BLOCKCHAIN

Explorando conceitos em uma blockchain simples

EPUSP, julho de 2022
PCS3544 – Prof. Dr. Marcos Antonio Simplicio Junior
Vinicius Augusto Carvalhaes Pimenta | NUSP: 10256363

Resumo — Neste roteiro autoguiado serão explorados alguns dos conceitos de blockchain vistos em aula através da utilização de um código pronto escrito em Python e requisições HTTP, que será o meio utilizado para "conversar" com a blockchain. O roteiro considera execução em ambiente Linux e usa o cliente HTTP Postman. É esperado que o aluno já tenha visto as aulas sobre blockchain e possua conhecimentos básicos de Python, APIs, JSON e operação de terminal Linux.

Palavras-chave — Blockchain, proof-of-work, hash, nonce, fork, API, Flask, Postman, Python, JSON

1. BREVE ANÁLISE DA IMPLEMENTAÇÃO

O código utilizado nesse roteiro é de autoria de Daniel van Flymen e pode ser encontrado em [1]. Nessa seção iremos analisar brevemente alguns métodos, estruturas e rotas do código contido no arquivo *blockchain.py*, bem as limitações da implementação. Para mais detalhes, consulte [2], artigo escrito pelo próprio autor do código.

Métodos e estruturas:

Como primeiro ponto, observe o método *new_block* da classe *Blockchain*. Nele é listada a estrutura que é utilizada em cada bloco. Note que cada bloco contém seu índice, um carimbo de tempo, a lista de transações, o número associado ao proof-of-work e o hash do bloco anterior.

```
110         block = {
111             'index': len(self.chain) + 1,
112             'timestamp': time(),
113             'transactions': self.current_transactions,
114             'proof': proof,
115             'previous_hash': previous_hash or self.hash(self.chain[-1]),
116         }
```

Em seguida, observe o método *new_transaction*. Nele é informada a estrutura de uma transação, que contém o remetente, o destinatário e a quantia a ser transferida. Todas as transações criadas são inseridas no próximo bloco minerado.

```
133         self.current_transactions.append({
134             'sender': sender,
135             'recipient': recipient,
136             'amount': amount,
137         })
```

Agora observe que o cálculo do hash do bloco anterior é feito pelo método *hash* e a função de hash utilizada é a SHA-256.

```
154         block_string = json.dumps(block, sort_keys=True).encode()
155         return hashlib.sha256(block_string).hexdigest()
```

Como mecanismo de consenso, o código utiliza um proof-of-work simples, cuja regra é "encontre um número p tal que hash($p_{\text{bloco anterior}}$, p, $\text{hash}_{\text{bloco anterior}}$) contenha 4 zeros à esquerda". A dificuldade do algoritmo pode ser ajustada modificando-se o número de zeros à esquerda da regra.

Os métodos que implementam essa regra e encontram o número p são *proof_of_work* e *valid_proof*. Observe que a busca do "nonce" p é feita por tentativa e erro.

```
157     def proof_of_work(self, last_block):
...         ...
168         last_proof = last_block['proof']
169         last_hash = self.hash(last_block)
170
171         proof = 0
172         while self.valid_proof(last_proof, proof, last_hash) is False:
173             proof += 1
...         ...

178     def valid_proof(last_proof, proof, last_hash):
...         ...
189         guess = f'{last_proof}{proof}{last_hash}'.encode()
190         guess_hash = hashlib.sha256(guess).hexdigest()
191         return guess_hash[:4] == "0000"
```

Note também que a cadeia de blocos que cresce mais rápido ganha a "corrida do consenso". Isso é implementado através do método *resolve_conflicts*.

```
 67     def resolve_conflicts(self):
...         ...
 78         # We're only looking for chains longer than ours
 79         max_length = len(self.chain)
...         ...
 81         # Grab and verify the chains from all the nodes in our network
...         ...
 89                 # Check if the length is longer and the chain is valid
 90                 if length > max_length and self.valid_chain(chain):
 91                     max_length = length
 92                     new_chain = chain
 93
 94         # Replace our chain if we discovered a … chain longer than ours
 95         if new_chain:
 96             self.chain = new_chain
 97             return True
```

Rotas da API:

Em relação às rotas da API, que permitirão a comunicação com a blockchain, o código utiliza a framework Flask. A tabela a seguir sumariza quais rotas são implementadas, seu tipo e função.

Rota	Método	Função
/mine	GET	Realiza mineração do próximo bloco
/transactions/new	POST	Cria nova transação
/chain	GET	Permite visualização completa da blockchain
/nodes/register	POST	Registra lista de nós participantes
/nodes/resolve	GET	Executa algoritmo de consenso

Todas as requisições utilizam JSON, que é facilmente convertido na estrutura de dicionário da linguagem Python. Para a criação de novas transações (*/transactions/new*), o corpo da requisição deve conter a seguinte estrutura:

```
{
    "sender": "endereço_de_origem",
    "recipient": "endereço_de_destino",
    "amount": valor_da_transação
}
```

Cabe ressaltar que a rota */mine* cria uma transação de 1 moeda com endereço de origem "0", o que significa que o nó recebeu uma nova moeda, como recompensa por ter validado as transações de um dado bloco.

Já para o registro dos nós participantes na blockchain, o corpo da requisição deve conter (*portaX* é o número da porta utilizada):

```
{
    "nodes": [
        "http://localhost:porta1",
        "http://localhost:porta2",
        "http://localhost:porta3"
    ]
}
```

Limitações:

Conforme visto em aula, sabemos que a blockchain não faz validação de eventos em si, ficando essa tarefa a cargo da aplicação. Na implementação usada não são feitas validações, com isso, surgem dois problemas: como não é feito o uso de assinatura digital sobre as transações, qualquer nó pode tentar criar uma transação com origem qualquer e (2) como o saldo não é verificado, pode haver gasto duplo e registro de transações mesmo que não haja fundos.

Outra limitação relevante é que, após a execução do algoritmo de consenso, as transações de um fork temporário abortado não voltam para a *mempool*, ou seja, são descartadas.

2. INSTALAÇÃO DAS FERRAMENTAS

Nessa seção serão dadas instruções para instalação e configuração de todas as ferramentas necessárias para execução do roteiro. O SO usado foi o Zorin OS Core 16, mas você pode usar qualquer outra distribuição baseada em Ubuntu.

(1) Instale o Python:

```
sudo apt-get update
sudo apt-get install python3.8
```

(2) Instale o PIP, um gerenciador de pacotes para Python:

```
sudo apt install python3-pip
```

(3) Instale o venv, para permitir a criação de ambientes virtuais:

```
sudo apt install python3-venv
```

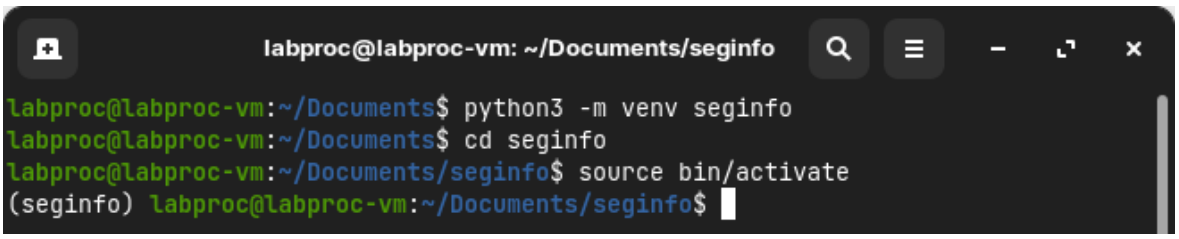
(4) Vá para a pasta na qual deseja criar o ambiente virtual (ex.: *Documents*) e execute o comando a seguir (chamaremos nosso ambiente de *seginfo*):

```
python3 -m venv seginfo
```

Em seguida, ative o ambiente virtual criado:

```
cd seginfo
source bin/activate
```

Após a execução desses comandos, seu terminal ficará prefixado com o nome do ambiente virtual criado, como na captura a seguir:



(*Nota: quando quiser sair do ambiente virtual basta executar o comando* `deactivate`)

(5) Instale o git, um sistema de controle de versão, e clone o repositório do projeto:

```
sudo apt install git
git clone https://github.com/dvf/blockchain
```

(*Nota: uma cópia do repositório é fornecida junto com o guia para o caso de indisponibilidade futura ou modificações que façam o roteiro não ser mais compatível com o código*)

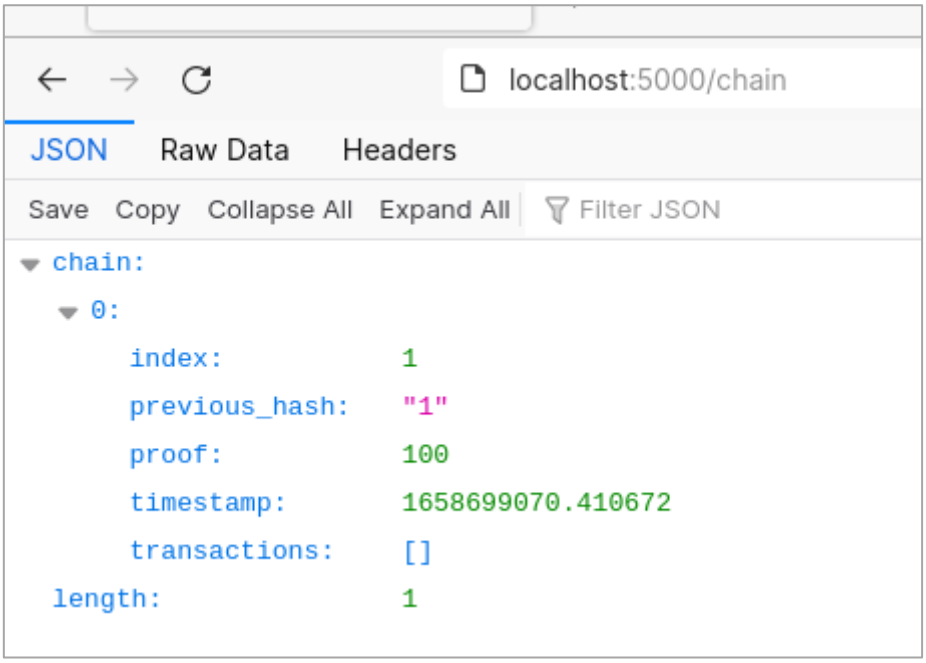
(6) Instale os pacotes Python necessários para execução do projeto:

```
pip install Flask requests
```

(7) Execute o projeto para verificar se tudo está funcionando corretamente:

```
cd blockchain
python blockchain.py
```

Em seguida, abra um navegador (recomenda-se o Firefox, por já possuir um visualizador de JSON integrado), e digite o endereço: http://localhost:5000/chain. Caso tudo esteja funcionando corretamente, você deve ser capaz de visualizar o bloco gênese da blockchain:



Por fim, pressione CTRL+C no terminal para encerrar o servidor.

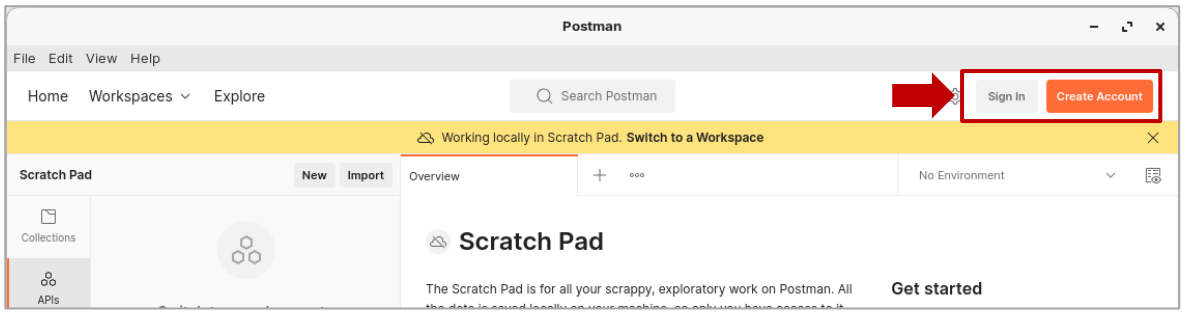
(8) Instale o Postman, uma plataforma de API que será usada para as requisições GET/POST:

```
snap install postman
```

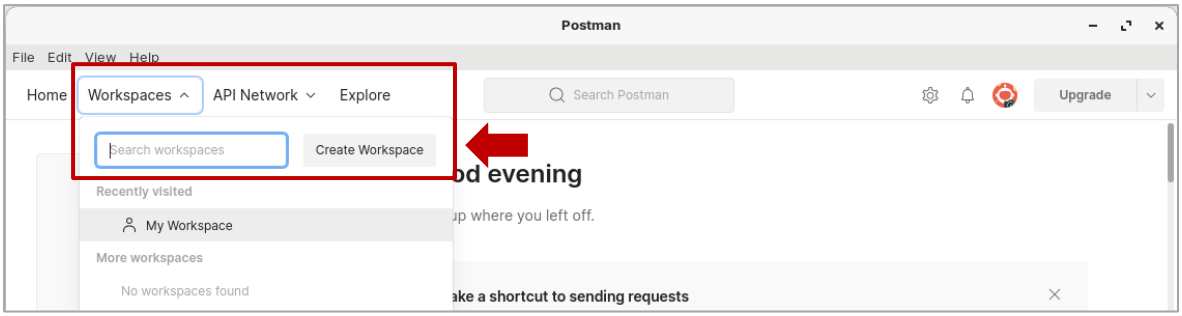
Em seguida, abra outro terminal e execute o postman:

```
postman
```

O programa será aberto, e você deverá realizar o login ou criar uma conta:



Após o login, crie um Workspace, que chamaremos de *seginfo*:

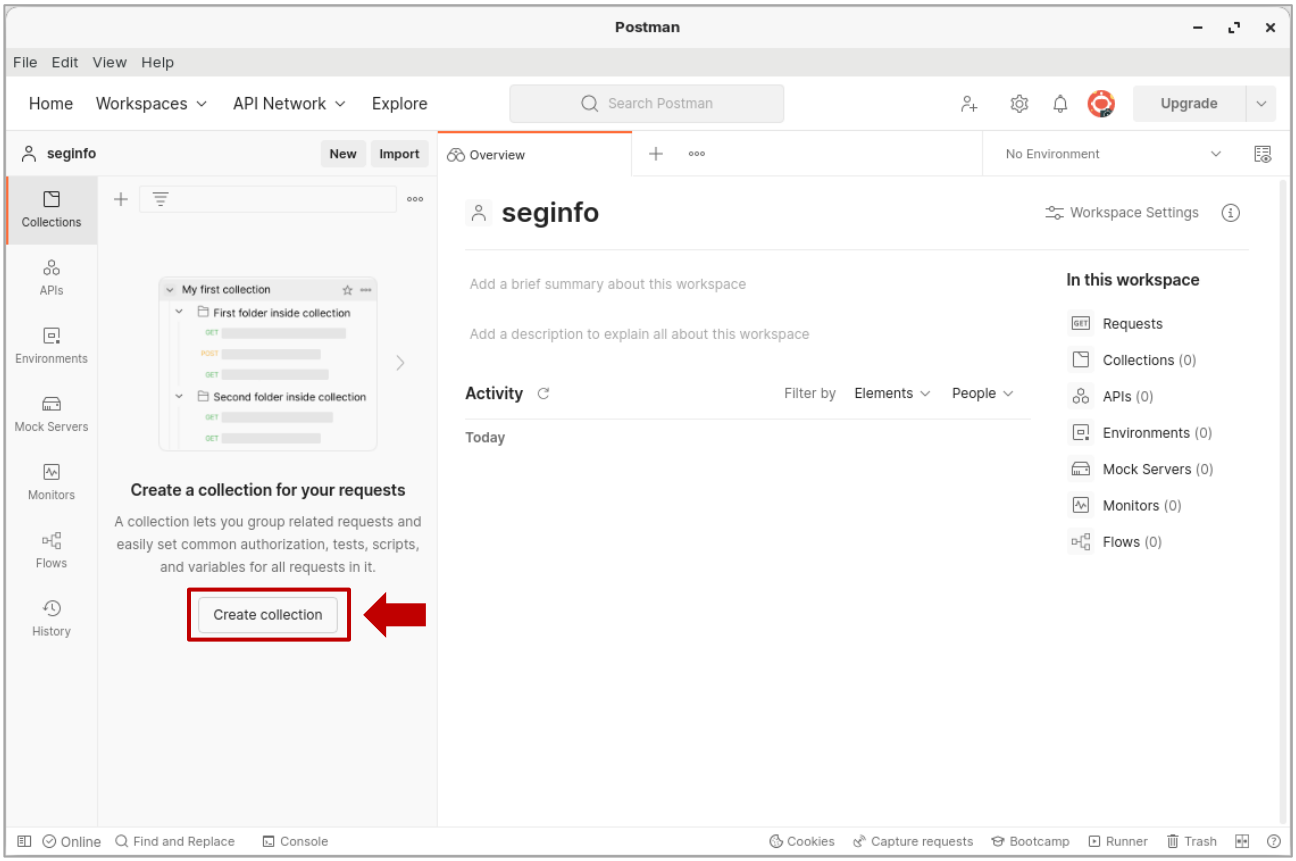


3. BLOCKCHAIN COM 1 NÓ MINERADOR

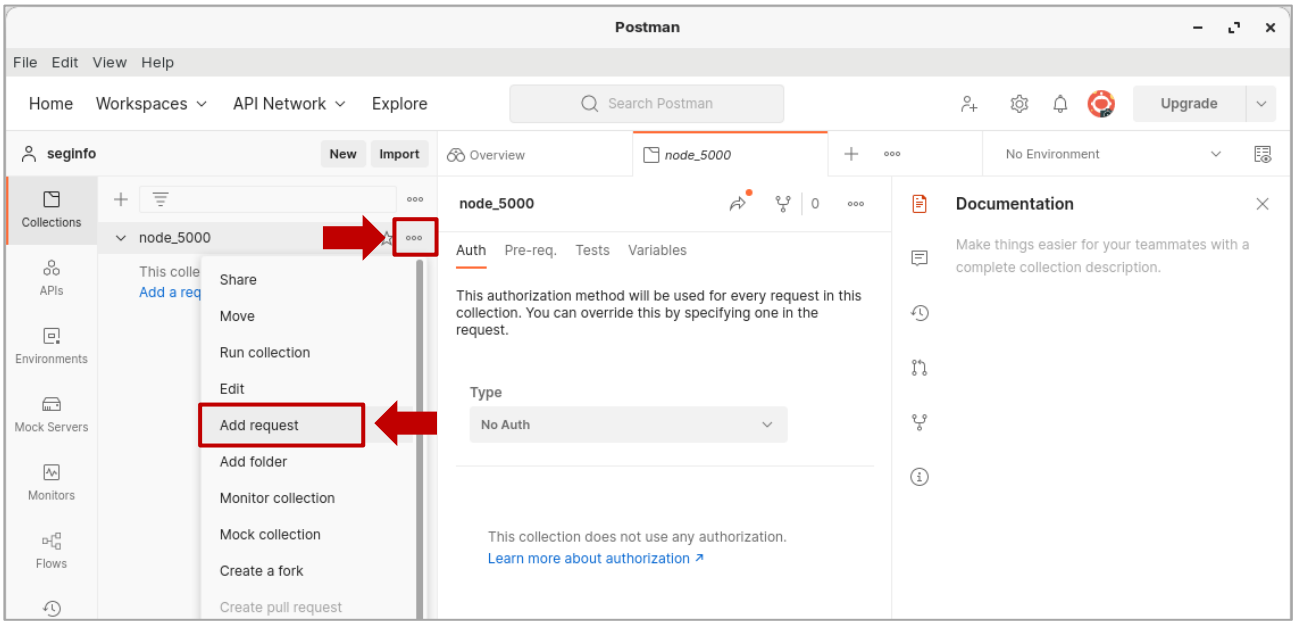
Começaremos com apenas 1 nó minerador a fim de verificar a operação básica de uma blockchain. Para isso, primeiramente execute o seguinte comando no terminal que está com o ambiente virtual ativado:

```
python blockchain.py --port 5000 &
```

Para as requisições do tipo POST, será usado o Postman, enquanto as do tipo GET serão observadas pelo navegador (você pode fazer ambos os tipos vis Postman se quiser). Para configurar uma requisição POST no Postman, primeiro crie uma coleção (chamaremos de *node_5000*):

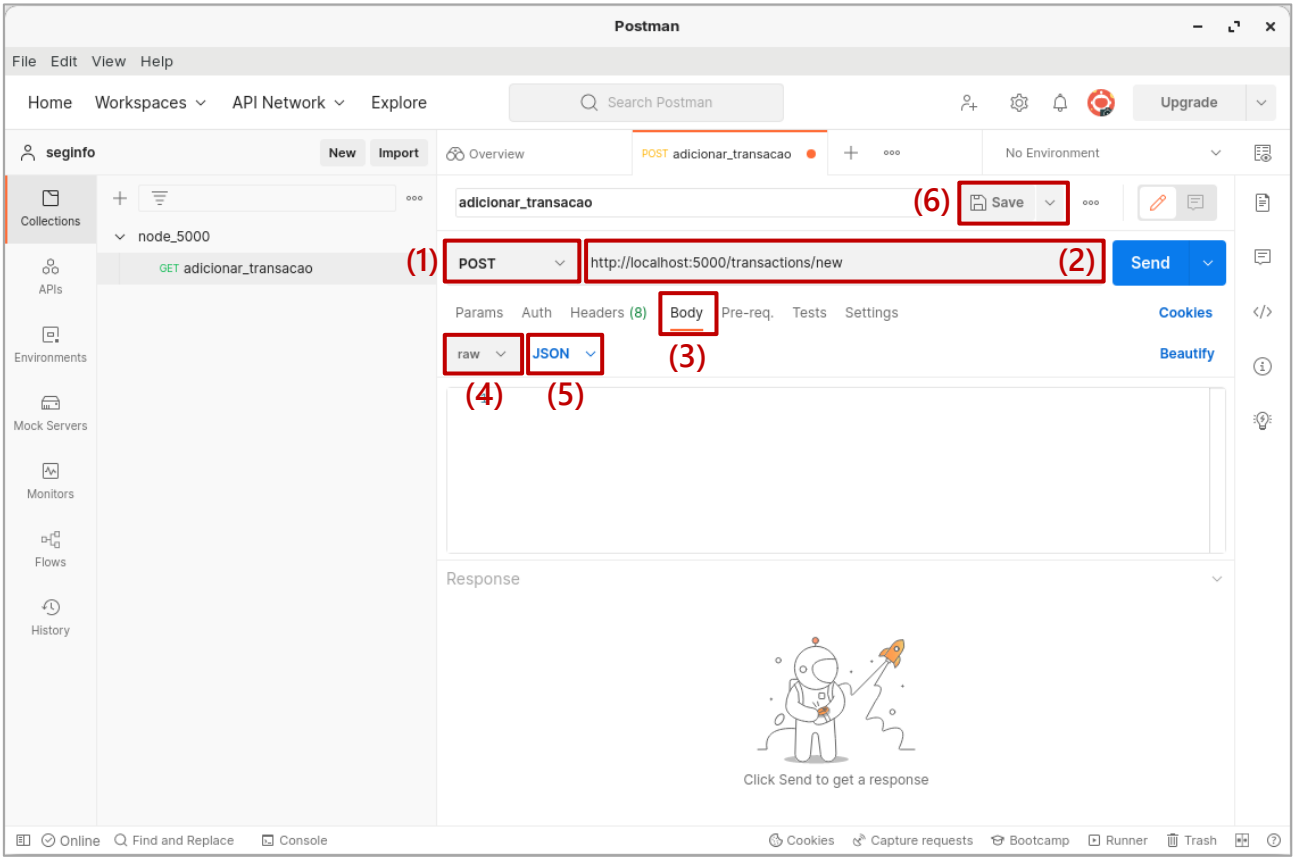


Em seguida, adicione uma requisição:



Nomeie a requisição (chamaremos de *adicionar_transacao*). Faça as seguintes modificações, conforme indicado na imagem a seguir:

(1) Altere o tipo de requisição para POST
(2) Adicione a rota da API para adição de transações: http://localhost:5000/transactions/new
(3) Selecione "Body"
(4) Mude para "raw"
(5) Mude para "JSON"
(6) Salve

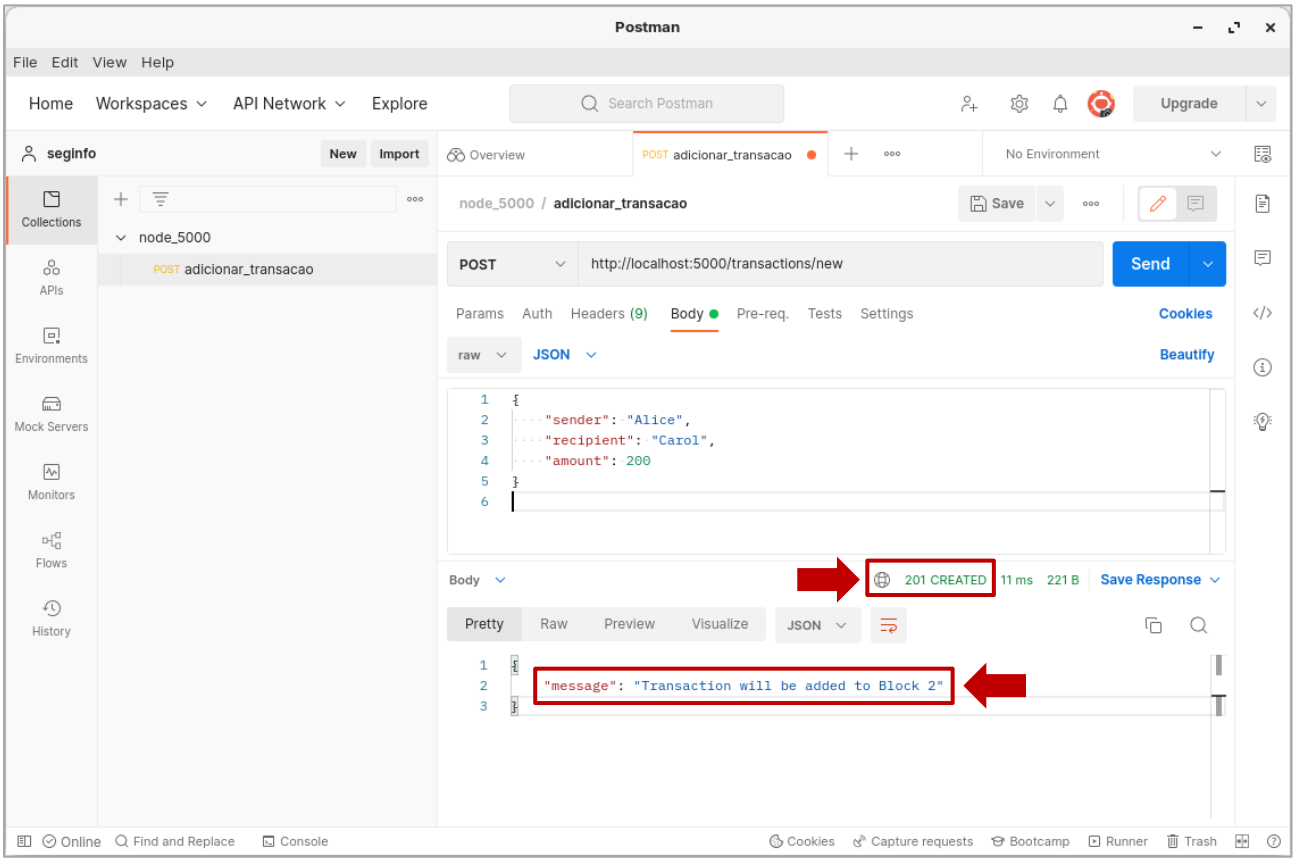


Adicione as transações a seguir e clique em "Send", uma por vez:

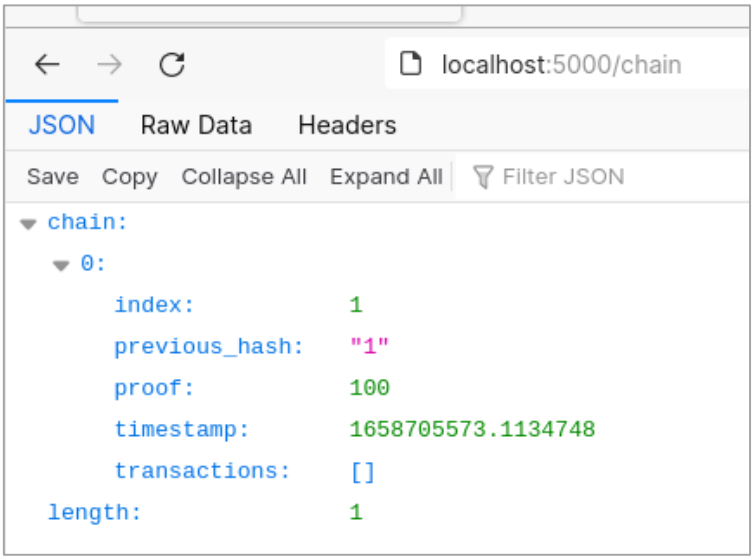
```
{
    "sender": "Alice",
    "recipient": "Bob",
    "amount": 100
}
```

```
{
    "sender": "Alice",
    "recipient": "Carol",
    "amount": 200
}
```

Caso a adição tenha sido bem-sucedida, você receberá o código de status 201 CREATED e uma mensagem informando que a transação será adicionada no próximo bloco. Para a segunda transação:



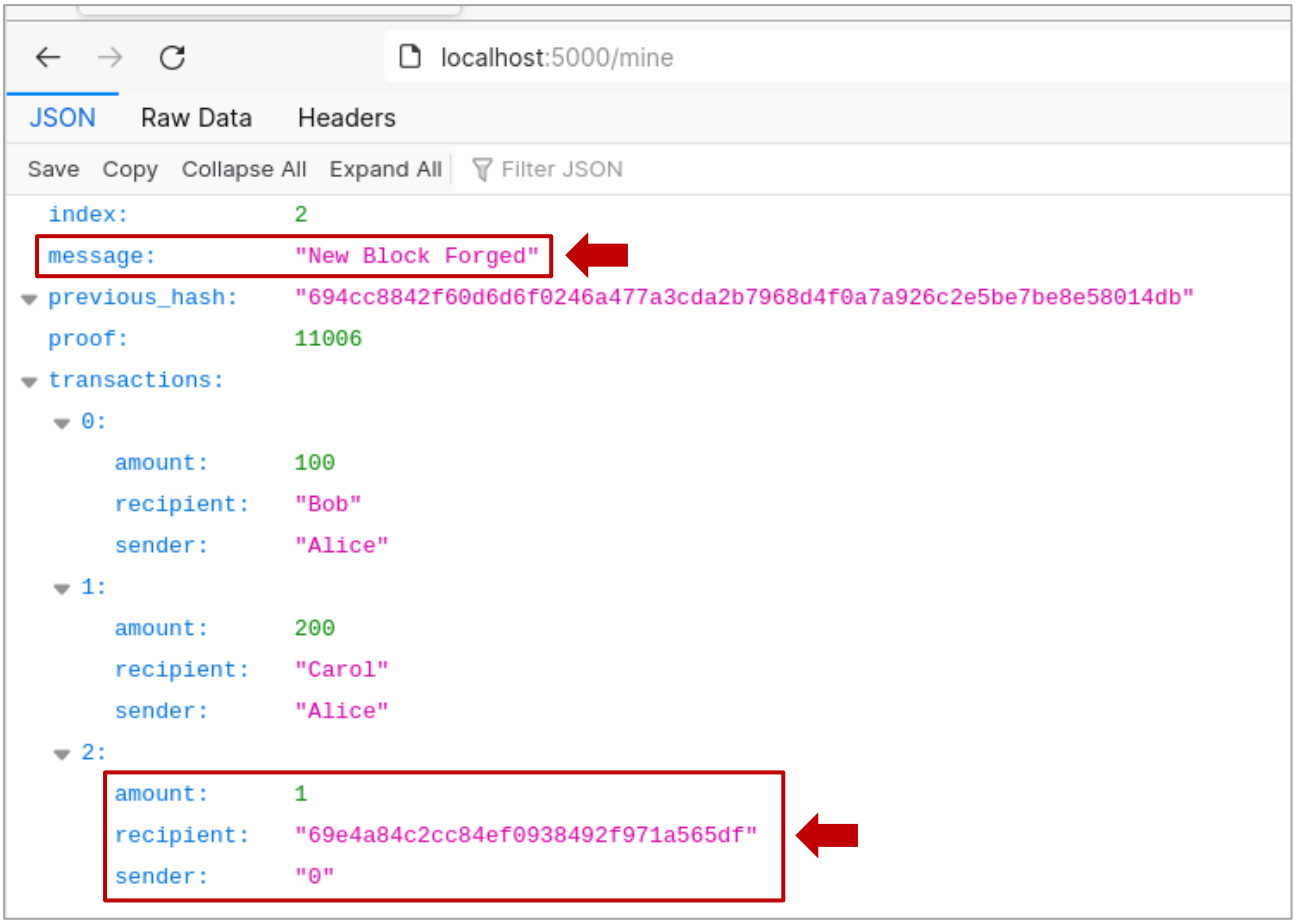
Agora veja quais transações estão registradas na blockchain, acessando no navegador o endereço http://localhost:5000/chain:



(*Nota: o carimbo de tempo 'timestamp' utiliza a hora atual do seu computador, então não será idêntico ao das imagens do roteiro. Como consequência, os parâmetros 'previous_hash' e 'proof' dos blocos seguintes também serão diferentes*)

Você deve ter percebido que as transações recém adicionadas ainda não constam na blockchain. Isso se deve ao fato de que elas estão na *mempool* desse nó, ou seja, na lista de transações não validadas.

Para validá-las, é necessário minerar o próximo bloco, acessando o endereço http://localhost:5000/mine. Note que a mensagem de resposta informa a respeito da criação de um novo bloco. É importante notar também que nesse novo bloco não há apenas 2 transações, mas sim 3, sendo a terceira a recompensa de uma unidade de moeda dada ao nó minerador por ter validado as transações do bloco:



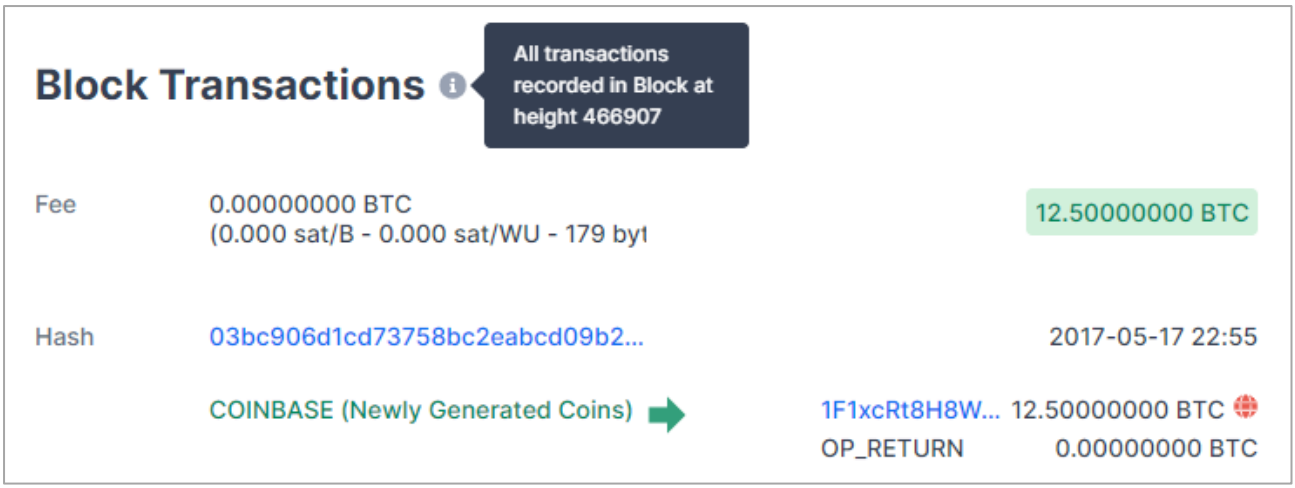
Verifique, acessando novamente http://localhost:5000/chain, que de fato agora as transações recém adicionadas estão registradas na blockchain.

Observe que não rodaremos o algoritmo de consenso para esse exemplo, pois não é necessário estabelecer consenso quando há apenas um nó na blockchain.

Minerando um bloco sem transações:

Tente minerar o próximo bloco (http://localhost:5000/mine) sem adicionar novas transações. Perceba que ainda assim a mineração é bem-sucedida e a única transação adicionada ao bloco é a recompensa do minerador.

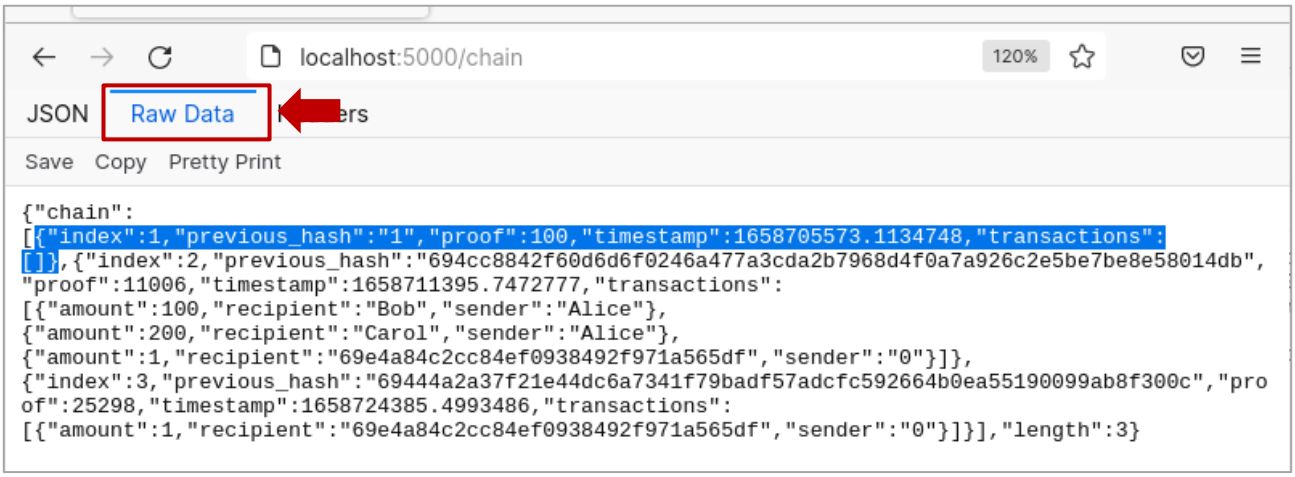
Note que isso não é uma característica exclusiva da implementação aqui utilizada. O bloco 466907 do Bitcoin, minerado em 17 de maio de 2017, por exemplo, não possuía transações, sendo composto apenas pela recompensa de 12.5 BTC do minerador [3]:



Verificando a validade da blockchain manualmente:

Vamos agora verificar a validade da blockchain criada, composta por 3 blocos, de forma manual. Para isso, é necessário verificar se as regras definidas na implementação estão sendo respeitadas, ou seja, se o hash informado no campo *previous_hash* é realmente o hash do bloco anterior e se a operação hash($p_{\text{bloco anterior}}$, p, $\text{hash}_{\text{bloco anterior}}$) produz um hash com 4 zeros à esquerda.

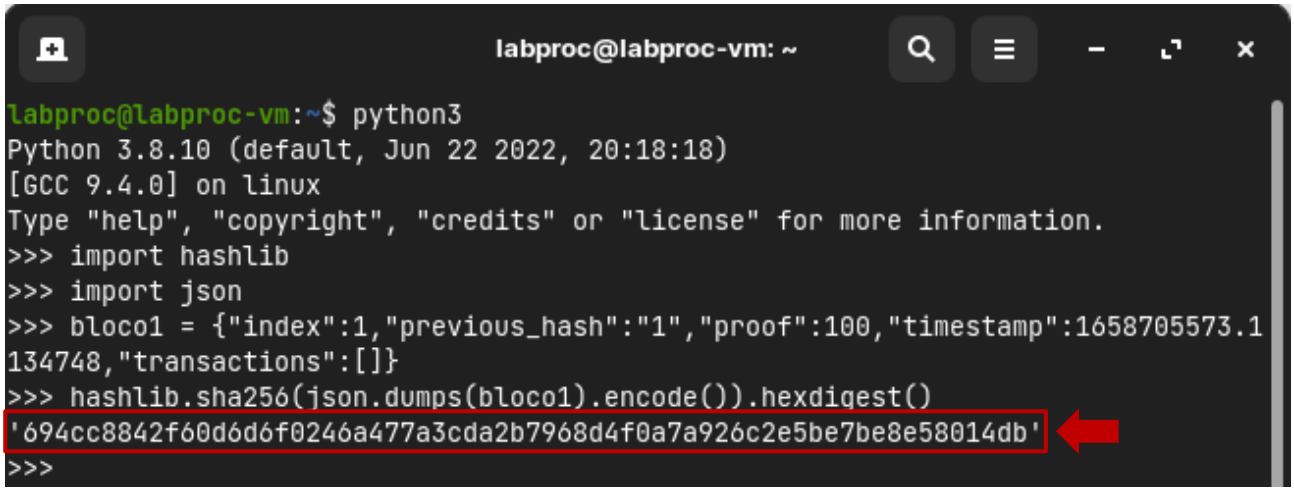
Para selecionar o conteúdo correto para cálculo do hash, basta mudar a forma de visualização dos dados para "Raw Data" e selecionar todo o conteúdo do bloco desejado. Para o bloco 1:



Usaremos um interpretador Python para as operações. Abra uma nova janela de terminal e digite o comando python3. Em seguida, execute os seguintes comandos para cálculo do hash do bloco:

```
import hashlib
import json
blocoX = conteúdo do bloco copiado do navegador
hashlib.sha256(json.dumps(blocoX).encode()).hexdigest()
```

Para o bloco 1, temos que o hash SHA-256 calculado é:

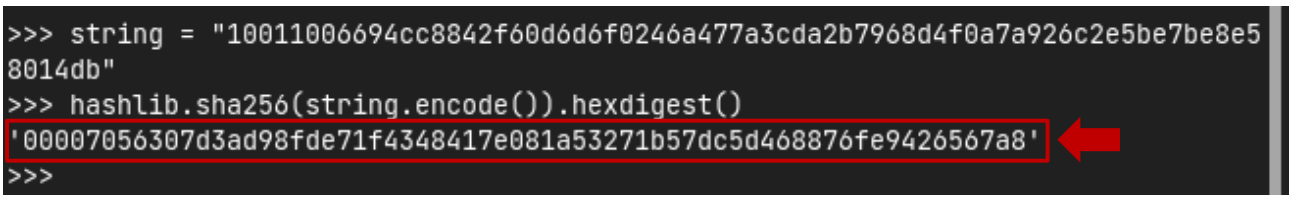


Note que o hash calculado é exatamente o mesmo indicado no campo *previous_hash* do bloco 2:

Já para a realização da operação hash($p_{\text{bloco anterior}}$, p, $\text{hash}_{\text{bloco anterior}}$), basta copiar os valores dos campos correspondentes e juntá-los em uma única string e, em seguida, executar os seguintes comandos no interpretador:

```
import hashlib
import json
string = “proof(bloco anterior) proof(bloco atual) previous_hash concatenados”
hashlib.sha256(string.encode()).hexdigest()
```

Para os dados do bloco 1 e bloco 2, note que o hash calculado de fato possui 4 zeros à esquerda:



Faça a mesma verificação para o próximo bloco para concluir que a blockchain é válida.

Por fim, observe que o modo como o nonce é calculado na implementação, que influencia no modo como a blockchain é validada, não é ideal, pois não se leva em consideração as transações do bloco atual. Com isso, se você alterasse as transações do último bloco, a blockchain permaneceria válida, sem necessidade de cálculo de um novo nonce.

4. CONSENSO E FORKS TEMPORÁRIOS

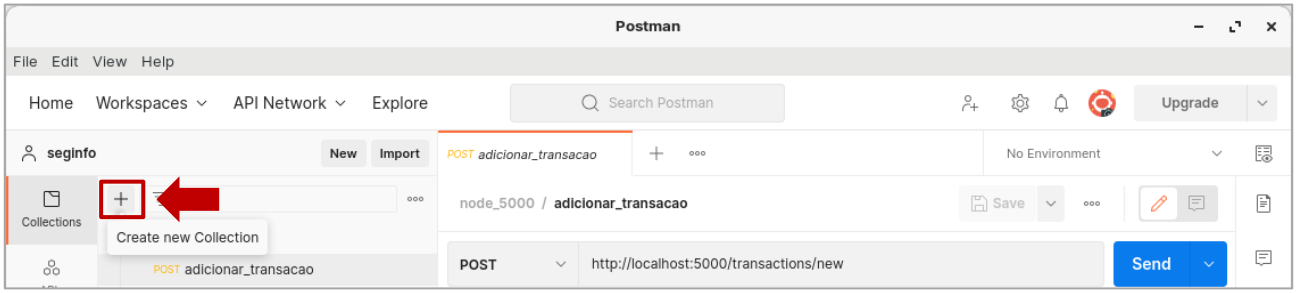
Agora criaremos uma blockchain com 3 nós mineradores a fim de verificar a operação do mecanismo de consenso e como a falta de consenso pode gerar forks temporários. Para isso, primeiramente encerre a execução dos nós usados anteriormente pressionando CTRL+C no terminal que está com o ambiente virtual ativado para inserir o seguinte comando:

```
kill -9 $(pgrep -f blockchain.py)
```

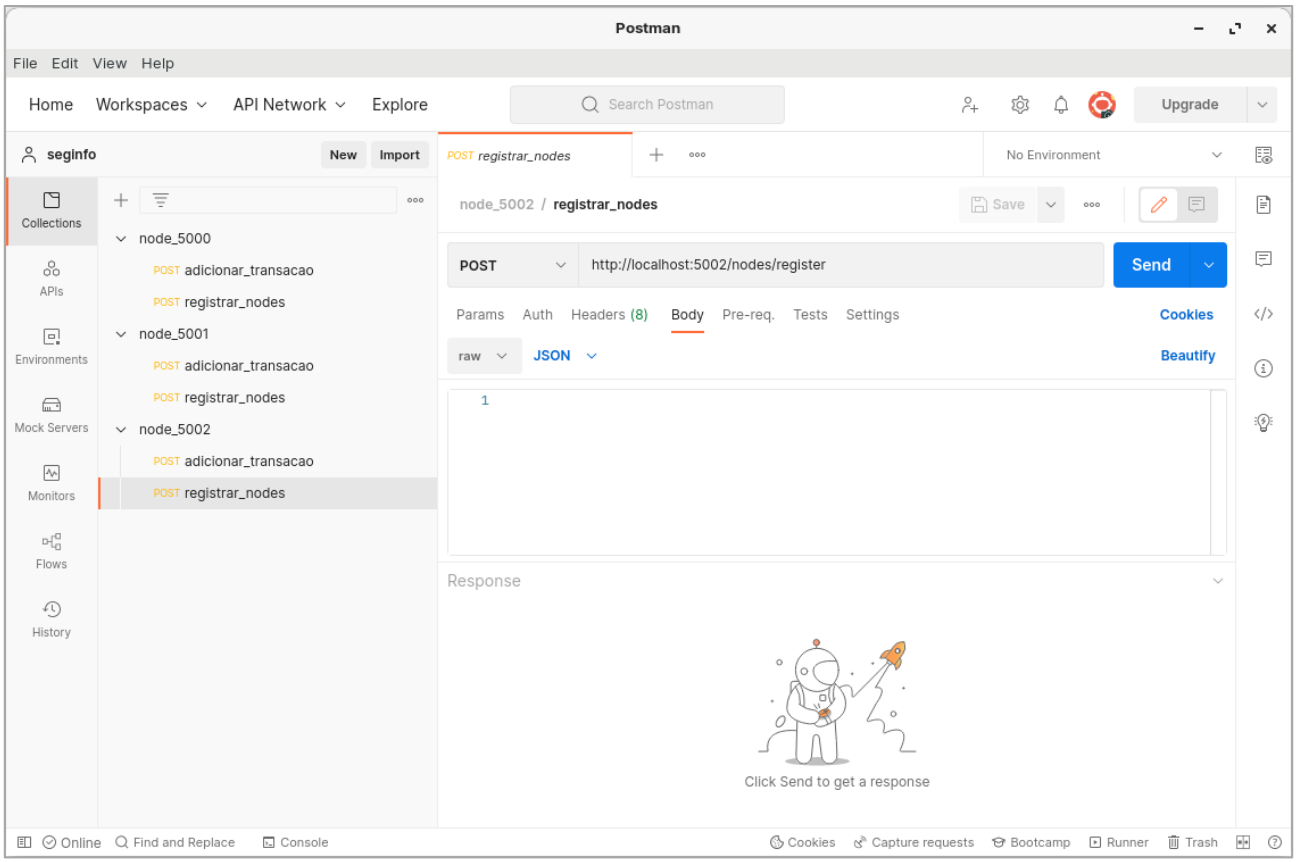
Em seguida, execute o seguinte comando para iniciar os 3 nós nas portas 5000, 5001 e 5002:

```
python blockchain.py --port 5000 & python blockchain.py --port 5001 & python blockchain.py --port 5002 &
```

Como estamos usando mais de um nó, é necessário configurar o método POST para as rotas /transaction/new e /nodes/register para cada nó. O procedimento é idêntico ao apresentado no início da Seção 3. Note que, para adicionar uma nova coleção, basta pressionar o botão indicado a seguir:



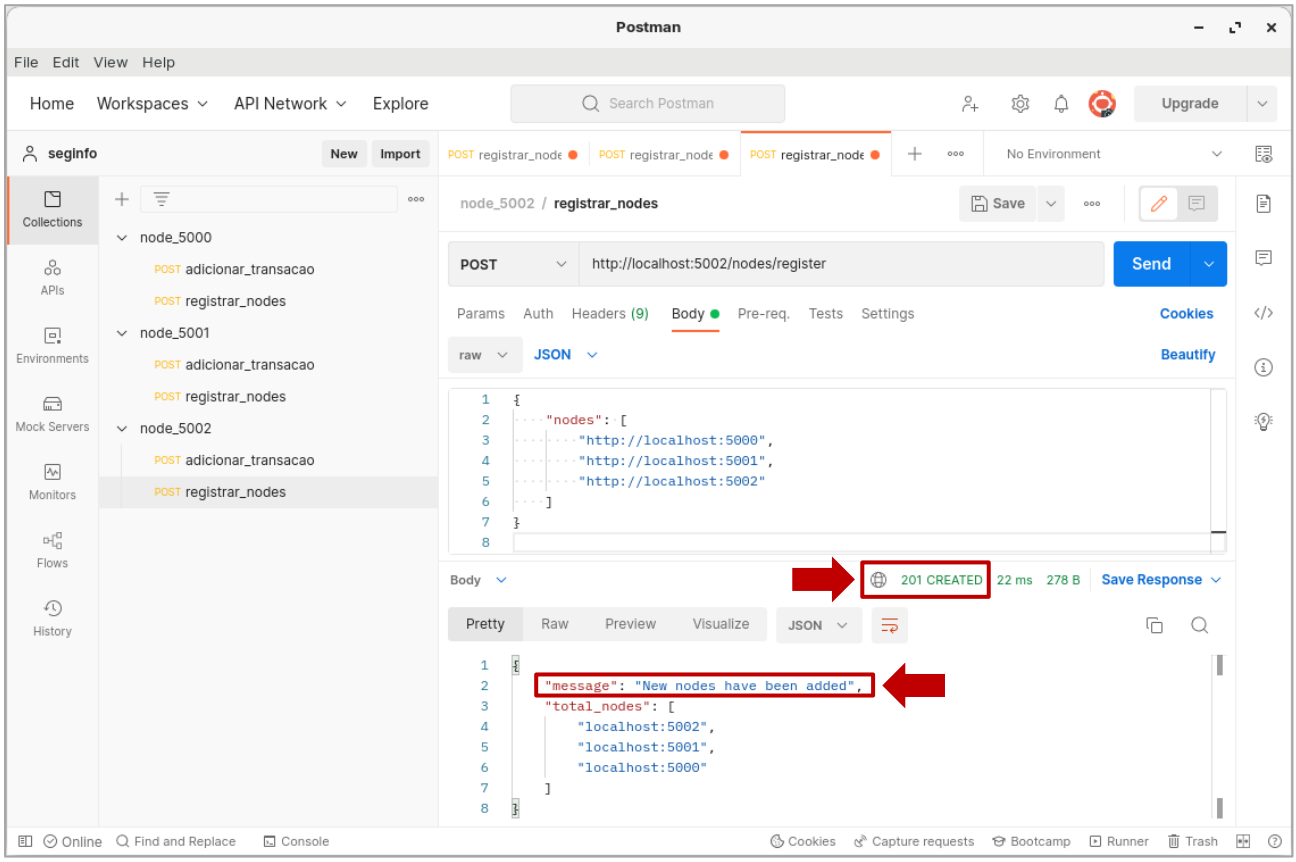
Ao final da configuração dos métodos, seu Postman deverá estar com uma estrutura similar à da captura a seguir:



Usando a estrutura JSON a seguir, registre os nós em cada um deles:

```
{
    "nodes": [
        "http://localhost:5000",
        "http://localhost:5001",
        "http://localhost:5002"
    ]
}
```

O registro bem-sucedido retornará o código de status 201 CREATED e uma mensagem informando que os nós foram adicionados. Para o nó associado à porta 5002:



Adicione as seguintes transações em seus respectivos nós:

node_5000

```
{
    "sender": "Alice_5000",
    "recipient": "Bob_5000",
    "amount": 100
}
```

node_5001

```
{
    "sender": "Carol_5001",
    "recipient": "Diana_5001",
    "amount": 250
}
```

node_5002

```
{
    "sender": "Eduardo_5002",
    "recipient": "Felipe_5002",
    "amount": 123
}
```

Faça a mineração em todos os nós (rota /mine) e observe o conteúdo da blockchain em cada nó (rota /chain). Em seguida, execute o algoritmo do consenso em cada nó acessando as seguintes rotas no navegador:

http://localhost:5000/nodes/resolve
http://localhost:5001/nodes/resolve
http://localhost:5002/nodes/resolve

Observe que a mensagem de resposta em cada um deles informa que não houve substituição da cadeia. Isso significa que o consenso não foi atingido e há 3 forks temporários.

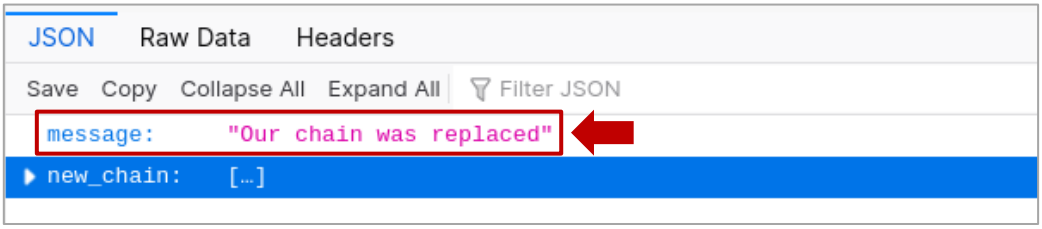
Note que o motivo da ausência de consenso é que a regra utilizada (a cadeia é substituída pela maior cadeia da rede) não pode ser aplicada, pois todas as cadeias possuem o mesmo tamanho.

Agora, adicione a seguinte transação no nó indicado, faça a mineração do bloco nesse nó e execute novamente o algoritmo de consenso em todos os nós.

node_5002

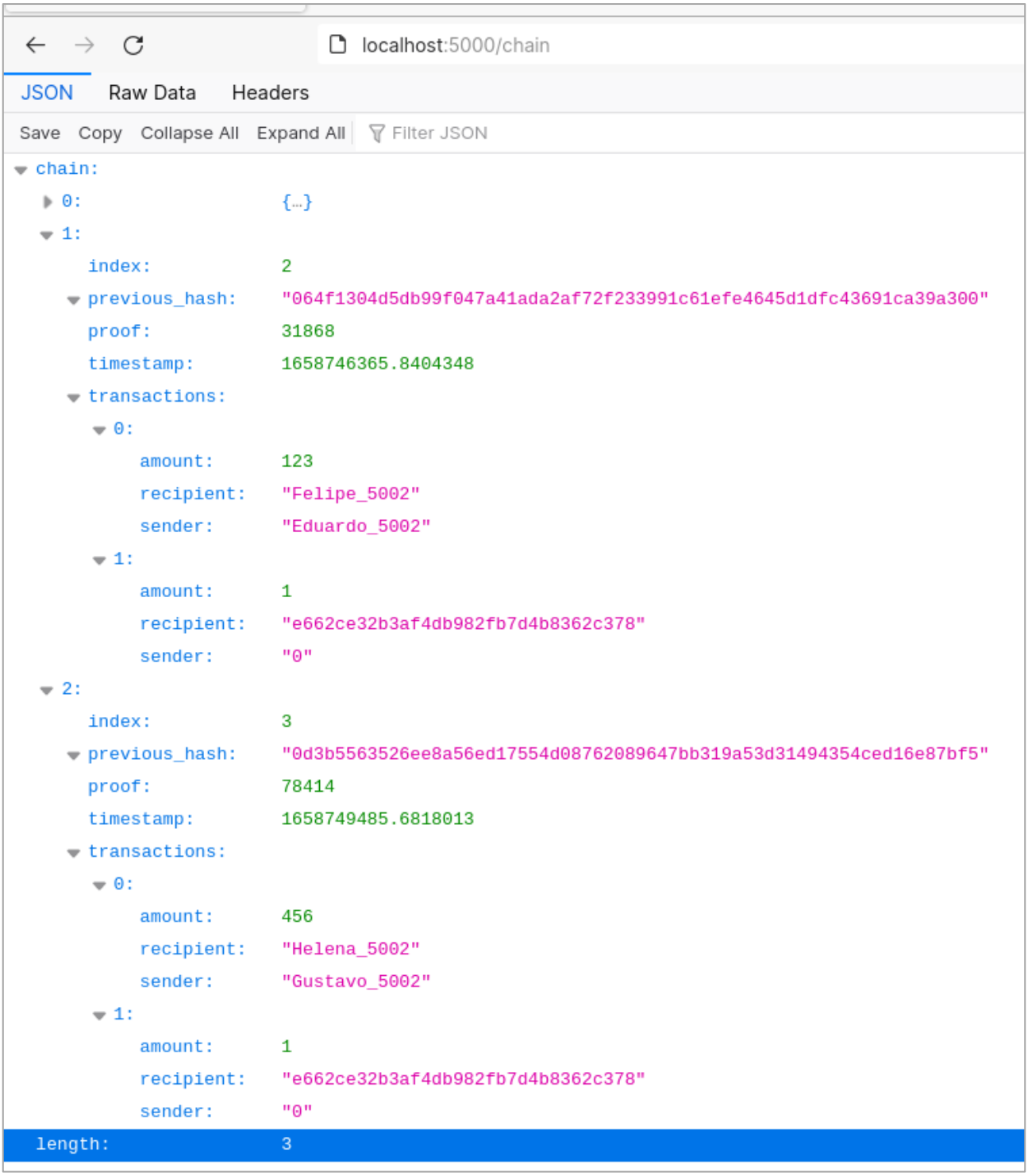
```
{
    "sender": "Gustavo_5002",
    "recipient": "Helena_5002",
    "amount": 456
}
```

Observe que dessa vez apenas o nó 5002 não teve sua cadeia substituída. Para os outros dois nós, a seguinte mensagem de resposta é apresentada:



Isso significa que a cadeia desses nós foi substituída pela maior cadeia da rede, que é a do nó 5002, ocasionando a extinção dos forks temporários. Com isso, pode-se dizer que a rede atingiu o consenso.

Perceba que devido ao consenso atingido, as transações adicionadas inicialmente aos nós 5000 e 5001 não estão mais registradas na blockchain, bem como a recompensa por sua validação:



5. REFERÊNCIAS

[1] GitHub dvf/blockchain

[2] Learn Blockchains by Building One (Daniel van Flymen)

[3] Bitcoin Explorer – Block 466907

[4] Bitcoin – Reason for Mining a Block without Transaction

[5] DigitalOcean – Configuração de Ambiente Virtual Python

[6] Slides da disciplina