Tutorial

Develop a blockchain application from scratch in Python

Implement a mini, fully-functional blockchain-based project

Save Like

By Satwik Kansal

Published January 29, 2020

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 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 available on GitHub.

Get the code.

Now, to understand blockchain from the ground up, let's walk through it together.

Prerequisites

- Basic programming knowledge of Python
- Knowledge of REST-APIs
- Familiarity with the Flask microframework (not mandatory, but nice to have)

Background

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 several cryptographic techniques and a peer-to-peer network to transfer payments without the involvement of any central authority (like a bank). A cryptocurrency named 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, blockchain has attracted interest 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.

What is "blockchain"?

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

All of the magic lies in the way this data is stored and added to the blockchain. A blockchain is essentially a linked list that contains ordered data, with a few constraints such as:

- Blocks can't be modified once added; in other words, it is *append only*.
- There are specific rules for appending data to it.
- Its architecture is distributed.

Enforcing these constraints yields the following benefits:

- Immutability and durability of data
- No single point of control or failure
- A verifiable audit trail of the order in which data was added

So, how can these constraints achieve these characteristics? We'll get more into that as we implement this blockchain. Let's get started.

About the application

Let's briefly define the scope of our mini-application. Our goal is to build an application that allows users to share information by posting. Since the content will be stored on the blockchain, it will be immutable and permanent. Users will interact with the application through a simple web interface.

Steps

1. Store transactions into blocks
2. Add digital fingerprints to the blocks
3. Chain the blocks
4. Implement a proof of work algorithm
5. Add blocks to the chain
6. Create interfaces
7. Establish consensus and decentralization
8. Build the application
9. Run the application

We'll implement things using a bottom-up approach. Let's begin by defining the structure of the data that we'll store in the blockchain. A **post** is a message that's posted by any user on our application. Each post will consist of three essential elements:

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"
}
```

Show more

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 in our example application.

The transactions are packed into blocks. 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):
        """
        Constructor for the `Block` class.
        :param index: Unique ID of the block.
        :param transactions: List of transactions.
        :param timestamp: Time of generation of the block.
        """
        self.index = index
        self.transactions = transactions
        self.timestamp = timestamp
```

Show more

2. Add digital fingerprints to the blocks

We'd like to prevent any kind of tampering in the data stored inside the block, and detection is the first step to that. To detect if the data in the block has been tampered with, you can use cryptographic hash functions.

A **hash function** is a function that takes data of any size and produces data of a fixed size from it (a hash), which is generally used to identify the input. The characteristics of an ideal hash function are:

- It should be easy to compute.
- It should be deterministic, meaning the same data will always result in the same hash.
- It should be uniformly random, meaning even a single bit change in the data should change the hash significantly.

The consequence of this is:

- It is virtually impossible to guess the input data given the hash. (The only way is to try all possible input combinations.)
- If you know both the input and the hash, you can simply pass the input through the hash function to verify the provided hash.

This asymmetry of efforts that's required to figure out the hash from an input (easy) vs. figuring out the input from a hash (almost impossible) is what blockchain leverages to obtain the desired characteristics.

There are various popular hash functions. Here's an example in Python that uses the SHA-256 hashing function:

```
>>> from hashlib import sha256
>>> data = b"Some variable length data"
>>> sha256(data).hexdigest()
'b919fbbcae38e2bdaebb6c04ed4098e5c70563d2dc51e085f784c058ff208516'
>>> sha256(data).hexdigest() # no matter how many times you run it, the result is
'b919fbbcae38e2bdaebb6c04ed4098e5c70563d2dc51e085f784c058ff208516'
>>>  data = b"Some variable length data2" # Added one character at the end.
'9fcaab521baf8e83f07512a7de7a0f567f6eef2688e8b9490694ada0a3ddeec8'

# Note that the hash has changed entirely!
```

Show more

We'll store the hash of the block in a field inside our `Block` object, and it will act like a digital fingerprint (or signature) of data contained in it:

```
from hashlib import sha256
import json

def compute_hash(block):
    """
    Returns the hash of the block instance by first converting it
    into JSON string.
    """
    block_string = json.dumps(self.__dict__, sort_keys=True)
    return sha256(block_string.encode()).hexdigest()
```

Show more

Site feedback

Note: In most cryptocurrencies, even the individual transactions in the block are hashed and then stored to form a hash tree (also known as a merkle tree). The root of the tree usually represents 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 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 the blocks in the Python list (the equivalent of an array). But this is not sufficient, because what if someone intentionally replaces an old block with a new block in the collection? Creating a new block with altered transactions, computing the hash, and replacing it with any older block is no big deal in our current implementation.

We need a way to make sure that any change in the previous blocks invalidates the entire chain. The Bitcoin way to do this is to create dependency among consecutive blocks by chaining them with the hash of the block immediately previous to them. By *chaining* here, we mean to include the hash of the previous block in the current block in a new field called `previous_hash`.

Okay, if every block is linked to the previous block through the `previous_hash` field, what about the very first block? That block is called the **genesis block** and it can be generated either manually or through some unique logic. Let's add the `previous_hash` field to the `Block` class and implement 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):
        """
```

```
        Constructor for the `Block` class.
        :param index:         Unique ID of the block.
        :param transactions:  List of transactions.
        :param timestamp:     Time of generation of the block.
        :param previous_hash: Hash of the previous block in the chain which this
        """
        self.index = index
```

Show more

Now, if the content of any of the previous blocks changes:

- The hash of that previous block would change.
- This will lead to a mismatch with the `previous_hash` field in the next block.
- Since the input data to compute the hash of any block also consists of the `previous_hash` field, the hash of the next block will also change.

Ultimately, the entire chain following the replaced block is invalidated, and the only way to fix it is to recompute the entire chain.

4. Implement a proof of work algorithm

There is one problem, though. If we change the previous block, the hashes of all the blocks that follow can be re-computed quite easily to create a different valid blockchain. To prevent this, we can exploit the asymmetry in efforts of hash functions that we discussed earlier to 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 "n leading zeroes" where n can be any positive integer.

We know that unless we change the data of the block, the hash is not going to change, and of course we don't want to change existing data. So what do we do? Simple! We'll add some dummy data that we can change. Let's introduce a new field in our block called **nonce**. A nonce is a number that we can keep on changing until we get a hash that satisfies our constraint. The nonce satisfying the constraint serves as proof that some computation has been performed. This technique is a simplified version of the Hashcash algorithm used in Bitcoin. The number of zeroes specified in the constraint determines the difficulty of our proof of work algorithm (the greater the number of zeroes, the harder it is to figure out the nonce).

Also, due to the asymmetry, proof of work is difficult to compute but very easy to verify once you figure out the nonce (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 the nonce to get a hash
        that satisfies our difficulty criteria.
        """
        block.nonce = 0
```

Show more

Notice that there is no specific logic to figuring out the nonce quickly; it's just brute force. The only definite improvement that you can make is to use hardware chips that are specially designed to compute the hash function in a smaller number of CPU instructions.

5. Add blocks to the chain

To add a block to the chain, we'll first have to verify that:

- The data has not been tampered with (the proof of work provided is correct).
- The order of transactions is preserved (the `previous_hash` field of the block to be added 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.
        Verification includes:
        * Checking if the proof is valid.
        * The previous_hash referred in the block and the hash of a latest block
          in the chain match.
        """
        previous_hash = self.last_block.hash
```

Show more

Mining

The transactions will be initially stored as 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 it can be 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:

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

    """
    Previous code contd...
    """

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

    def mine(self):
```

Show more

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 blockchain node to interact with the application we're going to build. We'll be using a popular Python microframework called Flask to create a REST API that interacts with and invokes various operations in our blockchain node. If you've worked with any web framework before, the code below shouldn't be difficult to follow along.

```
from flask import Flask, request
import requests

# Initialize flask application
app =  Flask(__name__)
```

```
# Initialize a blockchain object.
blockchain = Blockchain()
```

Show more

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:

```
# Flask's way of declaring end-points
@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 "Invalid transaction data", 404

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

    blockchain.add_new_transaction(tx_data)

    return "Success", 201
```

Show more

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 data 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})
```

Show more

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)

@app.route('/pending_tx')
```

```
def get_pending_tx():
    return json.dumps(blockchain.unconfirmed_transactions)
```

Show more

These REST endpoints can be used to play around with our blockchain by creating some transactions and then mining them.

7. Establish consensus and decentralization

Up to this point, the blockchain that we've implemented is meant to run on a single computer. Even though we're linking block with hashes and applying the proof of work constraint, we still can't trust a single entity (in our case, a single machine). We need the data to be distributed, we need multiple nodes maintaining the blockchain. So, to transition from a single node to a peer-to-peer network, let's first create a mechanism to let a new node become aware of other peers in the network:

```
# Contains the host addresses of other participating members of the network
peers = set()

# Endpoint to add new peers to the network
@app.route('/register_node', methods=['POST'])
def register_new_peers():
    # The host address to the peer node
    node_address = request.get_json()["node_address"]
    if not node_address:
        return "Invalid data", 400

    # Add the node to the peer list
    peers.add(node_address)

    # Return the blockchain to the newly registered node so that it
```

Show more

A new node participating in the network can invoke the `register_with_existing_node` method (via the `/register_with` endpoint) to register with existing nodes in the network. This will help with the following:

- Asking the remote node to add a new peer to its list of known peers.
- Initializing the blockchain of the new node with that of the remote node.
- Resyncing the blockchain with the network if the node goes off-grid.

However, there's a problem with multiple nodes. Due to intentional manipulation or unintentional reasons (like network latency), the copy of chains of a few nodes can differ.

In that case, the nodes need to agree upon some version of the chain to maintain the integrity of the entire system. In other words, we need to achieve *consensus*.

A simple consensus algorithm could be to agree upon the longest valid chain when the chains of different participating nodes 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 (remember proof of work is difficult to compute):

```
class Blockchain
    """
    previous code continued...
    """
    def check_chain_validity(cls, chain):
        """
        A helper method to check if the entire blockchain is valid.
        """
        result = True
        previous_hash = "0"

        # Iterate through every block
        for block in chain:
            block_hash = block.hash
            # remove the hash field to recompute the hash again
```

Show more

Next, 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 the mined block to their respective chains (remember that verification is easy once the nonce is known):

```
# endpoint to add a block mined by someone else to
# the node's chain. The node first verifies the block
# and then adds it to the chain.
@app.route('/add_block', methods=['POST'])
def verify_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:
```

Show more

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.

```
@app.route('/mine', methods=['GET'])
def mine_unconfirmed_transactions():
    result = blockchain.mine()
    if not result:
        return "No transactions to mine"
    else:
        # Making sure we have the longest chain before announcing to the network
        chain_length = len(blockchain.chain)
        consensus()
        if chain_length == len(blockchain.chain):
            # announce the recently mined block to the network
            announce_new_block(blockchain.last_block)
        return "Block #{} is mined.".format(blockchain.last_block.index
```

Show more

8. Build the application

Alright, the blockchain server 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 things 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 also be multiple nodes, as well.

```
import datetime
import json

import requests
from flask import render_template, redirect, request

from app import app

# Node in the blockchain network that our application will communicate with
# to fetch and add data.
CONNECTED_NODE_ADDRESS = "http://127.0.0.1:8000"

posts = []
```

Show more

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

```
def fetch_posts():
    """
```

```
    Function to fetch the chain from a blockchain node, parse the
    data, and store it locally.
    """
    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"]
```

Show more

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 fetched once we refresh our web page:

```
@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_ADDR
```

Show more

9. Run the application

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

Clone the project:

```
$ git clone https://github.com/satwikkansal/python_blockchain_app.git
```

Show more

Install the dependencies:

```
$ cd python_blockchain_app
$ pip install -r requirements.txt
```

Show more

Start a blockchain node server:

```
$ export FLASK_APP=node_server.py
$ flask run --port 8000
```

Show more

One instance of our blockchain node is now up and running at port 8000.

Run the application on a different terminal session:

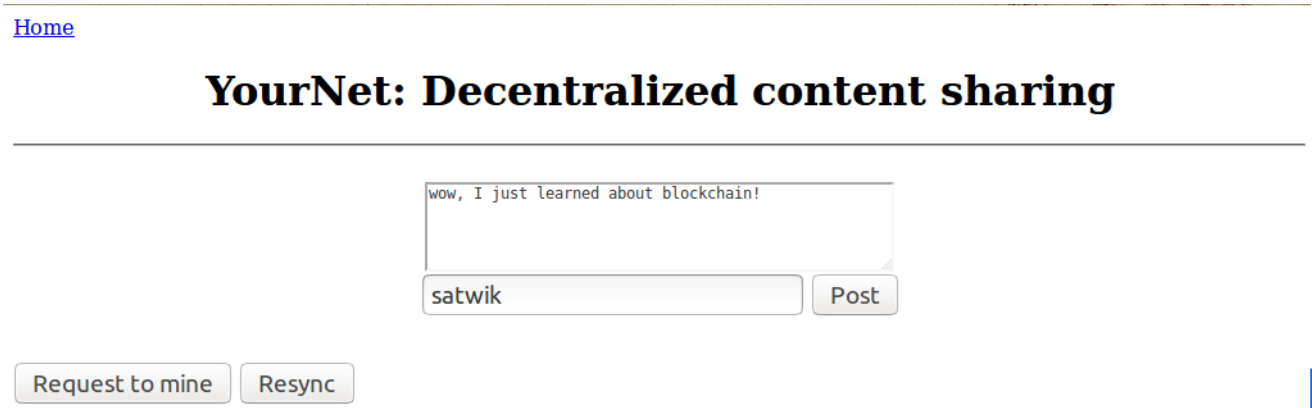
```
$ python run_app.py
```

Show more

The application should be up and running at http://localhost:5000.

Figures 1 – 3 illustrate how to post content, request a node to mine, and resync with the chain.

Figure 1. Posting some content



Site feedback

Figure 2. Requesting the node to mine

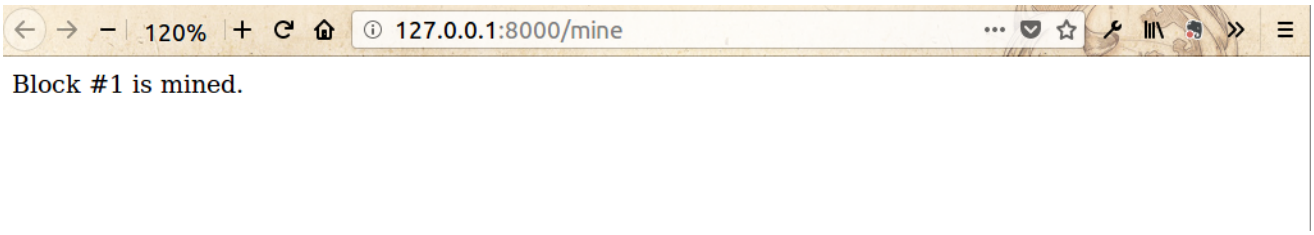
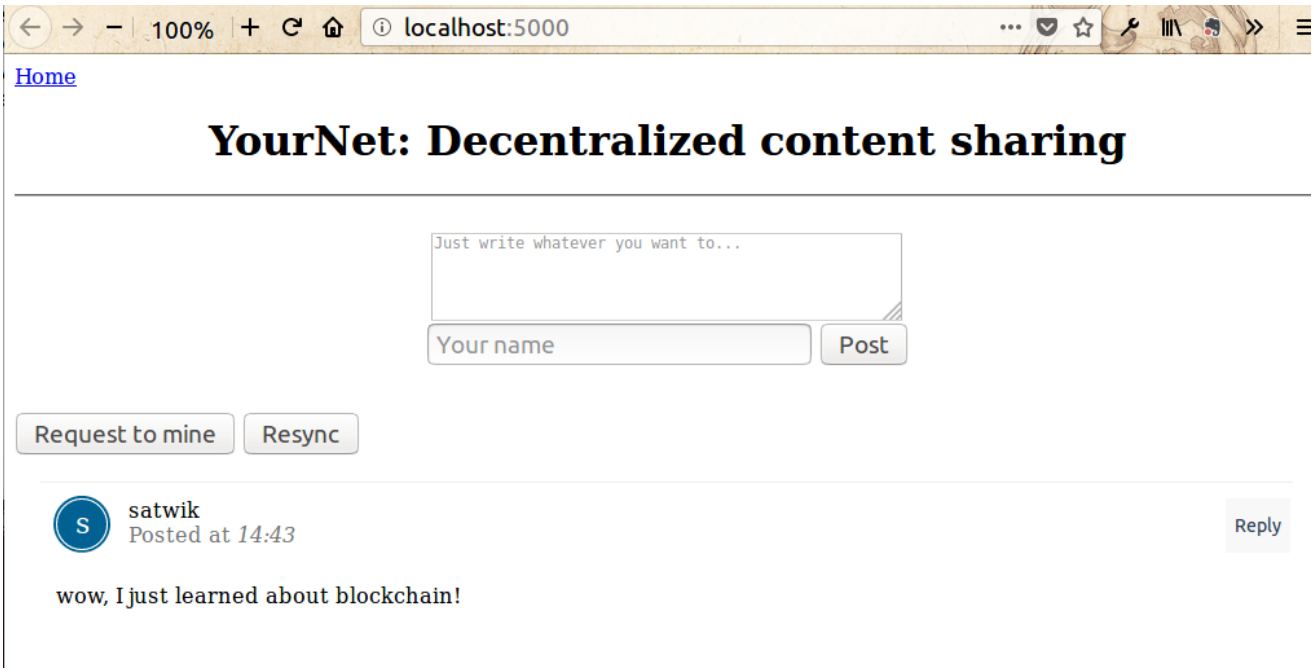


Figure 3. Resyncing with the chain for updated data



Running with multiple nodes

To play around by spinning off multiple custom nodes, use the `register_with/` endpoint to register a new node with the existing peer network.

Here's a sample scenario that you might want to try:

```
# already running
$ flask run --port 8000 &
# spinning up new nodes
$ flask run --port 8001 &
$ flask run --port 8002 &
```

Show more

You can use the following cURL requests to register the nodes at port `8001` and `8002` with the already running `8000`:

```
$ curl -X POST \
  http://127.0.0.1:8001/register_with \
  -H 'Content-Type: application/json' \
  -d '{"node_address": "http://127.0.0.1:8000"}'

$ curl -X POST \
  http://127.0.0.1:8002/register_with \
  -H 'Content-Type: application/json' \
  -d '{"node_address": "http://127.0.0.1:8000"}'
```

Show more

This will make the node at port `8000` aware of the nodes at port `8001` and `8002`, and vice-versa. The newer nodes will also sync the chain with the existing node so that they are able to participate in the mining process actively.

To update the node with which the frontend application syncs (default is localhost port `8000`), change the `CONNECTED_NODE_ADDRESS` field in the views.py file.

Once you do all this, you can run the application (`python run_app.py`) and create transactions (post messages via the web interface), and once you mine the transactions all the nodes in the network will update the chain. The chain of nodes can also be inspected by invoking the `/chain` endpoint using cURL or Postman.

```
$ curl -X GET http://localhost:8001/chain
$ curl -X GET http://localhost:8002/chain
```

Show more

Authenticate transactions

You might have noticed a flaw in the application: Anyone can change any name and post any content. Also, the post is susceptible to tampering while submitting the transaction to the blockchain network. One way to solve this is by creating user accounts using public-private key cryptography . Every new user needs a public key (analogous to username) and a private key to be able to post in our application. The keys are used to create and verify the digital signature. Here's how it works:

- Every new transaction submitted (post submitted) is signed with the user's private key. This signature is added to the transaction data along with the user information.
- During the verification phase, while mining the transactions, we can verify if the claimed owner of the post is the same as the one specified in the transaction data, and also that the message has not been modified. This can be done using the signature and the public key of the claimed owner of the post.

Conclusion

This tutorial covered the fundamentals of a public blockchain. If you've followed along, you should now be able to implement a blockchain from scratch and build a simple application that allows users to share information on the blockchain. This implementation is not as sophisticated as other public blockchains like Bitcoin or Ethereum (and still has some loopholes) — but if you keep asking the right questions as per your requirements, you'll eventually get there. The key thing to note is that most of the work in designing a blockchain for your needs is about the amalgamation of existing computer science concepts, nothing more!

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**. 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 IBM Developer**. Check out recent issues and subscribe .
- Stop by the **Blockchain hub** on IBM Developer. It's your source for tools and tutorials, along with code and community support, for developing and deploying blockchain solutions for business.
- Continue building your blockchain skills through the IBM Developer **Blockchain learning path**, which gives you the fundamentals and then shows you how to start creating apps, and offers helpful use cases for perspective.
- And be sure to check out the many **blockchain code patterns** on IBM Developer, which provide roadmaps for solving complex problems, and include overviews, architecture diagrams, process flows, repo pointers, and additional reading.

Resources

Site feedback

Related

Tutorial

IBM Blockchain 101: Quick-start guide for developers

June 5, 2020

Series

Learning Path: Start working with blockchain

June 12, 2019

Series

Build your first blockchain application

April 16, 2019

Site feedback

Build Smart
Build Secure

IBM Developer

About
FAQ
Report abuse
Third-party notice

Follow Us

Twitter
LinkedIn
Facebook
YouTube

Explore

Newsletters
Code patterns
Articles
Tutorials
Open source projects
Videos
Events
Cities

Select a language

English
中文
日本語
Português
Español
한글

Community
Privacy
Terms of use
Accessibility
Cookie preferences