Daniel van Flymen Follow
Blockchain Engineer in NYC | South African ZA | https://electron.network | http://dvf.nyc
Sep 24, 2017 · 9 min read

Learn Blockchains by Building One

The fastest way to learn how Blockchains work is to build one

You're here because, like me, you're psyched about the rise of Cryptocurrencies. And you want to know how Blockchains work—the fundamental technology behind them.

But understanding Blockchains isn't easy—or at least wasn't for me. I trudged through dense videos, followed porous tutorials, and dealt with the amplified frustration of too few examples.

I like learning by doing. It forces me to deal with the subject matter at a code level, which gets it sticking. If you do the same, at the end of this

guide you'll have a functioning Blockchain with a solid grasp of how they work.

Before you get started...

Remember that a blockchain is an *immutable, sequential* chain of records called Blocks. They can contain transactions, files or any data you like, really. But the important thing is that they're *chained* together using *hashes*.

If you aren't sure what a hash is, here's an explanation.

Who is this guide aimed at? You should be comfy reading and writing some basic Python, as well as have some understanding of how HTTP requests work, since we'll be talking to our Blockchain over HTTP.

What do I need? Make sure that Python 3.6+ (along with `pip`) is installed. You'll also need to install Flask and the wonderful Requests library:

```
pip install Flask==0.12.2 requests==2.18.4
```

Oh, you'll also need an HTTP Client, like Postman or cURL. But anything will do.

Where's the final code? The source code is available here.

. . .

Step 1: Building a Blockchain

Open up your favourite text editor or IDE, personally I ♡ PyCharm. Create a new file, called `blockchain.py`. We'll only use a single file, but if you get lost, you can always refer to the source code.

Representing a Blockchain

We'll create a `Blockchain` class whose constructor creates an initial empty list (to store our blockchain), and another to store transactions.

Here's the blueprint for our class:

```
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 transaction
        pass

    @staticmethod
    def hash(block):
```

Blueprint of our Blockchain Class

Our `Blockchain` class is responsible for managing the chain. It will store transactions and have some helper methods for adding new blocks to the chain. Let's start fleshing out some methods.

What does a Block look like?

Each Block has an *index*, a *timestamp* (in Unix time), a *list of transactions*, a *proof* (more on that later), and the *hash of the previous Block*.

Here's an example of what a single Block looks like:

```
block = {
    'index': 1,
    'timestamp': 1506057125.900785,
    'transactions': [
        {
            'sender': "8527147fe1f5426f9dd545de4b27ee00",
            'recipient': "a77f5cdfa2934df3954a5c7c7da5df1f"
            'amount': 5,
        }
```

Example of a Block in our Blockchain

At this point, the idea of a *chain* should be apparent—each new block contains within itself, the hash of the previous Block. **This is crucial because it's what gives blockchains immutability:** If an attacker corrupted an earlier Block in the chain then ***all*** subsequent blocks will contain incorrect hashes.

Does this make sense? If it doesn't, take some time to let it sink in—it's the core idea behind blockchains.

Adding Transactions to a Block

We'll need a way of adding transactions to a Block. Our `new_transaction()` method is responsible for this, and it's pretty straight-forward:

```
class Blockchain(object):
    ...

    def new_transaction(self, sender, recipient, amount):
        """
        Creates a new transaction to go into the next mined

        :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 hol
        """

        self.current_transactions.append({
```

After `new_transaction()` adds a transaction to the list, it returns the *index* of the block which the transaction will be added to—*the next one to be mined.* This will be useful later on, to the user submitting the transaction.

Creating new Blocks

When our `Blockchain` is instantiated we'll need to seed it with a *genesis* block—a block with no predecessors. We'll also need to add a *"proof"* to our genesis block which is the result of mining (or proof of work). We'll talk more about mining later.

In addition to creating the *genesis* block in our constructor, we'll also flesh out the methods for `new_block()` , `new_transaction()` and `hash()` :

```python
import hashlib
import json
from time import time


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

        # Create the genesis block
        self.new_block(previous_hash=1, proof=100)

    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
        :param previous_hash: (Optional) <str> Hash of prev
        :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(sel
        }

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

        self.chain.append(block)
        return block

    def new_transaction(self, sender, recipient, amount):
        """
        Creates a new transaction to go into the next mined

        :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 hol
        """
```

The above should be straight-forward—I've added some comments and *docstrings* to help keep it clear. We're almost done with representing our blockchain. But at this point, you must be wondering how new blocks are created, forged or mined.

Understanding Proof of Work

A Proof of Work algorithm (PoW) is how new Blocks are created or *mined* on the blockchain. The goal of PoW is to discover a number which solves a problem. The number must be **difficult to find but easy to verify**—computationally speaking—by anyone on the network. This is the core idea behind Proof of Work.

We'll look at a very simple example to help this sink in.

Let's decide that the *hash* of some integer `x` multiplied by another `y` must end in `0`. So, `hash(x * y) = ac23dc...0`. And for this simplified example, let's fix `x = 5`. Implementing this in Python:

```
from hashlib import sha256

x = 5
y = 0  # We don't know what y should be yet...

while sha256(f'{x*y}'.encode()).hexdigest()[-1] != "0":
    y += 1

print(f'The solution is y = {y}')
```

The solution here is `y = 21`. Since, the produced hash ends in `0`:

```
hash(5 * 21) = 1253e9373e...5e3600155e860
```

In Bitcoin, the Proof of Work algorithm is called *Hashcash*. And it's not too different from our basic example above. It's the algorithm that miners race to solve in order to create a new block. In general, the difficulty is determined by the number of characters searched for in a string. The miners are then rewarded for their solution by receiving a coin—in a transaction.

The network is able to *easily* verify their solution.

Implementing basic Proof of Work

Let's implement a similar algorithm for our blockchain. Our rule will be similar to the example above:

> Find a number *p* that when hashed with the previous block's solution a hash with 4 leading `0` s is produced.

```python
import hashlib
import json

from time import time
from uuid import uuid4


class Blockchain(object):
    ...

    def proof_of_work(self, last_proof):
        """
        Simple Proof of Work Algorithm:
         - Find a number p' such that hash(pp') contains le
         - 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

```

To adjust the difficulty of the algorithm, we could modify the number of leading zeroes. But 4 is sufficient. You'll find out that the addition of a single leading zero makes a mammoth difference to the time required to find a solution.

Our class is almost complete and we're ready to begin interacting with it using HTTP requests.

. . .

Step 2: Our Blockchain as an API

We're going to use the Python Flask Framework. It's a micro-framework and it makes it easy to map endpoints to Python functions. This allows us talk to our blockchain over the web using HTTP requests.

We'll create three methods:

- `/transactions/new` to create a new transaction to a block
- `/mine` to tell our server to mine a new block.
- `/chain` to return the full Blockchain.

Setting up Flask

Our "server" will form a single node in our blockchain network. Let's create some boilerplate code:

```
1  import hashlib
2  import json
3  from textwrap import dedent
4  from time import time
5  from uuid import uuid4
6
7  from flask import Flask
8
9
10 class Blockchain(object):
11     ...
12
13
14 # Instantiate our Node
15 app = Flask(__name__)
16
17 # Generate a globally unique address for this node
18 node_identifier = str(uuid4()).replace('-', '')
19
20 # Instantiate the Blockchain
21 blockchain = Blockchain()
22
23
24 @app.route('/mine', methods=['GET'])
25 def mine():
26     return "We'll mine a new Block"
27
```

A brief explanation of what we've added above:

- **Line 15:** Instantiates our Node. Read more about Flask here.
- **Line 18:** Create a random name for our node.
- **Line 21:** Instantiate our `Blockchain` class.
- **Line 24–26:** Create the `/mine` endpoint, which is a `GET` request.
- **Line 28–30:** Create the `/transactions/new` endpoint, which is a `POST` request, since we'll be sending data to it.
- **Line 32–38:** Create the `/chain` endpoint, which returns the full Blockchain.

- **Line 40–41:** Runs the server on port 5000.

The Transactions Endpoint

This is what the request for a transaction will look like. It's what the user sends to the server:

```
{
 "sender": "my address",
 "recipient": "someone else's address",
 "amount": 5
}
```

Since we already have our class method for adding transactions to a block, the rest is easy. Let's write the function for adding transactions:

```
1  import hashlib
2  import json
3  from textwrap import dedent
4  from time import time
5  from uuid import uuid4
6
7  from flask import Flask, jsonify, request
8
9  ...
10
11 @app.route('/transactions/new', methods=['POST'])
12 def new_transaction():
13     values = request.get_json()
14
15     # Check that the required fields are in the POST'ed dat
16     required = ['sender', 'recipient', 'amount']
```

A method for creating Transactions

The Mining Endpoint

Our mining endpoint is where the magic happens, and it's easy. It has to do three things:

1. Calculate the Proof of Work

2. Reward the miner (us) by adding a transaction granting us 1 coin
3. Forge the new Block by adding it to the chain

```
import hashlib
import json

from time import time
from uuid import uuid4

from flask import Flask, jsonify, request

...

@app.route('/mine', methods=['GET'])
def mine():
    # We run the proof of work algorithm to get the next pr
    last_block = blockchain.last_block
    last_proof = last_block['proof']
    proof = blockchain.proof_of_work(last_proof)

    # We must receive a reward for finding the proof.
    # The sender is "0" to signify that this node has mined
    blockchain.new_transaction(
        sender="0",
        recipient=node_identifier,
        amount=1,
    )
```

Note that the recipient of the mined block is the address of our node. And most of what we've done here is just interact with the methods on our Blockchain class. At this point, we're done, and can start interacting with our blockchain.

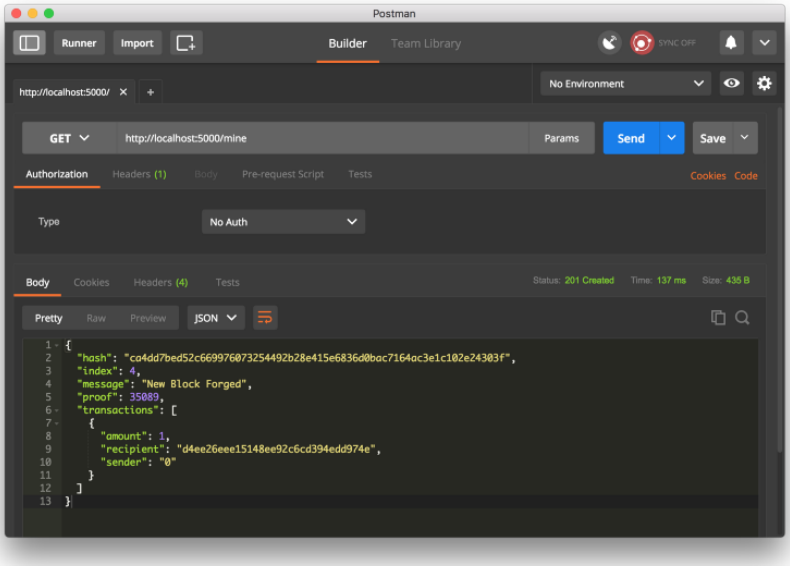
Step 3: Interacting with our Blockchain

You can use plain old cURL or Postman to interact with our API over a network.

Fire up the server:

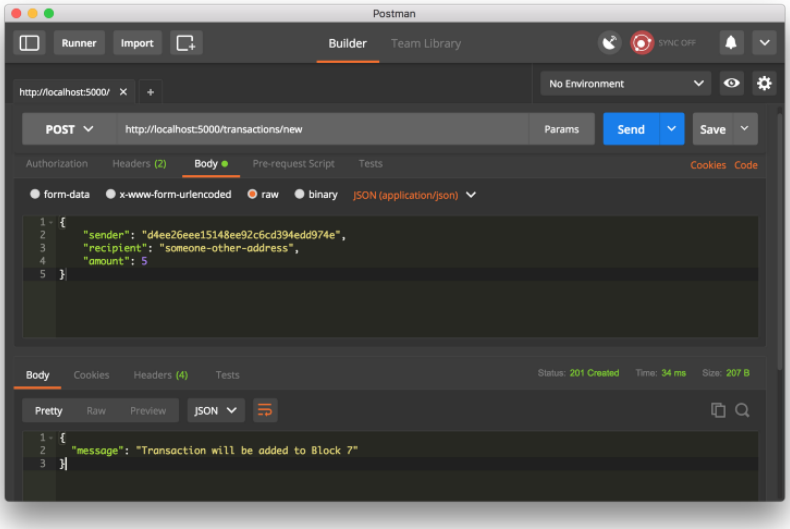
```
$ python blockchain.py

* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Let's try mining a block by making a `GET` request to `http://localhost:5000/mine`:



Using Postman to make a GET request

Let's create a new transaction by making a `POST` request to `http://localhost:5000/transactions/new` with a body containing our transaction structure:



Using Postman to make a POST request

If you aren't using Postman, then you can make the equivalent request using cURL:

```
$ curl -X POST -H "Content-Type: application/json" -d '{
 "sender": "d4ee26eee15148ee92c6cd394edd974e",
 "recipient": "someone-other-address",
 "amount": 5
}' "http://localhost:5000/transactions/new"
```

I restarted my server, and mined two blocks, to give 3 in total. Let's inspect the full chain by requesting `http://localhost:5000/chain`:

```
{
  "chain": [
    {
      "index": 1,
      "previous_hash": 1,
      "proof": 100,
      "timestamp": 1506280650.770839,
      "transactions": []
    },
    {
      "index": 2,
      "previous_hash": "c099bc...bfb7",
      "proof": 35293,
      "timestamp": 1506280664.717925,
      "transactions": [
```

```
          {
            "amount": 1,
            "recipient": "8bbcb347e0634905b0cac7955bae152b",
            "sender": "0"
          }
        ]
      },
      {
        "index": 3,
        "previous_hash": "eff91a...10f2",
        "proof": 35089,
        "timestamp": 1506280666.1086972,
        "transactions": [
          {
            "amount": 1,
            "recipient": "8bbcb347e0634905b0cac7955bae152b",
            "sender": "0"
          }
        ]
      }
    ],
    "length": 3
}
```

Step 4: Consensus

This is very cool. We've got a basic Blockchain that accepts transactions and allows us to mine new Blocks. But the whole point of Blockchains is that they should be *decentralized*. And if they're decentralized, how on earth do we ensure that they all reflect the same chain? This is called the problem of *Consensus*, and we'll have to implement a Consensus Algorithm if we want more than one node in our network.

Registering new Nodes

Before we can implement a Consensus Algorithm, we need a way to let a node know about neighbouring nodes on the network. Each node on our network should keep a registry of other nodes on the network. Thus, we'll need some more endpoints:

1. `/nodes/register` to accept a list of new nodes in the form of URLs.
2. `/nodes/resolve` to implement our Consensus Algorithm, which resolves any conflicts—to ensure a node has the correct chain.

We'll need to modify our Blockchain's constructor and provide a method for registering nodes:

```
...
from urllib.parse import urlparse
...


class Blockchain(object):
    def __init__(self):
        ...
        self.nodes = set()
        ...

    def register_node(self, address):
        """
        Add a new node to the list of nodes
```

A method for adding neighbouring nodes to our Network

Note that we've used a `set()` to hold the list of nodes. This is a cheap way of ensuring that the addition of new nodes is idempotent—meaning that no matter how many times we add a specific node, it appears exactly once.

Implementing the Consensus Algorithm

As mentioned, a conflict is when one node has a different chain to another node. To resolve this, we'll make the rule that *the longest valid chain is authoritative.* In other words, the longest chain on the network is the *de-facto* one. Using this algorithm, we reach *Consensus* amongst the nodes in our network.

```
...
import requests


class Blockchain(object)
    ...

    def valid_chain(self, chain):
        """
        Determine if a given blockchain is valid

        :param chain: <list> A blockchain
        :return: <bool> True if valid, False if not
        """

        last_block = chain[0]
        current_index = 1

        while current_index < len(chain):
            block = chain[current_index]
            print(f'{last_block}')
            print(f'{block}')
            print("\n-----------\n")
            # Check that the hash of the block is correct
            if block['previous_hash'] != self.hash(last_blo
                return False

            # Check that the Proof of Work is correct
            if not self.valid_proof(last_block['proof'], bl
                return False

            last_block = block
            current_index += 1

        return True

    def resolve_conflicts(self):
        """
        This is our Consensus Algorithm, it resolves confli
        by replacing our chain with the longest one in the

```

```
        :return: <bool> True if our chain was replaced, Fal
        """

```

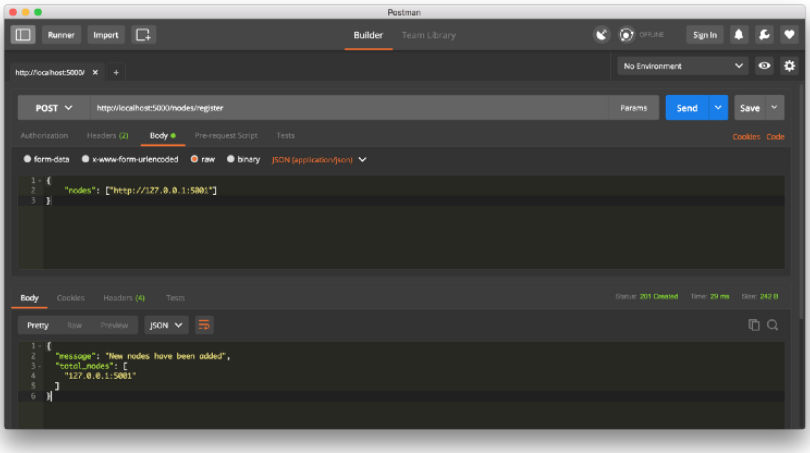
The first method `valid_chain()` is responsible for checking if a chain is valid by looping through each block and verifying both the hash and the proof.

`resolve_conflicts()` is a method which loops through all our neighbouring nodes, *downloads* their chains and verifies them using the above method. **If a valid chain is found, whose length is greater than ours, we replace ours.**

Let's register the two endpoints to our API, one for adding neighbouring nodes and the another for resolving conflicts:

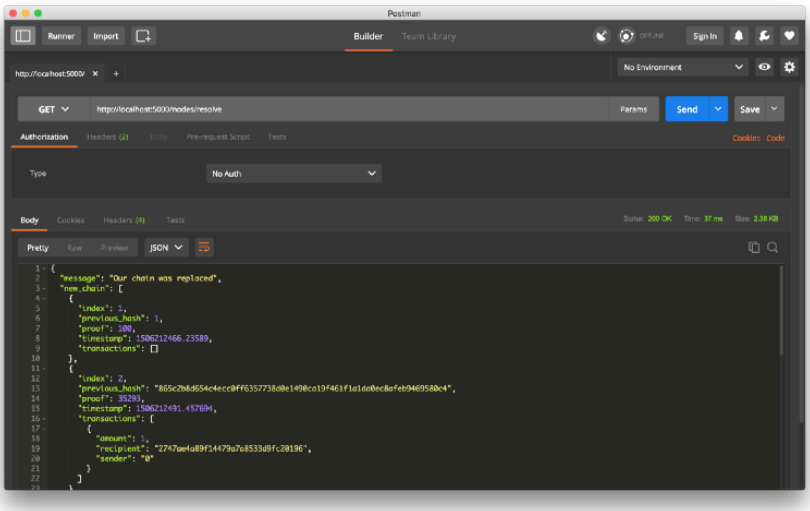
```
@app.route('/nodes/register', methods=['POST'])
def register_nodes():
    values = request.get_json()

    nodes = values.get('nodes')
    if nodes is None:
        return "Error: Please supply a valid list of nodes"

    for node in nodes:
        blockchain.register_node(node)

    response = {
        'message': 'New nodes have been added',
        'total_nodes': list(blockchain.nodes),
    }
    return jsonify(response), 201


@app.route('/nodes/resolve', methods=['GET'])
def consensus():
    replaced = blockchain.resolve_conflicts()

```

At this point you can grab a different machine if you like, and spin up different nodes on your network. Or spin up processes using different ports on the same machine. I spun up another node on my machine, on a different port, and registered it with my current node. Thus, I have two nodes: `http://localhost:5000` and `http://localhost:5001`.



Registering a new Node

I then mined some new Blocks on node 2, to ensure the chain was longer. Afterward, I called `GET /nodes/resolve` on node 1, where the chain was replaced by the Consensus Algorithm:



Consensus Algorithm at Work

And that's a wrap... Go get some friends together to help test out your Blockchain.

. . .

I hope that this has inspired you to create something new. I'm ecstatic about Cryptocurrencies because I believe that Blockchains will rapidly change the way we think about economies, governments and record-keeping.

Update: I'm planning on following up with a Part 2, where we'll extend our Blockchain to have a Transaction Validation Mechanism as well as discuss some ways in which you can productionize your Blockchain.

> *If you enjoyed this guide, or have any suggestions or questions, let me know in the comments. And if you've spotted any errors, feel free to contribute to the code here!*