Develop a blockchain application from scratch in Python

Get the code for a mini, fully functional blockchain-based project

Satwik Kansal

April 03, 2018

This tutorial introduces Python developers, of any programming skill level, to blockchain. You'll discover exactly what a blockchain is by implementing a public blockchain from scratch and building a simple application to leverage it. Python is an easy programming language to understand, and so I've chosen it for this tutorial.

This tutorial introduces Python developers, of any programming skill level, to blockchain. You'll discover exactly what a blockchain is by implementing a *public blockchain* from scratch and by building a simple application to leverage it.

You'll be able to create endpoints for different functions of the blockchain, such as adding a transaction, using the Flask microframework, and then run the scripts on multiple machines to create a decentralized network. You'll also see how to build a simple user interface that interacts with the blockchain and stores information for any use case, such as peer-to-peer payments, chatting, or e-commerce.

Python is an easy programming language to understand, so that's why I've chosen it for this tutorial. As you progress through this tutorial, you'll implement a public blockchain and see it in action. The code for a complete sample application, written using pure Python, is on GitHub.

Get the code

The main logic lies in the file views.py. To really understand blockchain from the ground up, let's walk through it together.

Prerequisites

- A basic programming knowledge of Python
- The Flask microframework (to create endpoints for the blockchain server)

© Copyright IBM Corporation 2018

Background

Public vs. private blockchain

A *public blockchain network*, like the Bitcoin network, is completely open to the public, and anyone can join and participate.

On the other hand, businesses who need greater privacy, security, and speed of transactions opt for a *private blockchain network*, where participants need to an invitation to join. Learn more.

In 2008, a whitepaper titled Bitcoin: A Peer-to-Peer Electronic Cash System was released by an individual (or maybe a group) named Satoshi Nakamoto. The paper combined cryptographic techniques and a peer-to-peer network without the need to trust a centralized authority (like banks) to make payments from one person to another person. Bitcoin was born. Apart from Bitcoin, that same paper introduced a distributed system of storing data (now popularly known as "blockchain"), which had far wider applicability than just payments, or cryptocurrencies.

Since then, interest in blockchain has exploded across nearly every industry. Blockchain is now the underlying technology behind fully digital cryptocurrencies like Bitcoin, distributed computing technologies like Ethereum, and open source frameworks like Hyperledger Fabric, on which the IBM Blockchain Platform is built.

Learn more about the IBM Blockchain Platform Starter Plan (free in beta!)

What is "blockchain"?

Blockchain is a way of storing digital data. The data can literally be anything. For Bitcoin, it's the transactions (transfers of Bitcoin from one account to another account), but it can even be files; it doesn't matter. The data is stored in the form of blocks, which are chained together using hashes. Hence the name "blockchain."

All of the magic lies in the way this data is added and stored in the blockchain, which yields some highly desirable characteristics:

- Immutability of history
- Un-hackability of the system
- Persistence of the data
- No single point of failure

So, how is blockchain able to achieve these characteristics? We'll get more into that as we implement one. Let's get started.

About the application

Let's define what the application we're building will do. Our goal is to build a simple website that allows users to share information. Because the content will be stored on the blockchain, it is immutable and permanent.

We'll follow a bottom-up approach to implement things. Let's begin by defining the structure of the data that we'll store in the blockchain. A post (a message posted by any user on our application) will be identified by three essential things:

1. Content
2. Author
3. Timestamp

1. Store transactions into blocks

We'll be storing data in our blockchain in a format that's widely used: JSON. Here's what a post stored in blockchain will look like:

```
{
    "author": "some_author_name",
    "content": "Some thoughts that author wants to share",
    "timestamp": "The time at which the content was created"
}
```

The generic term "data" is often replaced on the internet by the term "transactions." So, just to avoid confusion and maintain consistency, we'll be using the term "transaction" to refer to data posted in our example application.

The transactions are packed into blocks. So a block can contain one or many transactions. The blocks containing the transactions are generated frequently and added to the blockchain. Because there can be multiple blocks, each block should have a unique id:

```
class Block:
    def __init__(self, index, transactions, timestamp):
        self.index = []
        self.transactions = transactions
        self.timestamp = timestamp
```

2. Make the blocks immutable

We'd like to detect any kind of tampering in the data stored inside the block. In blockchain, this is done using a hash function.

A ***hash function*** is a function that takes data of any size and produces data of a fixed size from it, which generally works to identify the input. Here's an example in Python using the `sha256` hashing function:

```
>>> from hashlib import sha256
>>> data = "Some variable length data"
>>> sha256(data).hexdigest()
'b919fbbcae38e2bdaebb6c04ed4098e5c70563d2dc51e085f784c058ff208516'
>>> sha256(data).hexdigest() # no matter how many times you run it, the
 result is going to be the same 256 character string
'b919fbbcae38e2bdaebb6c04ed4098e5c70563d2dc51e085f784c058ff208516'
```

The characteristics of an ideal hash function are:

- It should be computationally easy to compute.
- Even a single bit change in data should make the hash change altogether.
- It should not be possible to guess the input from the output hash.

You now know what a hash function is. We'll store the hash of every block in a field inside our `Block` object to act like a digital fingerprint of data contained in it:

```
from hashlib import sha256
import json

def compute_hash(block):
     """
    A function that creates the hash of the block.
    """
    block_string = json.dumps(self.__dict__, sort_keys=True)
    return sha256(block_string.encode()).hexdigest()
```

Note: In most cryptocurrencies, even the individual transactions in the block are hashed, to form a hash tree (also known as a merkle tree), and the root of the tree might be used as the hash of the block. It's not a necessary requirement for the functioning of the blockchain, so we're omitting it to keep things neat and simple.

3. Chain the blocks

Okay, we've now set up the blocks. The blockchain is supposed to be a collection of blocks. We can store all of the blocks in the Python list (the equivalent of an array). But this is not sufficient, because what if someone intentionally replaces a block back in the collection? Creating a new block with altered transactions, computing the hash, and replacing with any older block is no big deal in our current implementation, because we will maintain the immutability and order of the blocks.

We need a way to make sure that any change in the past blocks invalidates the entire chain. One way to do this is to chain the blocks by the hash. By chaining here, we mean to include the hash of the previous block in the current block. So, if the content of any of the previous blocks changes, the hash of the block would change, leading to a mismatch with the `previous_hash` field in the next block.

Alright, every block is linked to the previous block by the `previous_hash` field, but what about the very first block? The very first block is called the ***genesis block*** and is generated manually or by some unique logic, in most cases. Let's add the `previous_hash` field to the `Block` class and implement the initial structure of our `Blockchain` class (see Listing 1).

Listing 1. The initial structure of our Blockchain class

```
from hashlib import sha256
import json
import time
class Block:
    def __init__(self, index, transactions, timestamp, previous_hash):
        self.index = index
        self.transactions = transactions
        self.timestamp = timestamp
        self.previous_hash = previous_hash

    def compute_hash(self):
        block_string = json.dumps(self.__dict__, sort_keys=True)
        return sha256(block_string.encode()).hexdigest()
```

And here's our Blockchain class:

```
class Blockchain:

    def __init__(self):
        self.unconfirmed_transactions = [] # data yet to get into blockchain
        self.chain = []
        self.create_genesis_block()

    def create_genesis_block(self):
        """
        A function to generate genesis block and appends it to
        the chain. The block has index 0, previous_hash as 0, and
        a valid hash.
        """
        genesis_block = Block(0, [], time.time(), "0")
        genesis_block.hash = genesis_block.compute_hash()
        self.chain.append(genesis_block)

    @property
    def last_block(self):
        return self.chain[-1]
```

4. Implement a Proof of Work algorithm

Selective endorsement vs. Proof of Work

Consensus in a blockchain for business, as supported by the IBM Blockchain Platform, is not achieved through mining but through a process called *selective endorsement*. The network members control exactly who verifies transactions, much in the same way that business happens today. Learn more about blockchain for business.

There's a problem, though. If we change the previous block, we can re-compute the hashes of all the following blocks quite easily and create a different valid blockchain. To prevent this, we must make the task of calculating the hash difficult and random.

Here's how we do this. Instead of accepting any hash for the block, we add some constraint to it. Let's add a constraint that our hash should start with two leading zeroes. Also, we know that unless we change the contents of the block, the hash is not going to change.

So we're going to introduce a new field in our block called *nonce*. A nonce is a number that we'll keep on changing until we get a hash that satisfies our constraint. The number of leading zeroes (the value 2, in our case) decides the "difficulty" of our Proof of Work algorithm. Also, you may

notice that our Proof of Work is difficult to compute but easy to verify once we figure out the nonce (to verify, you just have to run the hash function again):

```
class Blockchain:
    # difficulty of PoW algorithm
    difficulty = 2

    """
    Previous code contd..
    """

    def proof_of_work(self, block):
        """
        Function that tries different values of nonce to get a hash
        that satisfies our difficulty criteria.
        """
        block.nonce = 0

        computed_hash = block.compute_hash()
        while not computed_hash.startswith('0' * Blockchain.difficulty):
            block.nonce += 1
            computed_hash = block.compute_hash()

        return computed_hash
```

Notice that there is no definite logic to figure out the nonce quickly; it's just brute force.

5. Add blocks to the chain

To add a block to the chain, we'll first have to verify if the Proof of Work provided is correct and if the `previous_hash` field of the block to be added then points to the hash of the latest block in our chain.

Let's see the code for adding blocks into the chain:

```
class Blockchain:
    """
    Previous code contd..
    """
    def add_block(self, block, proof):
        """
        A function that adds the block to the chain after verification.
        """
        previous_hash = self.last_block.hash

        if previous_hash != block.previous_hash:
            return False

        if not self.is_valid_proof(block, proof):
            return False

        block.hash = proof
        self.chain.append(block)
        return True

    def is_valid_proof(self, block, block_hash):
        """
        Check if block_hash is valid hash of block and satisfies
        the difficulty criteria.
        """
        return (block_hash.startswith('0' * Blockchain.difficulty) and
                block_hash == block.compute_hash())
```

Mining

The transactions are initially stored in a pool of unconfirmed transactions. The process of putting the unconfirmed transactions in a block and computing Proof of Work is known as the *mining* of blocks. Once the nonce satisfying our constraints is figured out, we can say that a block has been mined, and the block is put into the blockchain.

In most of the cryptocurrencies (including Bitcoin), miners may be awarded some cryptocurrency as a reward for spending their computing power to compute a Proof of Work. Here's what our mining function looks like:

```
class Blockchain:
    """
    Previous code contd...
    """

    def add_new_transaction(self, transaction):
            self.unconfirmed_transactions.append(transaction)

    def mine(self):
        """
        This function serves as an interface to add the pending
        transactions to the blockchain by adding them to the block
        and figuring out Proof of Work.
        """
        if not self.unconfirmed_transactions:
            return False

        last_block = self.last_block

        new_block = Block(index=last_block.index + 1,
                          transactions=self.unconfirmed_transactions,
                          timestamp=time.time(),
                          previous_hash=last_block.hash)

        proof = self.proof_of_work(new_block)
        self.add_block(new_block, proof)
        self.unconfirmed_transactions = []
        return new_block.index
```

Alright, we're almost there. You can see the combined code up to this point on GitHub.

6. Create interfaces

Okay, now it's time to create interfaces for our node to interact with other peers as well as with the application we're going to build. We'll be using Flask to create a REST-API to interact with our node. Here's the code for it:

```
from flask import Flask, request
import requests

app =  Flask(__name__)

# the node's copy of blockchain
blockchain = Blockchain()
```

We need an endpoint for our application to submit a new transaction. This will be used by our application to add new data (posts) to the blockchain:

```
@app.route('/new_transaction', methods=['POST'])
def new_transaction():
    tx_data = request.get_json()
    required_fields = ["author", "content"]

    for field in required_fields:
        if not tx_data.get(field):
            return "Invlaid transaction data", 404

    tx_data["timestamp"] = time.time()

    blockchain.add_new_transaction(tx_data)

    return "Success", 201
```

Here's an endpoint to return the node's copy of the chain. Our application will be using this endpoint to query all of the posts to display:

```
@app.route('/chain', methods=['GET'])
def get_chain():
    chain_data = []
    for block in blockchain.chain:
        chain_data.append(block.__dict__)
    return json.dumps({"length": len(chain_data),
                       "chain": chain_data})
```

Here's an endpoint to request the node to mine the unconfirmed transactions (if any). We'll be using it to initiate a command to mine from our application itself:

```
@app.route('/mine', methods=['GET'])
def mine_unconfirmed_transactions():
    result = blockchain.mine()
    if not result:
        return "No transactions to mine"
    return "Block #{} is mined.".format(result)


# endpoint to query unconfirmed transactions
@app.route('/pending_tx')
def get_pending_tx():
    return json.dumps(blockchain.unconfirmed_transactions)


app.run(debug=True, port=8000)
```

Now, if you'd like, you can play around with our blockchain by creating some transactions and then mining them using a tool like cURL or Postman.

7. Establish consensus and decentralization

The code that we've implemented till now is meant to run on a single computer. Even though we're linking block with hashes, we still can't trust a single entity. We need multiple nodes to maintain our blockchain. So, let's create an endpoint to let a node know of other peers in the network:

```
# the address to other participating members of the network
peers = set()

# endpoint to add new peers to the network.
@app.route('/add_nodes', methods=['POST'])
def register_new_peers():
    nodes = request.get_json()
    if not nodes:
        return "Invalid data", 400
    for node in nodes:
        peers.add(node)

    return "Success", 201
```

You might have realized that there's a problem with multiple nodes. Due to intentional manipulation or unintentional reasons, the copy of chains of a few nodes can differ. In that case, we need to agree upon some version of the chain to maintain the integrity of the entire system. We need to achieve consensus.

A simple consensus algorithm could be to agree upon the longest valid chain when the chains of different participants in the network appear to diverge. The rationale behind this approach is that the longest chain is a good estimate of the most amount of work done:

```
def consensus():
    """
    Our simple consensus algorithm. If a longer valid chain is found, our chain is replaced with it.
    """
    global blockchain

    longest_chain = None
    current_len = len(blockchain)

    for node in peers:
        response = requests.get('http://{}/chain'.format(node))
        length = response.json()['length']
        chain = response.json()['chain']
        if length > current_len and blockchain.check_chain_validity(chain):
            current_len = length
            longest_chain = chain

    if longest_chain:
        blockchain = longest_chain
        return True

    return False
```

And now finally, we need to develop a way for any node to announce to the network that it has mined a block so that everyone can update their blockchain and move on to mine other transactions. Other nodes can simply verify the proof of work and add it to their respective chains:

```
# endpoint to add a block mined by someone else to the node's chain.
@app.route('/add_block', methods=['POST'])
def validate_and_add_block():
    block_data = request.get_json()
    block = Block(block_data["index"], block_data["transactions"],
                  block_data["timestamp", block_data["previous_hash"]])

    proof = block_data['hash']
    added = blockchain.add_block(block, proof)
```

```
    if not added:
        return "The block was discarded by the node", 400

    return "Block added to the chain", 201

    def announce_new_block(block):
    for peer in peers:
        url = "http://{}/add_block".format(peer)
        requests.post(url, data=json.dumps(block.__dict__, sort_keys=True))
```

The `announce_new_block` method should be called after every block is mined by the node, so that peers can add it to their chains.

8. Build the application

Alright, the backend is all set up. You can see the code up to this point on GitHub.

Now, it's time to start working on the interface of our application. We've used Jinja2 templating to render the web pages and some CSS to make the page look nice.

Our application needs to connect to a node in the blockchain network to fetch the data and also to submit new data. There can be multiple nodes as well:

```
import datetime
import json

import requests
from flask import render_template, redirect, request

from app import app

.
CONNECTED_NODE_ADDRESS = "http://127.0.0.1:8000"

posts = []
```

The `fetch_posts` function gets the data from node's /`chain` endpoint, parses the data and stores it locally.

```
def fetch_posts():
    get_chain_address = "{}/chain".format(CONNECTED_NODE_ADDRESS)
    response = requests.get(get_chain_address)
    if response.status_code == 200:
        content = []
        chain = json.loads(response.content)
        for block in chain["chain"]:
            for tx in block["transactions"]:
                tx["index"] = block["index"]
                tx["hash"] = block["previous_hash"]
                content.append(tx)

        global posts
        posts = sorted(content, key=lambda k: k['timestamp'],
                       reverse=True)
```

The application has an HTML form to take user input and then makes a `POST` request to a connected node to add the transaction into the unconfirmed transactions pool. The transaction is then mined by the network, and then finally will be fetched once we refresh our website:

```
@app.route('/submit', methods=['POST'])
def submit_textarea():
    """
    Endpoint to create a new transaction via our application.
    """
    post_content = request.form["content"]
    author = request.form["author"]

    post_object = {
        'author': author,
        'content': post_content,
    }

    # Submit a transaction
    new_tx_address = "{}/new_transaction".format(CONNECTED_NODE_ADDRESS)

    requests.post(new_tx_address,
                  json=post_object,
                  headers={'Content-type': 'application/json'})

    return redirect('/')
```

9. Run the application

It's done! You can find the final code on GitHub.

To run the application:

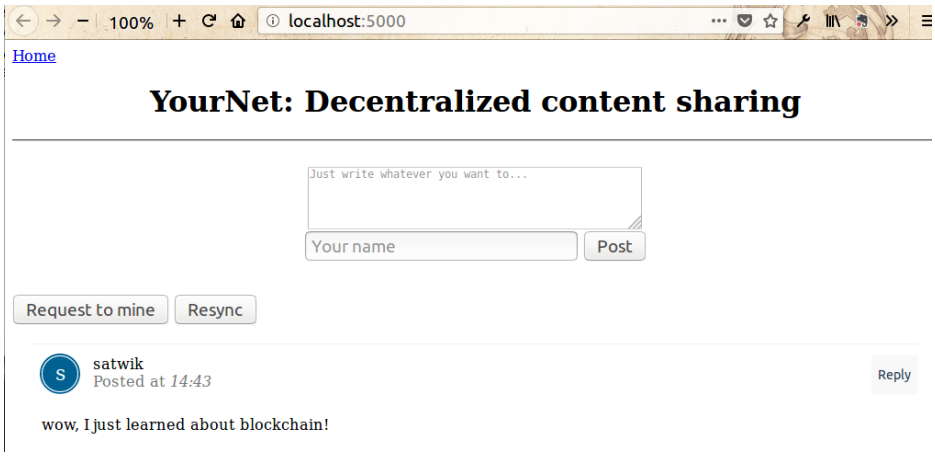
1. Start a blockchain node server:
   `python node_server.py`
2. Run the application:
   `python run_app.py`

You should see the running application at http://localhost:5000.

a. Try posting some data, and you should see something like this:

b. Click the **Request to mine** button, and you should see something like this:

c. Click the **Resync** button, and you should see the application resyncing with the chain:



Verify transactions

You might have noticed a flaw in the application: Anyone can change any name and post any content. One way to solve this is by creating accounts using public-private key cryptography. Every new user needs a public key (analogous to username) and the private key to be able to post in our application. The keys act as a digital signature. The public key can only decode the content encrypted by the corresponding private key. The transactions will be verified using the public key of the author before adding to any block. This way, we'll know who exactly wrote the message.

Conclusion

This tutorial covered the fundamentals of a public blockchain. If you followed along, you implemented a blockchain from scratch and built a simple application allowing users to share information on the blockchain.

Next steps

You can spin off multiple nodes on the cloud and play around with the application you've built. You can deploy any Flask application to the IBM Cloud.

Alternatively, you can use a tunneling service like ngrok to create a public URL for your localhost server, and then you'll be able to interact with multiple machines.

There's a lot to explore in this space! Here are several ways to continue building your blockchain skills:

- Continue exploring blockchain technology by getting your hands on the new IBM Blockchain Platform Starter Plan (free beta). You can quickly spin up a blockchain pre-production network, deploy sample applications, and develop and deploy client applications. **Get started**!
- Stay in the know with the Blockchain Newsletter from developerWorks. Check out the **current issue** and **subscribe**.

- Stop by the **Blockchain Developer Center** on developerWorks. It's your source for tools and tutorials, along with code and community support, for developing and deploying blockchain solutions for business.
- Take the **Blockchain essentials course for developers** to learn the ins and outs of asset transfers. At the end of this free, self-paced, 2-hour course, take a quiz, get a badge, and start planning useful blockchain applications for your business network.
- Advance to the **IBM Blockchain foundation developer course**, a free 6-hour course that expands on "Blockchain essentials" and provides a more detailed picture of the components and architecture of blockchain business networks, as well as experience building a network and creating an application.
- Check out the many **blockchain code patterns**, which provide roadmaps for solving complex problems, and include overviews, architecture diagrams, process flows, repo pointers, and additional reading.

About the author

Satwik Kansal is a Software Developer with experience in blockchain technologies and Data Science. Connect with him on Twitter or Linkedin or at his website.

© Copyright IBM Corporation 2018
(www.ibm.com/legal/copytrade.shtml)
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)