:
285         response = {
286             'message': 'Our chain is authoritative',
287             'chain': blockchain.chain,
288         }
289     return jsonify(response), 200
290
291 if __name__ == '__main__':
292     from argparse import ArgumentParser
293     parser = ArgumentParser()
294     parser.add_argument('-p', '--port', default=5000, type=int, help='port to listen on')
295     args = parser.parse_args()
296     port = args.port
297
298     # Register our chain if we discovered a new, valid chain longer than our own
299     if new_chain:
300         self.chain = new_chain
301         return True
302     return False
303
304 def new_block(self, proof, previous_hash):
305     """
306     Create a new block in the Blockchain
307     :param proof: The proof given by the Proof of Work algorithm
308     :param previous_hash: Hash of previous Block
309     :return: New Block
310     """
311     block = {
312         'index': len(self.chain) + 1,
313         'timestamp': time(),
314         'transactions': self.current_transactions,
315         'proof': proof,
316         'previous_hash': previous_hash or self.hash(self.chain[-1]),
317     }
318     # Reset the current list of transactions
319     self.current_transactions = []
320     self.chain.append(block)
321     return block
322
323 def new_transaction(self, sender, recipient, amount):
324     """
325     Create a new transaction to go into the next mined block
326     :param sender: Address of the Sender
327     :param recipient: Address of the Recipient
328     :param amount: Amount
329     :return: The index of the Block that will hold this transaction
330     """
331     self.current_transactions.append(
332         {
333             'sender': sender,
334             'recipient': recipient,
335             'amount': amount,
336         }
337     )
338     return self.last_block['index'] + 1
339
340 @property
341 def last_block(self):
342     return self.chain[-1]
343
344 @staticmethod
345 def hash(block):
346     """
347     Creates a SHA-256 hash of a Block
348     :param block: Block
349     """
350     # We must make sure that the Dictionary is ordered, or we'll have inconsistency
351     block_string = json.dumps(block, sort_keys=True).encode()
352     return hashlib.sha256(block_string).hexdigest()
353
354 def proof_of_work(self, last_block):
355     """
356     Simple Proof of Work Algorithm:
357     - Find a number p' such that hash(pp') contains leading 4 zeroes
358     - where p is the previous proof, and p' is the new proof
359     """
360     p = last_block['proof']
361     p_prime = p + 1
362     while not self.valid_proof(p_prime, last_block['previous_hash']):
363         p_prime = p + 1
364     p = p_prime
365     return p
```



The two nodes are simultaneously running in separate console windows 1(A) and 2(A). In Postman, the two tabs are used to interact with them independently.

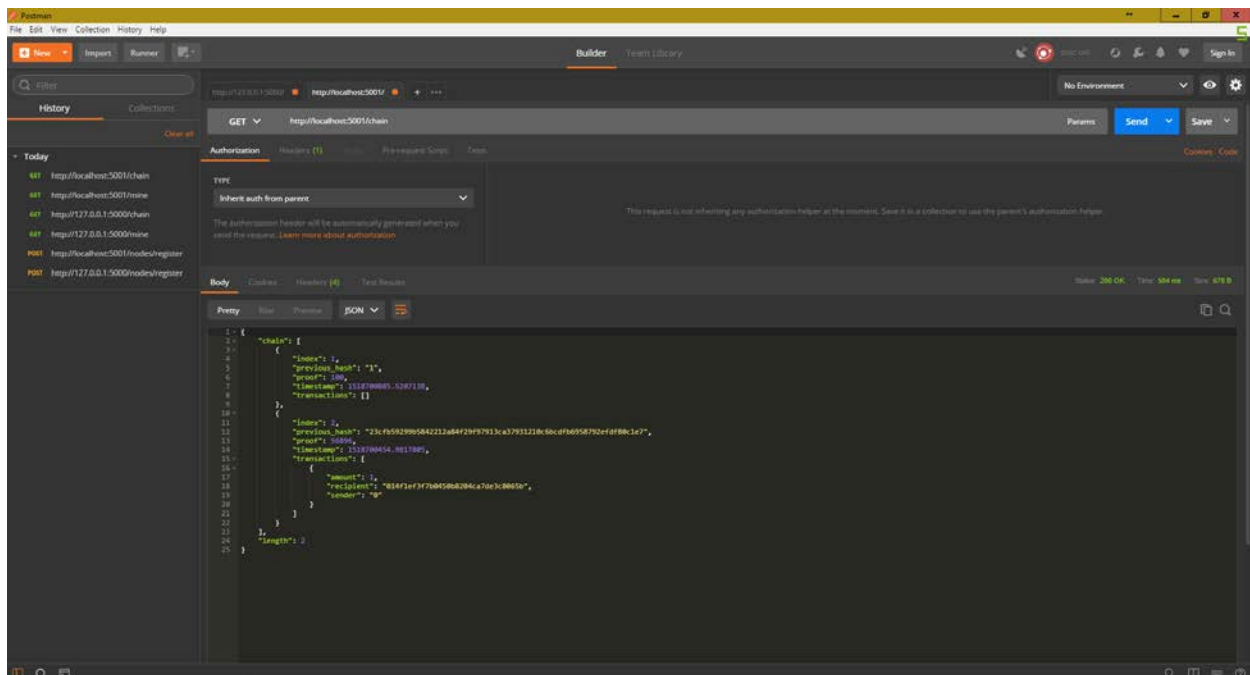
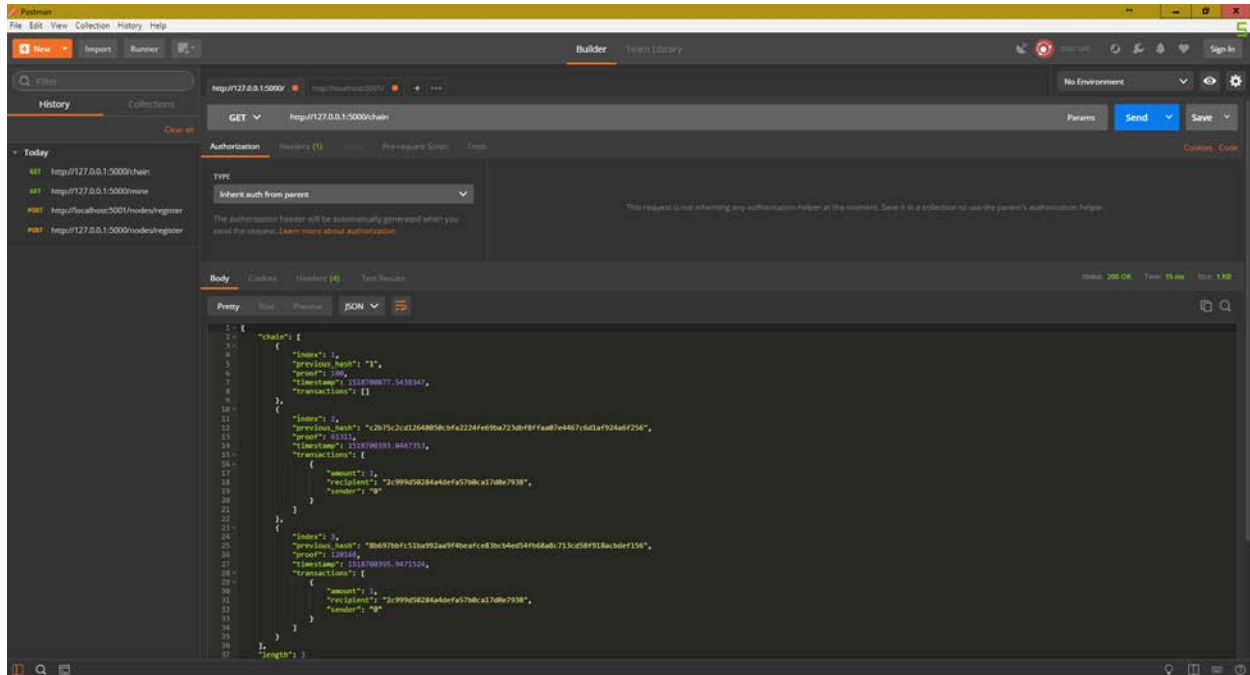
## Step 2 – Node Registration



Both nodes are made aware of the other's presence using the `/nodes/register` method.

### Step 3 – Mining blocks

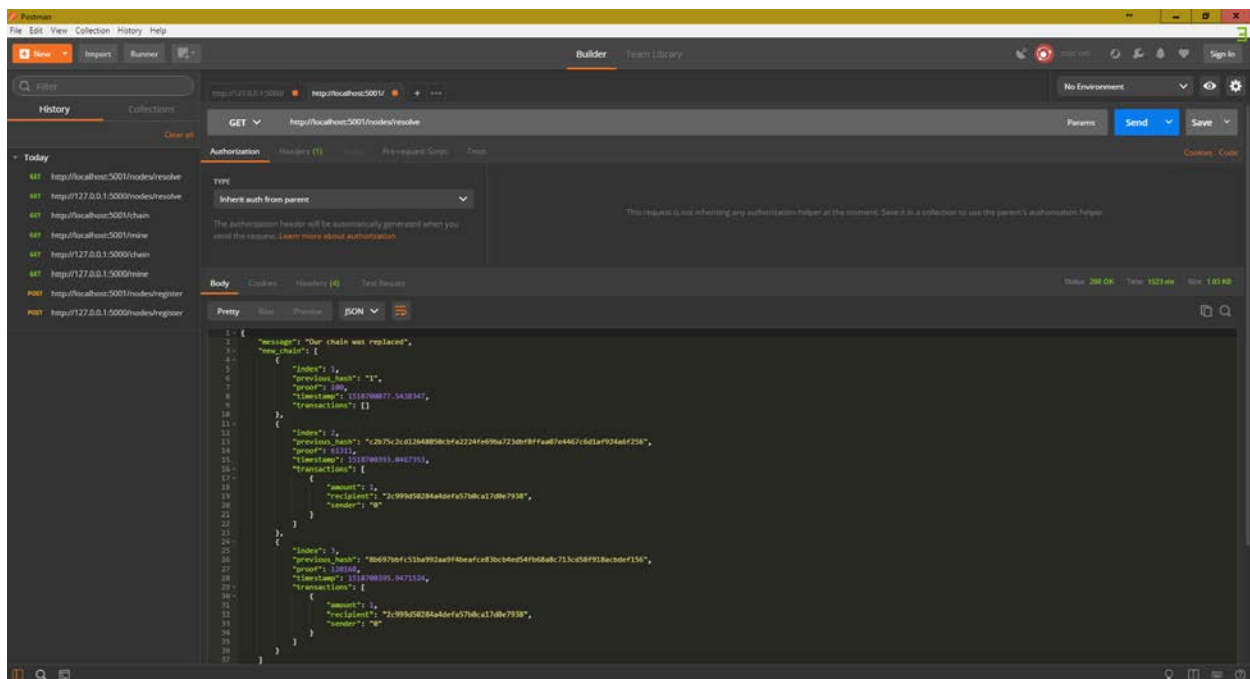
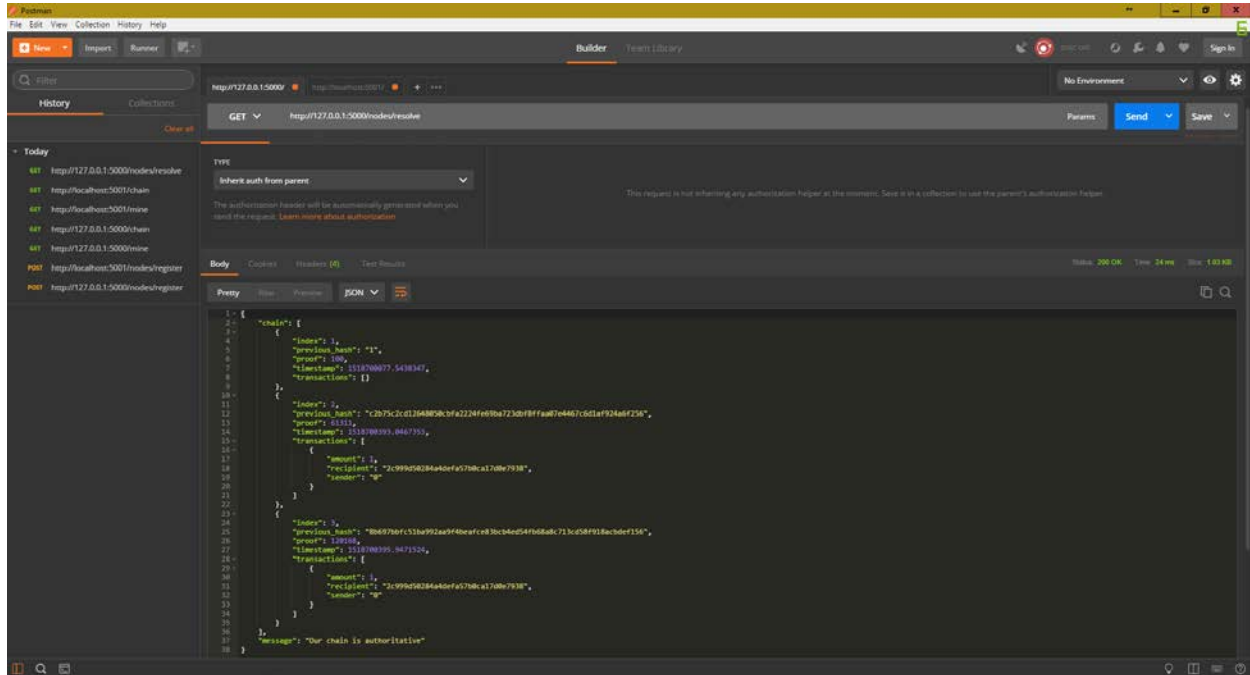
Two additional blocks are mined on node 1 and one additional block on node 2. This way, there is a conflict in which node has the longer chain.



The idea here is that once the conflict is detected, the shorter chain will be replaced by the longer, while the longer stays the same, unifying the network.

## Step 4 – Consensus Resolution

The `/nodes/resolve` method is used to detect and resolve conflict on both chains. On applying to node 1 (PORT 5000), it changes little, the existing chain is declared to be authoritative. However, on node 2 (PORT 5001), the existing chain is replaced, and simultaneously checked for validity.



This is the end console appearance for each node in Spyder.

*Node 1:*

```
IPython console
Console 2/A Console 1/A
Python 3.6.3 |Anaconda, Inc.| (default, Oct 15 2017, 03:27:45) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 6.1.0 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/anish/Desktop/Ph360/Exercise 2/HW2Q2/blockchain.py',
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [15/Feb/2018 08:09:02] "GET /chain HTTP/1.1" 200 -
127.0.0.1 - - [15/Feb/2018 08:10:48] "POST /nodes/register HTTP/1.1" 201 -
127.0.0.1 - - [15/Feb/2018 08:13:13] "GET /mine HTTP/1.1" 200 -
127.0.0.1 - - [15/Feb/2018 08:13:15] "GET /mine HTTP/1.1" 200 -
127.0.0.1 - - [15/Feb/2018 08:13:26] "GET /chain HTTP/1.1" 200 -
127.0.0.1 - - [15/Feb/2018 08:14:53] "GET /nodes/resolve HTTP/1.1" 200 -
127.0.0.1 - - [15/Feb/2018 08:23:45] "GET /chain HTTP/1.1" 200 -
```

*Node 2:*

```
IPython console
Console 2/A Console 1/A
Python 3.6.3 |Anaconda, Inc.| (default, Oct 15 2017, 03:27:45) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 6.1.0 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/anish/Desktop/Ph360/Exercise 2/HW2Q2/node2.py',
* Running on http://0.0.0.0:5001/ (Press CTRL+C to quit)
127.0.0.1 - - [15/Feb/2018 08:12:06] "POST /nodes/register HTTP/1.1" 201 -
127.0.0.1 - - [15/Feb/2018 08:14:14] "GET /mine HTTP/1.1" 200 -
127.0.0.1 - - [15/Feb/2018 08:14:21] "GET /chain HTTP/1.1" 200 -
127.0.0.1 - - [15/Feb/2018 08:14:53] "GET /chain HTTP/1.1" 200 -
127.0.0.1 - - [15/Feb/2018 08:23:45] "GET /nodes/resolve HTTP/1.1" 200 -
{'index': 1, 'previous_hash': '1', 'proof': 100, 'timestamp':
151870077.5438347, 'transactions': []}
{'index': 2, 'previous_hash':
'c2b75c2cd12648050cbfa2224fe69ba723dbf8ffaa07e4467c6d1af924a6f256', 'proof':
61311, 'timestamp': 1518700393.0467353, 'transactions': [{'amount': 1,
'recipient': '2c999d50284a4defa57b0ca17d0e7938', 'sender': '0'}]}}
-----
{'index': 2, 'previous_hash':
'c2b75c2cd12648050cbfa2224fe69ba723dbf8ffaa07e4467c6d1af924a6f256', 'proof':
61311, 'timestamp': 1518700393.0467353, 'transactions': [{'amount': 1,
'recipient': '2c999d50284a4defa57b0ca17d0e7938', 'sender': '0'}]}}
{'index': 3, 'previous_hash':
'8b697bbfc51ba992aa9f4beafce83bcb4ed54fb68a8c713cd58f918acbedf156', 'proof':
120168, 'timestamp': 1518700395.9471524, 'transactions': [{'amount': 1,
'recipient': '2c999d50284a4defa57b0ca17d0e7938', 'sender': '0'}]}}
-----
```