

Table of contents

- 1. Brief description of the project**
- 2. Important Variables and data structures**
- 3. 10 essential functions used in our project**
- 4. Libraries used**
- 5. Our application in action**
- 6. Who did what?**
- 7. Bibliography**

Blockchain Documentation

Brief description of our project

The aim of our work was to create a recent security mechanism, which is more and more used in the computing universe, known as the blockchain. In this project, we put in place algorithms and security measures, which will prevent data from being modified by other users or anyone. We are also going to put in place a user interface to enable us to perform certain operations on our blockchain. These operations are described more in details below.

Important variables

For a better comprehension of this documentation, here are important variables and data structure names used in our code.

1. Blockchain: Our list of chains
2. Open transactions: The transactions not yet mined
3. Public key: The public key of node
4. Private key: The private key of a node
5. Signature: The signature of a node

NB: A node is a member of our blockchain network

The 10 essential functions

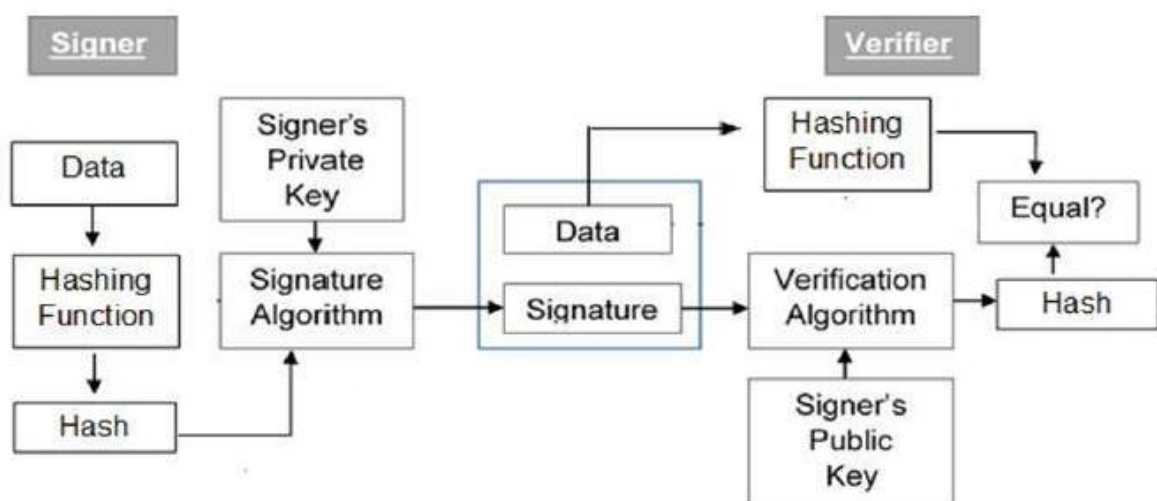
In the following, there is a list of important and crucial functions to our blockchain, listed in a chronological order, so that the lecturer can better understand what is being done.

1. **Hash_block** : The hash_block function is the function responsible to hash our blocks. After adding the different transactions, they are stored in the open_transactions list and need to be added to the blockchain (during the mining process). This function makes use of the **hashlib** package of python and the **SHA256** hashing method. The **sha256** creates a 64 character hash. Values accepted are just strings. So, since the block is a dictionary, we can convert it to a string with the **JSON** method, which is imported from the **JSON** package of python. JSON encodes objects and data structures as a string. The **Encode()** method is used to ensure that the string is converted in UTF-8. The hash generated is not a string, but a byte hash. The **hexdigest()** method returns the string version of the hash.

Syntax: `hashlib.sha256(json.dumps(block).encode())`

Note: We are not using a hash because we want to hide the data. The data must be publicly available and read from anyone. We use the hash in order to store a large amount of data in a small amount of space.

- **Generate_keys:** Each member in our blockchain is called a Node. These Nodes need identifiers in order to be identified on the node. They also need public and private keys in order to be able to sign a transaction as described below. Pycryptodome is an external module used to generate private and public keys. It is not shipped with python but it can be installed using either pip or anaconda. We are going to generate our public and private keys using the RSA algorithm. To import this package, we type: `from Crypto.PublicKey import RSA`. We also import a package that will generate a random number such as `randint` of python. The private key is generated randomly with the **`RSA.generate()`** method of the `RSA` class which takes the wished length of the key and a random number as parameters and the public key is calculated from the randomly generated private key using the **`public_key()`** function of **`pycryptodome`**.
- **Sign_transaction:** When new transactions are added, they are stored in a list waiting to be mined in order to be added in the blockchain. During this time, they can be hacked and modified since there are no security measures on it. In order to prevent this, the transactions are signed by the sender by the time it is emitted. The algorithm used to generate these signatures is `PKCS1_v1_5` from the package `pycryptodome`. A digital signature is a key generated from some data (the data we wish to keep authentic) and a key known only by the user (private key).



Since we need data and a key that just the user knows, these data will be the combinations of the sender's id, amount sent and recipient's id. The key will be the private key. Here in creating the signatures, we actually do the reverse operation of when we generate a key in order to have the private key in binary form. After having generated the signature, we add this to the transaction dictionary as a data of the transaction.

- **Verify_signature:** Since the signature is used as a security measure, it needs to be verified when this is necessary, like when we add a new transaction. To verify the signatures, there exists a verify method of the `pycryptodome` package which takes in as parameters the hash of the data used and the binary version of the transaction's signature. So, this function gets the value of the public key of our node and convert it from string to binary data. It also gets the signature of the transaction and the other values that were used **sign_transaction** to generate this signature. The **verify** method of the **pycryptodome** package is used and these data are passed as argument. This function returns True if the transaction is correct and False if it was being modified.
- **Mine_block:** Mining blocks should be challenging, i.e, it must require a great amount of time. This is:
 - To control the rate at which new groups of transactions (blocks) are added to the network.
 - For security. The comparison of the hash of the previous block and the current block was already a security measure, but it is still possible to overcome it by changing the data in a block and adjust the hash of subsequent blocks so that it can accept the modification. The prove of work basically prevents this because the modifications of just one of the blocks will require a great amount of time and computing power to recalculate the proof.

This function starts by verifying if we have an owner. If it is the case, it verifies if the transactions in the open transactions are correct. This is with the help of **verify_signature**. If it is again the case, the proof is then calculated by using the **proof_of_work_rec** function and its value is added to the block's field, **proof**. When everything is done successfully, we empty the open transactions list and return True but if something goes wrong, we return False.

- **Proof_of_work_env:** `Proof_of_work_env` is our recursive function. It is used to protect the chain against a mass attack as described above. This

function is an envelop function which returns **Proof_of_work**. The function `proof_of_work()` is recursive. This function initializes a variable `proof` with 0 and calls **valid_proof** recursively and each time increments `proof` until it returns True. What does **valid_proof** does? `Valid_proof` concatenates the open transactions, the last hash and the `proof` variable and returns the hash of the string formed. The two first values of this hash are verified, and we check if it's equal to two zeros, three or four depending on the time we want the mining process to take. When our condition is satisfied, we return True and when not, we return False. The **proof** number generated by **proof_of_work** is used to verify a block. This verification method is described below

- **Verify chain:** In order to mine a block, we should ensure that the chain we are having is valid. This is verified with the **verify_chain()** method. This method goes through every block in the blockchain and for each block, it concatenates the transactions in this block with the previous hash and the `proof` stored in the block. If this concatenated chain generates a hash with two, three, four or more leading zeros depending on our decided condition, then, this block is valid.
- **Load_keys() and load_data():** As described above, we have different data for our blockchain. These data are the blockchain, the open transactions, the public and private keys. When a new node is being added on the blockchain, we need to generate keys for them and initialize the blockchain and the open transaction list. This function is responsible for this and is made up of an error handling method. In the **try** block of these functions, we try to have access to the files called **wallet** for the function **load_keys()** and **blockchain** for the **load_data()** function which are stored on the server and contains our public key, private key, blockchain and open transactions in case it is already created. If this file doesn't exist (which is the case for a new user), the **except** block plays its role. It initializes the public and private key with the help of **generate_keys** and the blockchain with the genesis block and the open transactions as an empty list. In the other case, if it exists, it reads the file with the help of the **readlines** function of python and initializes the public, private key, blockchain and open transactions with the data stored in this file.
- **Save_keys() and save_data():** When we create our keys, we need to save them. **Save_keys()** just save them in a file. **Save_data()** save the blockchain and the open transactions each time they are modified. These are modified when we add a transaction and mine a block.

Libraries used

1. **Json**: JSON (Javascript object notation) is a package used to store or transfer some particular data structures in order to be used later or in other projects. It is also used to convert these data structures into strings and from strings back to these structures. It has two principal methods: **dumps** and **loads**, which are respectively used to convert data to string and back to what they were. JSON is used in our project to hash data which are not strings. It is also used to write our blockchain, open transactions, public and private key in a file. The responses when we add requests and routes are done with JSON but this time from the jsonify package of Flask.
2. **Hashlib** : Hashlib contains different hashing methods. The one we used is SHA256 because it is the most stable today. We need a hash function when hashing our blocks during the mining and the verification process. The hash function is also used to generate a signature.
3. **Pycryptodome** : Pycryptodome is a package used in python to generate private and public keys. These keys are generated as explained above in **generate_keys()**.
4. **PKCS1_v1_5** : This package is contains methods such as **sign()** and **verify()** which generates respectively a signature and verifies later if this signature and the data stored are correct.
5. **Binscii** : Binscii is used to convert binary data types from binary to ascii strings and vice versa. This is used when returning the generated keys and signatures, because they need to be returned as strings and not as binary data. When these keys are verified, they need to be in binary form again, and hence need Binscii once more.
6. **Flask** : Flask is a framework and a class of python which is indispensable when dealing with requests and routes. The Flask package was used to launch functions written in our blockchain file from an html page and javascript code. The html, css and bootstrap were used to design the page as shown below. Javascript was now used to insert events on buttons which will trigger some functions dans will establish our request and receive the responses. The main method used in javascript was **axios** as well as try and catch blocks to collect possible errors and respond correctly them.
7. **Request** : Request was used to establish POST requests that required an additional data. For example, a **mine** request (request to mine a block) is

a POST request (since we are adding data to our server). But to mine, we just need to call this function and it is all. To add a transaction for example is also a POST request, but here we need to say who is the recipient and what is the amount. The **request** method of Flask allows us to that.

8. Argparse(): Used to get the port on which to run a node.

Our application in action

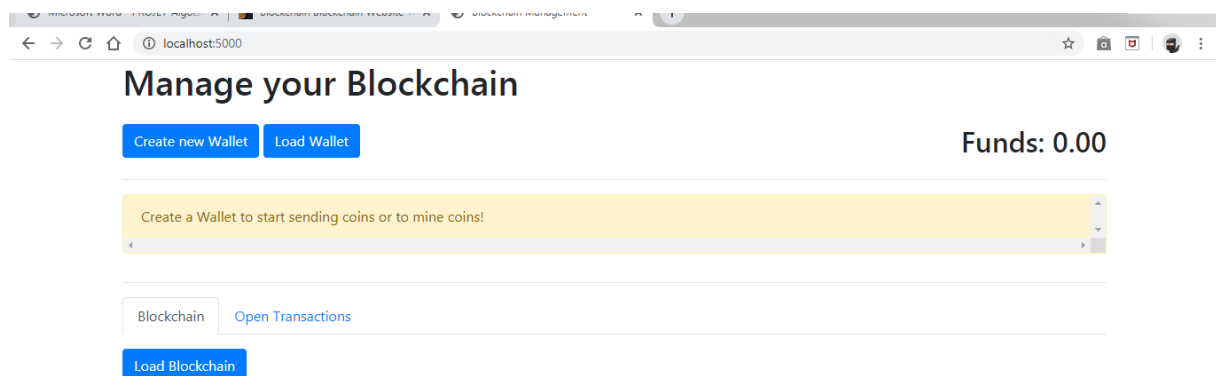
Starting a node.

To start a node, we can pass through the command line, then go to the location of our **node.py** file and launch it indicating the port on which to run with the option -p.

```
C:\Users\Christopher>cd desktop/blockchain
C:\Users\Christopher\Desktop\blockchain>node.py -p 5000
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Launching the web interface

As seen on the image, our application is running on the address <http://0.0.0.0:5000>. We type this on our navigator's address bar and obtain the page below.



As we can read on the image, we should first create a new Wallet. This will consist of generating a public and private key for this node. When everything goes fine, we have the image below on the green board, telling us that the wallet has been created successfully.

The screenshot shows a web browser window with the address bar at localhost:5000. The page title is "Manage your Blockchain". A green message box displays the following text: "Created Wallet! Public Key: 30819f300d06092a864886f70d0101050003818d0030818902818100b8997ca6789def27f91cb42555ce613c51c334fabe0098e8eb1816645436930b0c704 Private Key: 3082025d02010002818100b8997ca6789def27f91cb42555ce613c51c334fabe0098e8eb1816645436930b0c704f5d478506a9741f0c35550df8eb57a97269c". Below the message box are two buttons: "Create new Wallet" and "Load Wallet". To the right of these buttons, the text "Funds: 0.00" is displayed. Below the buttons is a form with two input fields: "Recipient Key" (containing "Enter key") and "Amount of Coins" (containing "0"). Below the "Amount of Coins" field is a small text note: "Fractions are possible (e.g. 5.67)". Below the form is a "Send" button. At the bottom of the page, there are two tabs: "Blockchain" and "Open Transactions". Below the tabs are two buttons: "Load Blockchain" and "Mine Coins".

We then have a form that can allow us to operate a transaction.

On the image below, we see that our fund is 0. If we try to add a transaction this will fail as shown below because our funds are insufficient.

The screenshot shows the same web browser window as the previous one. The page title is "Manage your Blockchain". A red message box displays the following text: "Error during the adding of the transaction". Below the message box are two buttons: "Create new Wallet" and "Load Wallet". To the right of these buttons, the text "Funds: 0.00" is displayed. Below the buttons is a form with two input fields: "Recipient Key" (containing "Michael") and "Amount of Coins" (containing "2"). Below the "Amount of Coins" field is a small text note: "Fractions are possible (e.g. 5.67)". Below the form is a "Send" button. At the bottom of the page, there are two tabs: "Blockchain" and "Open Transactions". Below the tabs are two buttons: "Load Blockchain" and "Mine Coins".

In order to have an amount, we should mine the block, which will inturn reward us with 10 coins.

The screenshot shows a web browser at localhost:5000 with the title "Manage your Blockchain". A green notification bar at the top states "adding block succeeded". Below this, there are two blue buttons: "Create new Wallet" and "Load Wallet". To the right, the text "Funds: 10.00" is displayed. The form includes a "Recipient Key" field with the placeholder "Enter key", an "Amount of Coins" field with the value "2", and a "Send" button. Below the form, there are tabs for "Blockchain" and "Open Transactions", and buttons for "Load Blockchain" and "Mine Coins". At the bottom, a grey bar shows "Block #0".

We see that the block has been added successfully and the funds are updated to 10 coins.

Let us now repeat the sending operation and send 2 coins to Michael.

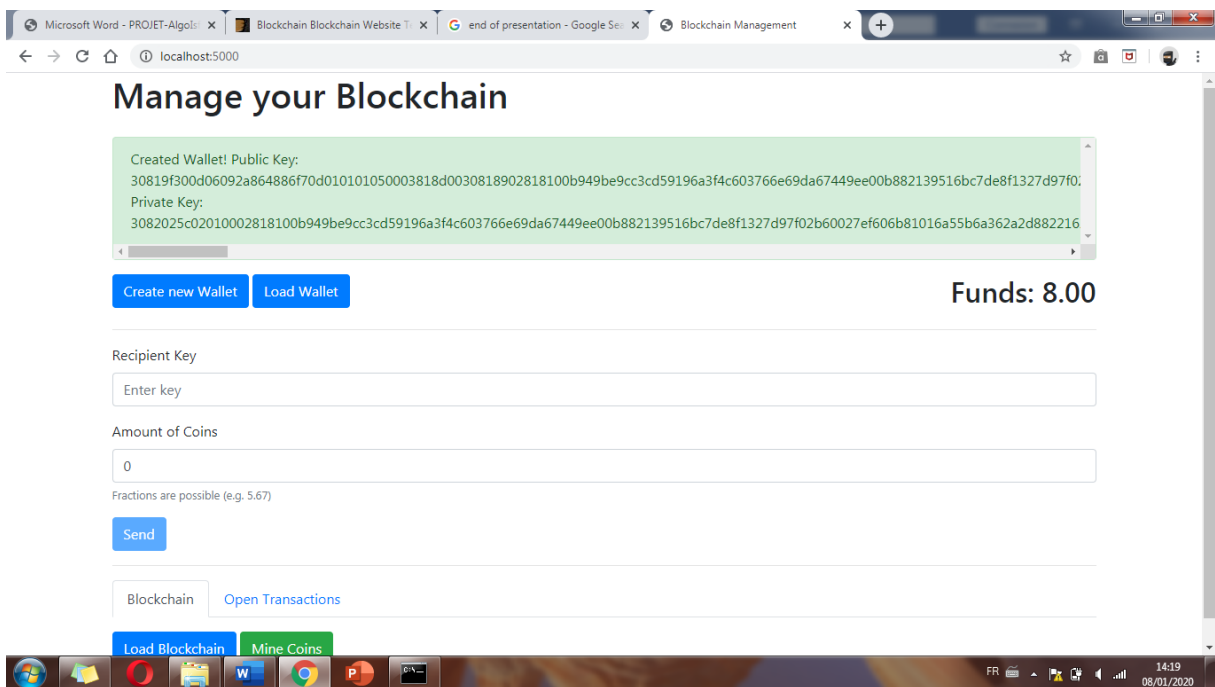
The screenshot shows the same web application after a transaction. The green notification bar now states "The amount has been added successfully". The "Funds: 8.00" text is updated. The "Recipient Key" field now contains the name "Michael". The "Amount of Coins" field still shows "2". The "Send" button is still present. The "Blockchain" and "Open Transactions" tabs, "Load Blockchain" and "Mine Coins" buttons, and the "Block #0" bar remain the same.

We see that the amount has been added successfully and our fund is now 8.

After adding this transaction, we can decide to mine the block once more so that it can enter in the blockchain. We can also perform more transactions before mining them. If we click on mine coins and have a look in the console or command line, we see how the proof of work is calculated. By looking the last hash, we see that it starts with 2 leading zeros.

```
Command Prompt - node.py
31a4630c383b36913b4cb14401ce945b96b2bff1ea28d37484b4a1c47f72603ef
17444219efec7ab14f8b7f3f0ce6c27726fd8ff4d1c8b2d1f975badb0f2820ca
be9375ce8343776550150acb44cecafe13ae0774438db7184a49bf7ad749871c
9657fc83078c197b4da40707e5d03d27811e645342eedce1cc393bf8d9f8434
dc9b9bbae60211d2c23f632858edd6ec3935daeb63c39608684f1286b400114e
aab5d6247251f5af6117fa7acb139799319dcd82d431f71a6274926ac73398a9
7544dd26f78596f98b4888c09c9e0dfce6cdca45c9bc5e8a8c12b65f1b72909e
803b445894927c5cbbfe37a762e6a905c76acdb8dbdb3f88b7c357b18c54a6fa
a44989736a5f924297bc79af6d7f442388a3e35889b37e9f9455789abaeff8f5
8d35ba002d2b00cf3f7063f75341a89ac73d4e2ac5a66a500f33536c23cc426f
50254757cadb3baf8aca4c097467cabe776dbc0e84d3dfbbf35bba13f1c4f395
cc877f6d47958824742a1bcd03dd12988c1408f7815d6447a43ea5100924612
2be65dbba6d7a8b7f35d902550c2ef3b8e8f970df38d19945db93dace3e9e889
6d48c0deee020a7fe7a4d621582e52ca472c208da6304eeeb59723a0afe0b6d0
5bac450a2c253db52caf828a6aa856bd4685a3d1a1d3a409d65982040ef84c19
e12daac30c66dcbb511a2bf05634f176cc52ceadc2745780abb03716be6113c
7c68f5e3b32349a87305524a4976be1acbab782f2349553fffdd832d7ac608180
a018eef4fcbec6265e67de02b4ddc231119e91c79325987cee855c3e0205ee81c
243f8a2ff232a9a09b257462cd6d9fb6e64eb920fa28b305604654acff11c04b
c1de60832512bfec1f45372f5ab75189e1f45f5571e8418542cc39905c8d88fe
0c8dc2ccaa8986c85acf749503646ba6b47f8c5a67887cb1273b7d3970f56832
0098b02fc535df4c4d3916ff14698f624b8e4eea5d7e55866953303185c58abf
127.0.0.1 - - [08/Jan/2020 14:28:19] "POST /mine HTTP/1.1" 201 -
127.0.0.1 - - [08/Jan/2020 14:28:23] "GET /chain HTTP/1.1" 200 -
```

If we quit and launch the node again through the command line on the same port, launch the web page, then click on **load wallet**, we get our pair of public and private keys, the funds and everything back as we left.



Our blockchain confronted to security measure

When we perform what ever operation we want to perform on the blockchain, it stores it in `blockchain_port.txt`. If a data in this file is modified, further mining of blocks is no more possible.

N.B: For the modifications made directly in the `blockchain_port.txt` file to be taken in consideration, we should relaunch the application.

Running multiple nodes

To run multiple nodes, we just need to launch the app in the console by giving a different port number with the `-p` option and typing the address in the address bar of the navigator.

For example: `node.py -p 5001`

In the navigator: `localhost:5001`

General URL: `localhost:port`

Problems encountered

There were important logical errors that prevented us from easily attaining our goals. These errors were being located thanks to the powerful text editor **visual studio code**, which gave us a debug mode which allows to execute our code step by step as in **python tutor** and to have the state of each variable at each step. This helped us enormously to locate and correct our errors.

Bibliography

- <https://hackernoon.com/learn-blockchains-by-building-one-117428612f46>
- <https://www.youtube.com/watch?v=7W7WPMX7arI>
- <https://docs.python.org/3/library/hashlib.html>
- <http://user.oc-static.com/pdf/729324-creez-vos-applications-web-avec-flask.pdf>
- <https://flaviocopes.com/axios/>
- <http://user.oc-static.com/pdf/309961-dynamisez-vos-sites-web-avec-javascript.pdf>
- <http://user.oc-static.com/pdf/683140-prenez-en-main-bootstrap.pdf>
- <http://user.oc-static.com/pdf/13666-apprenez-a-creer-votre-site-web-avec-html5-et-css3.pdf>
- https://en.bitcoin.it/wiki/Proof_of_work
- <https://stackoverflow.com/questions/4232389/signing-and-verifying-data-using-pycrypto-rsa>
- <https://themeforest.net/tags/blockchain?term=blockchain>