# Exercises: Build Your Own Blockchain

In this exercise you will learn how **Blockchain networks** work by building a simple one in Python. **Blockchain** is an immutable, sequential chain of records called **blocks**. Blocks can hold **transactions**, documents, files or any data you like, really, but the important thing is that blocks are chained together using hashes. The blockchain networks consists of interconnected **nodes** which update the blocks together using a **consensus algorithm** (like proof-of-work). Original source: <https://hackernoon.com/learn-blockchains-by-building-one-117428612f46>.

You will need **Python 3.6+**, **Python IDE** and HTTP client like **Postman** or **cURL**. Also, you will need to install the **Flask** Requests Library for Python.

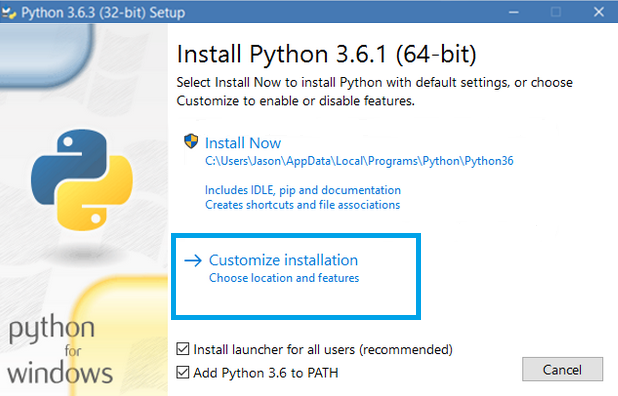
## Install Python and Flask

To **install Python** and all required libraries in Windows, use the steps below:

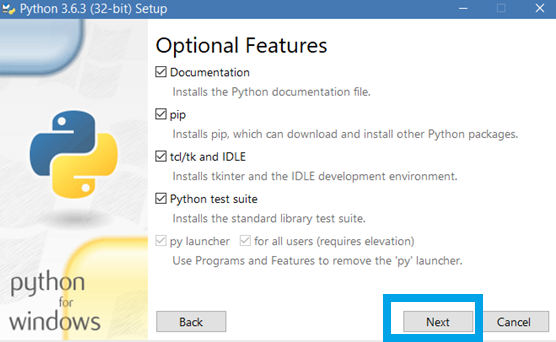
* 1. If you have installed Python, you can skip this step. If not, go to python website: <https://www.python.org/downloads/> and download “**Python 3.6**” or later. Then install it.



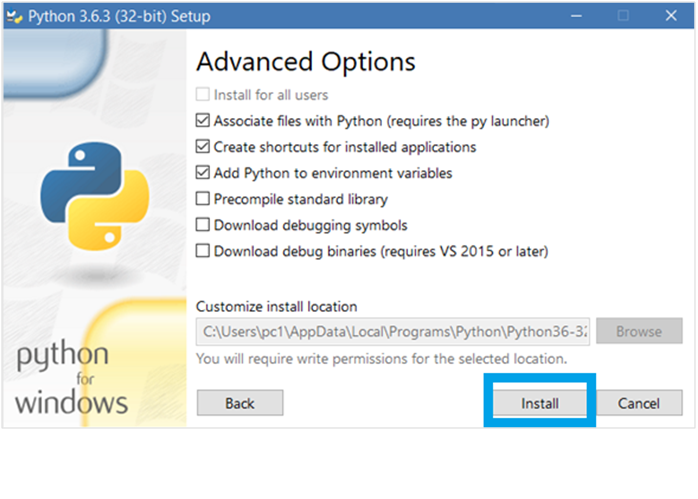
* 1. Choose the [Customize installation] link.



* 1. Check all the options and click [Next].



Check also the **“Add Python to environment variables”** (by default it is not checked) and click [Install].



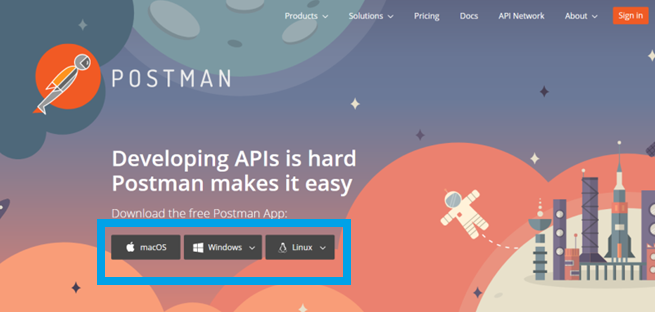
* 1. After completing open Command Line Interpreter and write the command: “**python -V**”. You should see the Python version. Note the capital “**V**”.
  2. Then install the “**Flask**” library using the command:

|  |
| --- |
| pip install Flask==0.12.2 requests==2.18.4 |

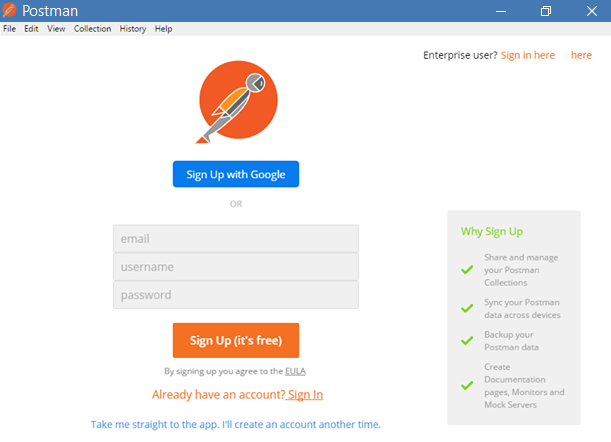
## Install Postman

We will need also **Postman** HTTP client as we mention it above.

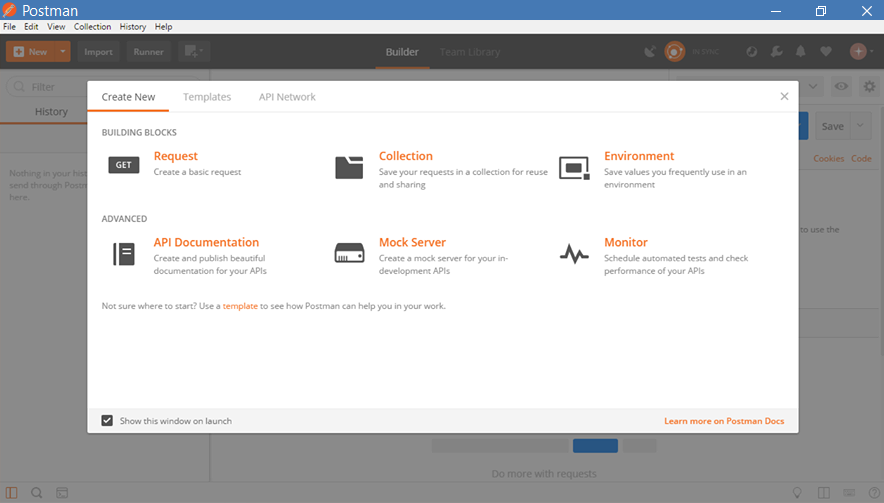
1. Go to the official Postman web site: <https://www.getpostman.com>, download and install Postman.



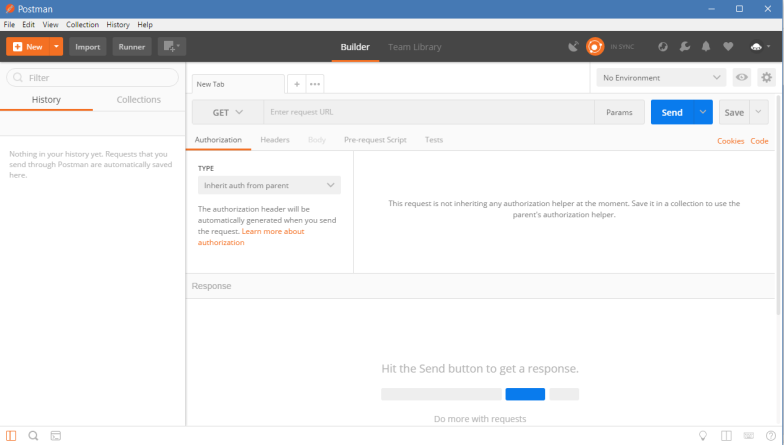
1. After installing it, you should sign up (it is free).



1. After signing up you will see the following window. You can close the inner one.



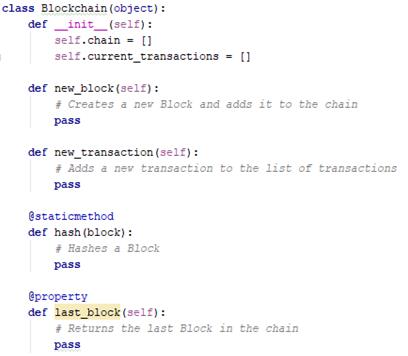
1. Finally you should see window like this one below:



1. Now we have all we need to start writing and testing the code.

## Building Our Blockchain

1. Firstly, open your favorite text editor or Python IDE (e.g. **PyCharm**).
2. Create a new file, called “blockchain.py**”**.
   * If you are new to Python, remember to **never mix tabs with spaces**.
   * You may get errors like this “TabError: inconsistent use of tabs and spaces in indentation”.
   * Use **4 tabs** for indenting code blocks.
3. We will create a **Blockchain** class whose constructor creates an initial **empty list** to store our **blockchain**, and another to store **transactions**. Here is how it will look like:

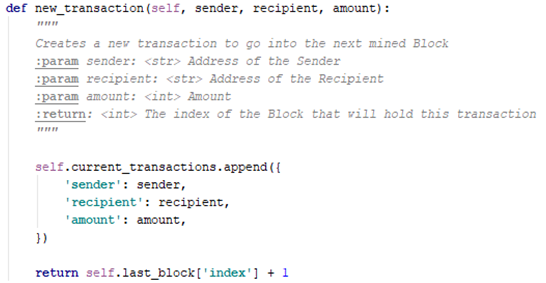


1. This is how does a **block** looks like:



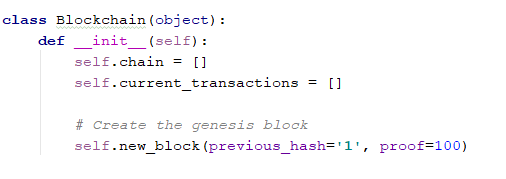
Each **Block** has an **index**, a **timestamp**, **a list of transactions**, **a proof** and the **hash of the previous Block**.

1. Now we will create the method new\_transaction() which will add the new transactions to the block.

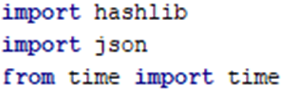


After new\_transaction() adds a **transaction** to the list, it returns the index of the block which the transaction will be added to—the next one to be mined.

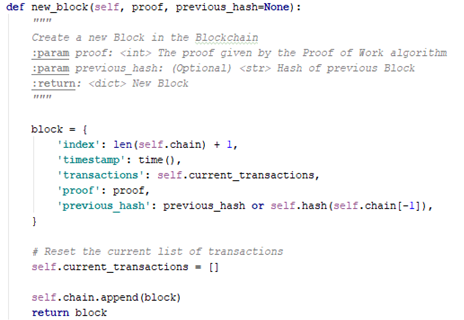
When our **Blockchain** is instantiated, we’ll need to seed it with a **genesis block**—a block with no predecessors. We’ll also need to add a “proof” to our genesis block which is the result of mining (or proof of work).



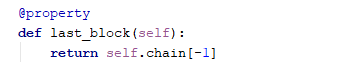
1. In addition to creating the **genesis block** in our constructor, we’ll also flesh out the methods for new\_block(), new\_transaction() and hash(). Now we will create the new\_block() and hash() functions.
2. Firstly we should import some libraries to the class.



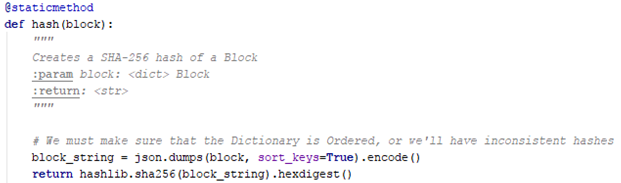
1. Then we will implement the new\_block() function.



1. We now have blocks and we should implement **last\_block()** function to be able to see the last block.



1. And the hash() function should look like this:

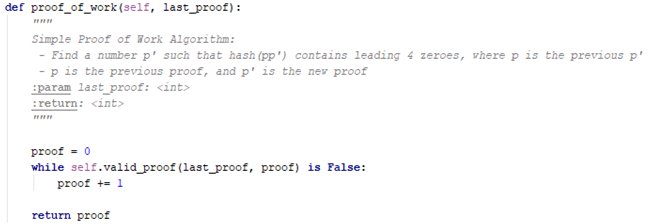


A **proof-of-work** algorithm is how **new Blocks** are created or **mined** on the **blockchain**. The goal of **PoW** is to discover a number which solves a cryptographic problem. The number must be difficult to find but easy to verify—computationally speaking—by anyone on the network. This is the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

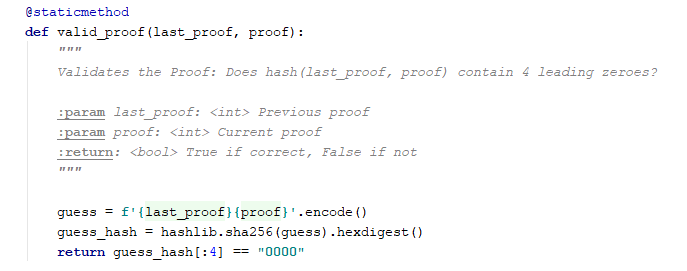
1. We should import:

|  |
| --- |
| from uuid import uuid4 |

1. Now the implementation of basic **proof-of-work** algorithm:



1. We will also need the valid\_proof() static method implemented.



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 are ready to begin interacting with it using **HTTP requests**.

## Exposing a Blockchain 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 API endpoints:

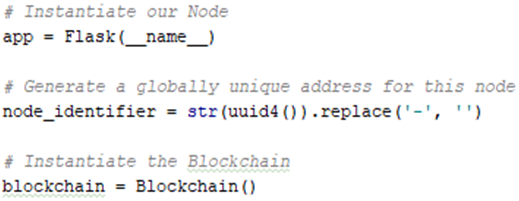
* /transactions/new – to create a new transaction to a block
* /mine – to tell our server to mine a new block.
* /chain – to return the full blockchain.

Let’s set up the **Flask** framework.

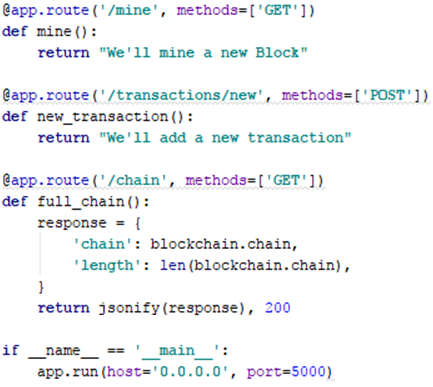
1. We need the following import:

|  |
| --- |
| from flask import Flask, jsonify, request |

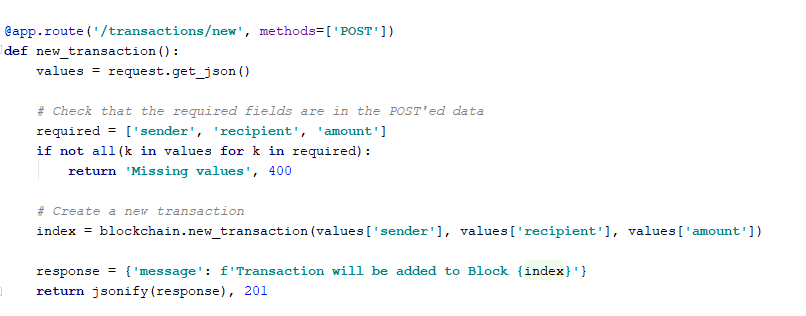
1. Firstly, we instantiate our **node**.
2. Then create a random name for our node.
3. And instantiate out the Blockchain class.



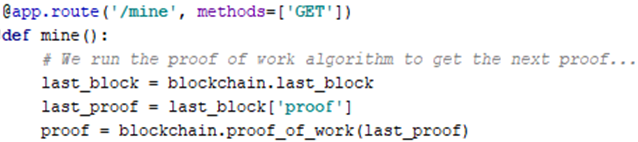
1. Create the /mine endpoint, which is a **GET** request.
2. Create the /transactions/new endpoint, which is a **POST** request, since we’ll be sending data to it.
3. Create the /chain endpoint, which returns the full blockchain.
4. Runs the server on port **5000**.



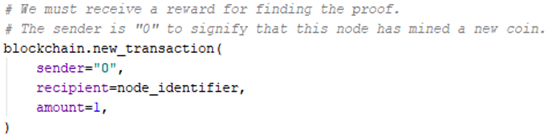
1. We will implement the new\_transaction()function:



1. It is time to implement the **mine()** function:
   1. Calculate the Proof of Work



* 1. Reward the miner (us) by adding a transaction granting us 1 coin



* 1. Add the new block to the chain and forget it



## Implementing a Consensus Algorithm

So far, 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**. Now we’ll have to implement a **Consensus Algorithm** if we want more than one node in our network.

1. Each node on our network should keep a **registry** of other nodes on the network. Thus, we’ll need some more endpoints:

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

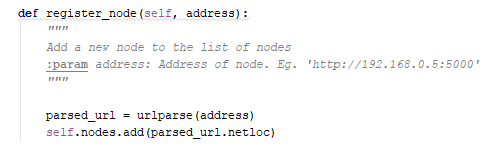
1. We’ll need to modify our **Blockchain’s constructor** and provide a method for registering nodes:
2. We should add to the constructor:

|  |
| --- |
| self.nodes = set() |

1. We should import:

|  |
| --- |
| from urllib.parse import urlparse |
|  |

And the register\_nodes()method:

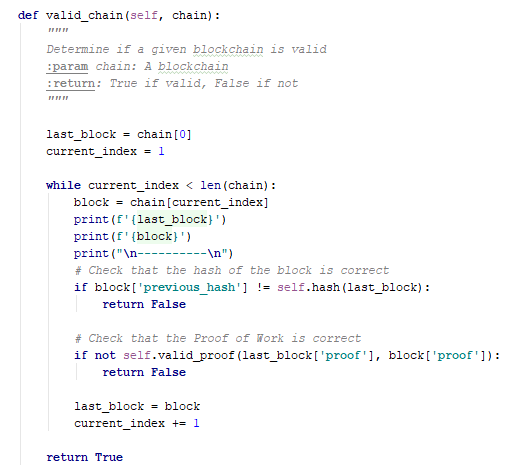


1. We should import:

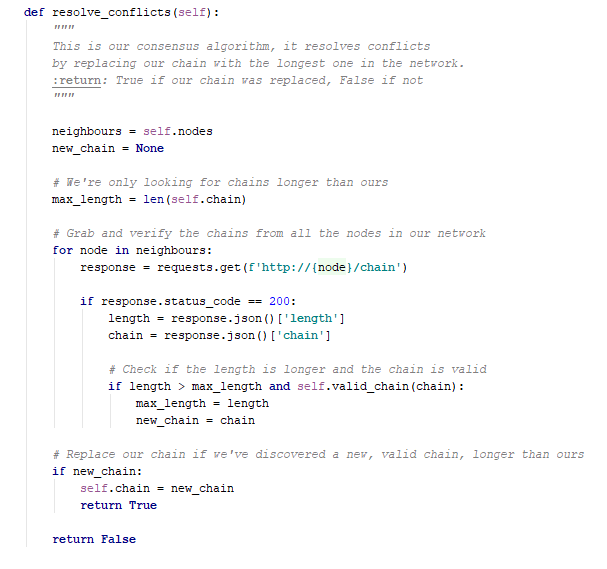
|  |
| --- |
| import requests |

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.

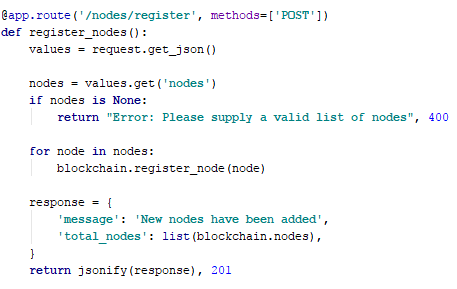
1. The first method valid\_chain() is responsible for checking if a chain is valid by looping through each block and verifying both the hash and the proof.

va

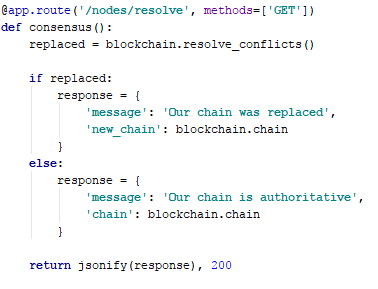
1. The method resolve\_conflicts() loops through all our neighboring nodes, downloads their chains and verifies them using the above method.
   * If a valid chain is found, whose length is bigger than ours, we replace ours.



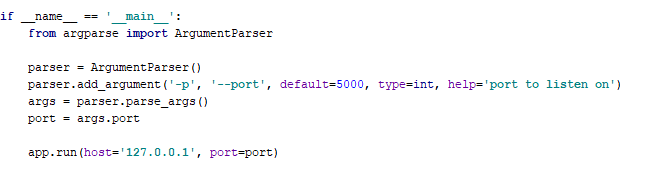
1. Let’s **register** the two endpoints to our **API**, one for adding neighboring nodes and the another for resolving conflicts:
2. First the register\_nodes() function:



1. Then the consensus()function:



1. Finally, to run the app you will need this method:



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**.

* It is easier to spin up another node on your machine, on a different port, and register it with your current node.
* Thus, you will have two nodes: <http://localhost:5000> and <http://localhost:5001>.

## Interacting with our Blockchain

1. Now open command shell and go to your project file directory.
2. Start your first blockchain node:

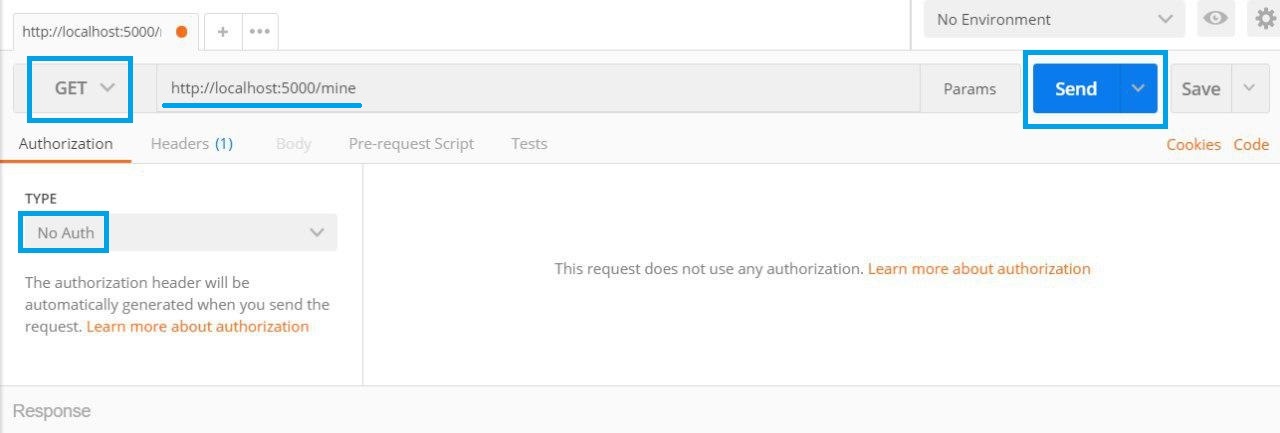
Run the command:

|  |
| --- |
| python blockchain.py |

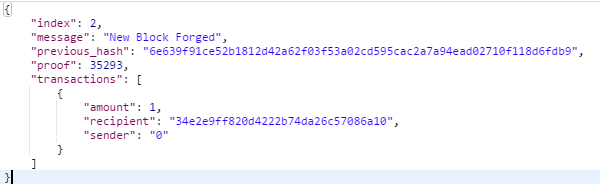
If everything is OK you will see the following result:

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

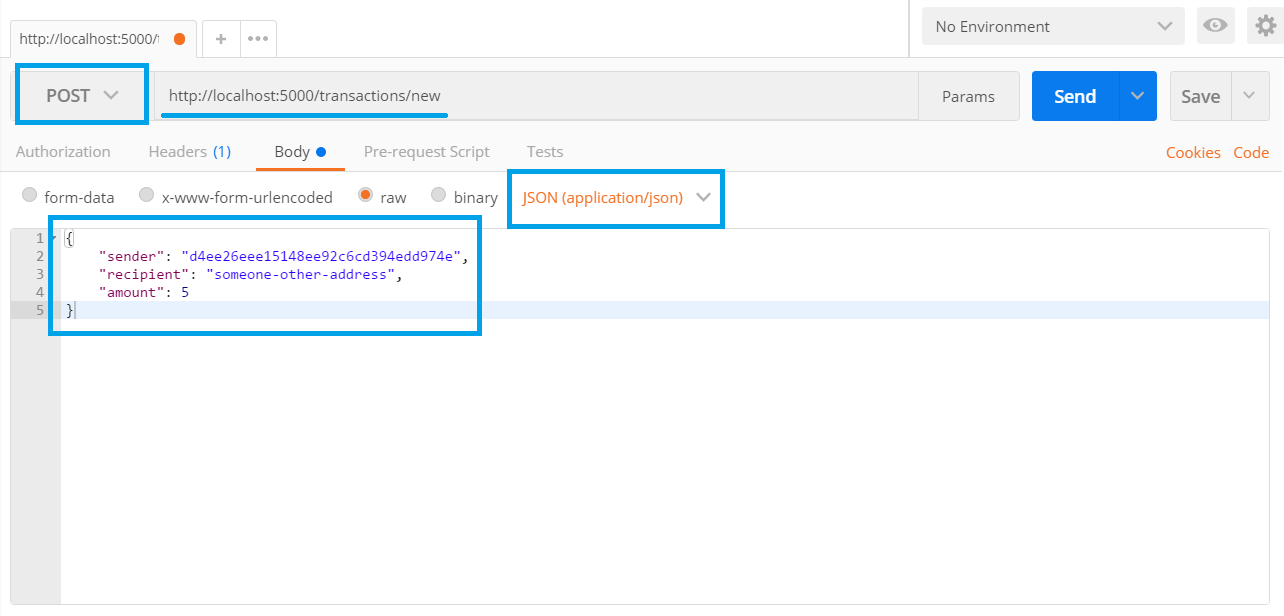
1. Now open **Postman**.
2. Let’s try **mining a block** by making a GET request to<http://localhost:5000/mine>. It should look like this:



1. The result should be:



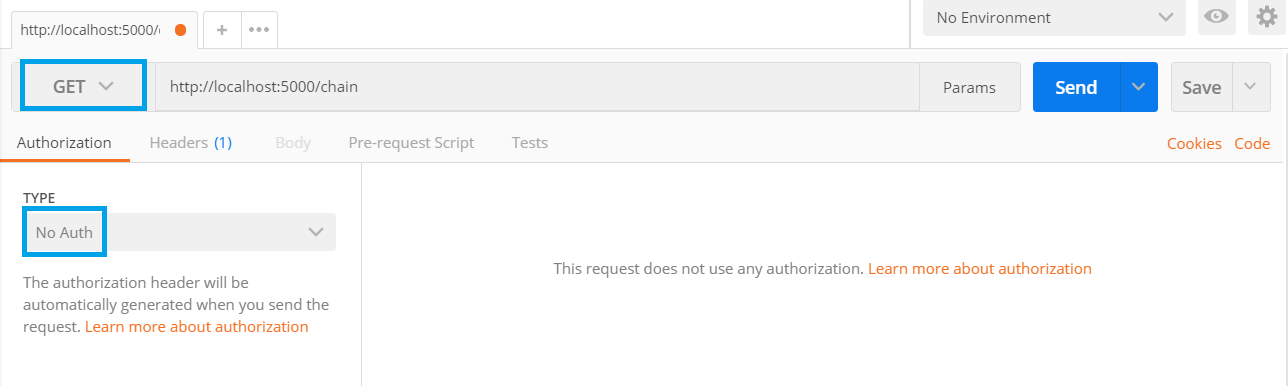
1. Let’s **create a new transaction** by making a **POST** request to <http://localhost:5000/transactions/new> with a body containing our transaction JSON structure:



1. The result should be:

asdqwreqweqwе.png

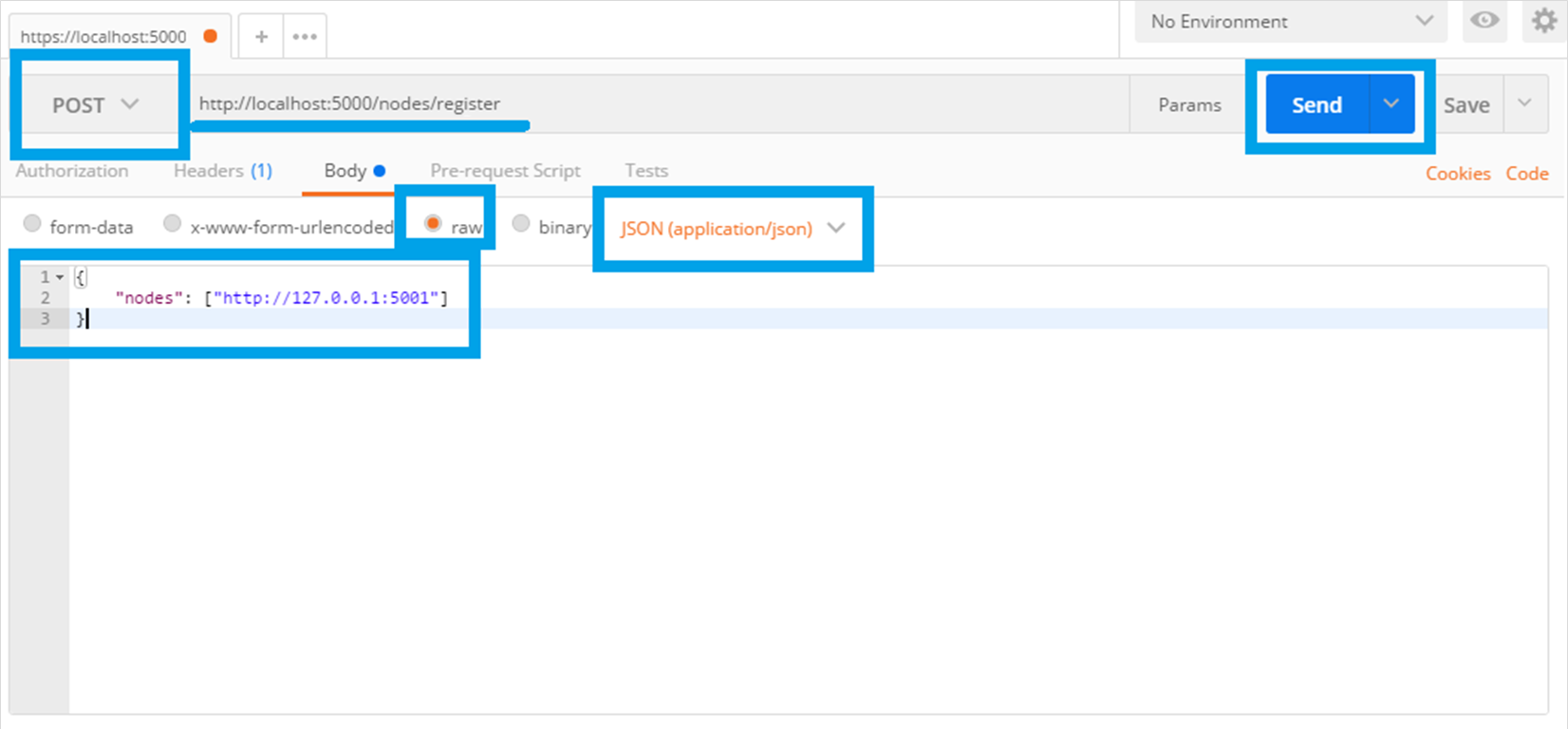
1. Let’s **inspect the full chain** by requesting <http://localhost:5000/chain>.



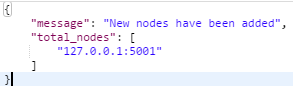
1. The result should be like this:



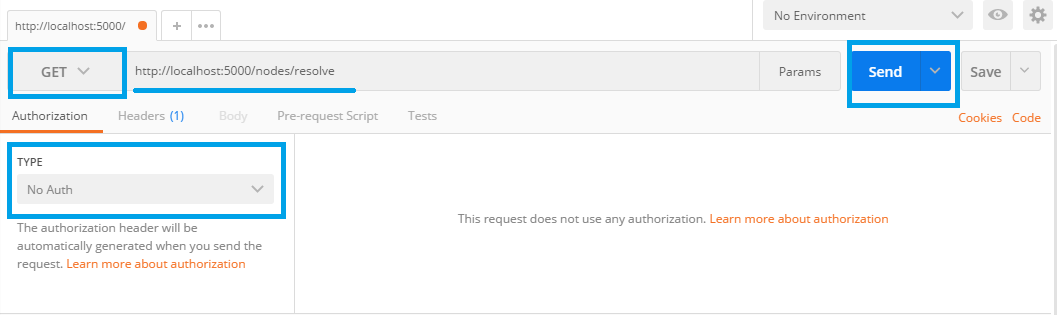
1. Register a new node:



1. **Do not forget** to run another node with port 5001 on your machine
2. The result should be:



1. Finally resolve conflicts:



1. The result should be:



# What to Submit?

Create a **zip file** (e.g. your-name-own-blockchain-exercise.zip) holding the source code files like **blockchain.py**.

Submit your zip file as **homework** at the course Web site.