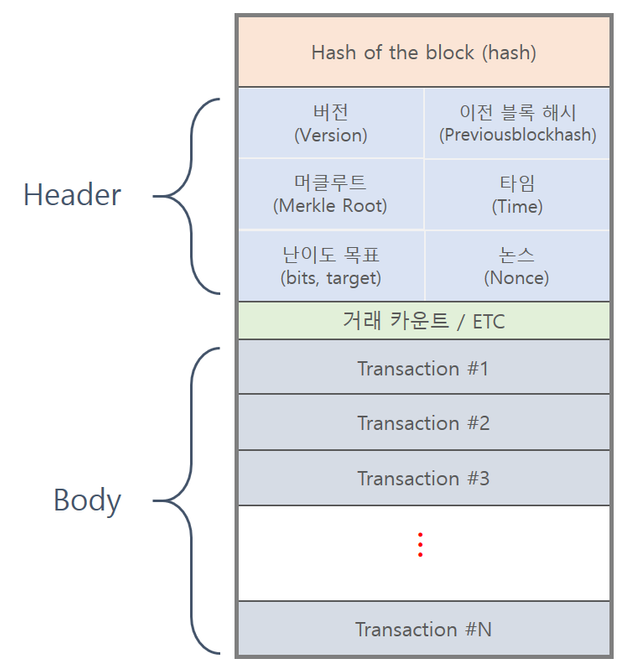
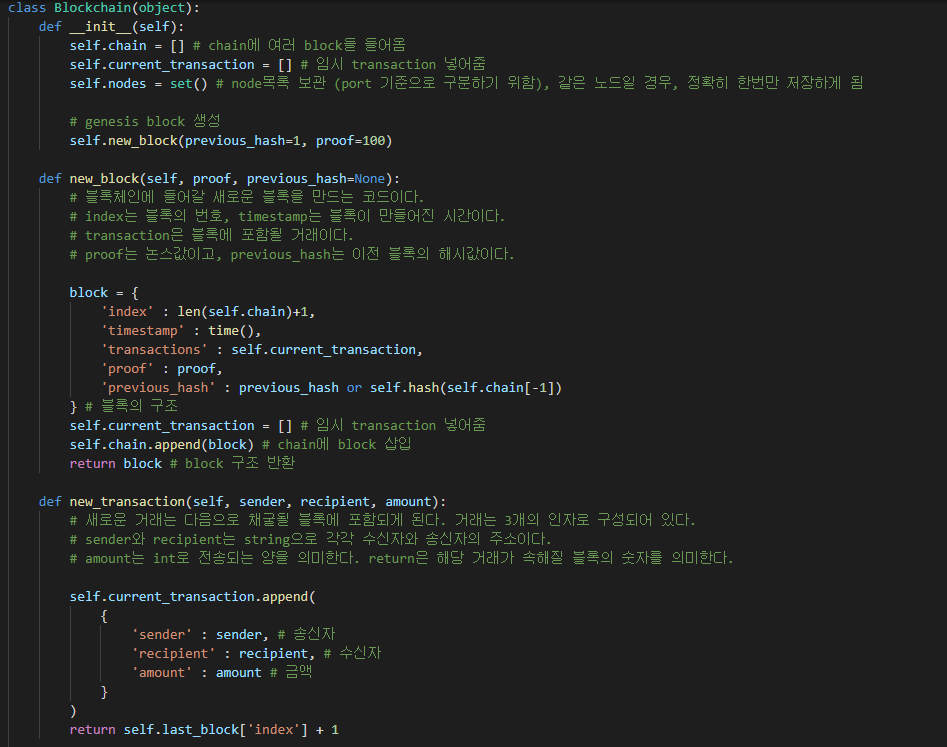
블록체인 : 블록과 블록을 체인형태(Linked List)로 연결한 자료구조이다.

블록 : 블록체인의 원소 개념으로, 다수의 거래 정보의 묶음을 의미한다.

(block구조의 예시)

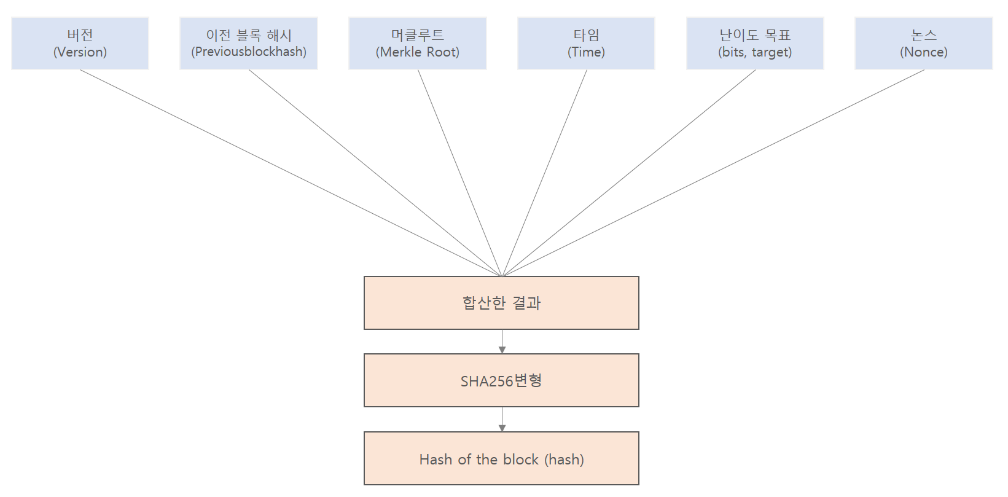


블록 해시 : 블록의 헤더 정보인 버전, 이전 블록 해시, 머클 루트, 타임, bits, 논스 정보를 모두 더하여 합을 구한 후 SHA256으로 변환한 결과 값

이번 코드에서는 블록 해시를 총 5가지로 정해놨다. 상기한 주석을 보면 알 수 있듯이, index는 블록의 번호, timestamp는 블록이 만들어진 시간이다. transaction은 블록에 포함될 거래이다. proof는 논스값이고, previous\_hash는 이전 블록의 해시값이다.

Transaction은 총 3가지로 구성된다. sender와 recipient는각각 수신자와 송신자의 주소이다. amount는 전송되는 양을 의미한다.

Transaction 함수는 상기한 주석과 같이 새로운 거래를 추가하는 함수로도 사용된다. Postman을 이용하여 거래를 추가하면, 다음 채굴될 블록에 자동으로 거래가 추가되게 된다.



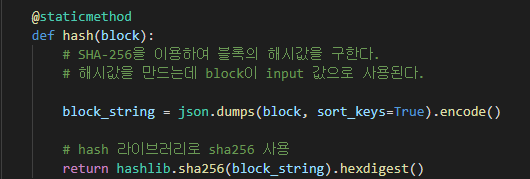
블록 헤더의 합산 정보 -> SHA256으로 변환

Hash 함수 : 임의의 길이를 갖는 메시지를 입력 받아 고정된 길이의 해시값을 출력하는 함수

특징 : 1. 어떤 입력 값에도 항상 고정된 길이의 해시값을 출력한다.

2. 입력 값의 아주 일부만 변경되어도 전혀 다른 결과 값을 출력한다. (avalanche effect)

3. 출력된 결과 값을 토대로 입력값을 유추할 수 없다.



상기한 block 정보를 가져와서 hash하는 과정이다. python에서 기본 제공하는 hashlib를 사용했다.

채굴 : 블록을 생성해내는 과정에 따른 보상.

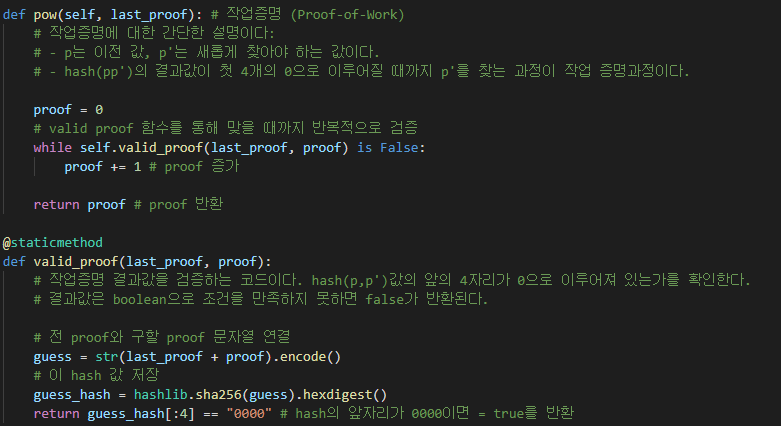


블록의 채굴 과정 :

1. 보상을 받기 위해서는 '미완성 블록'(nonce값이 확정되지 않은 블록)을 '완성된 블록'(nonce값이 확정된 블록)으로 변환시키는 수학문제(조건)를 풀어야 한다.
2. 수학문제의 정답이 되기 위한 조건은 '블록해시'정보가 'target'정보 보다 작아야 한다.
3. 해시함수의 특징으로 인해 'target'보다 작은 '블록해시'를 만들어 내기 위한 'nonce'의 값은 절대 유추할 수 없다.
4. 'nonce'의 값을 유추할 수 없기 때문에 'nonce'의 정보를 0부터 1씩 증가하여 답이 맞는지 확인하는 단순 반복작업이 필요하다. (반복문의 필요)

현실의 코인들은 이 nonce 값을 구하는 과정을 점점 어렵게 하여 코인의 발급량을 줄이고는 한다. 이는 경제학적인 관점의 이론이므로 여기서는 생략하기로 하겠다.

작업증명 (Proof – of – work)



과정이 두가지이다. pow는 nonce값을 올리면서 맞는 nonce값을 찾을 경우 반환하는 함수이고, valid proof는 nonce값을 맞추기 위한 pow 구조의 핵심이다.

Nonce (여기서는 proof)는 0에서부터 시작하여 PoW의 조건을 만족할 때까지 계속 상승한다. 이 과정중에서는 위 변조를 방지하기 위해 전 블록에서의 proof값을 가져오게 된다. (이 값은 블록에 저장되어 있다.)

전 블록에서의 proof값 = p, 현재 블록의 proof값 = p’라 해보자. 이때, p는 고정된 인수이므로, p’의 변화에 따라서 hash값은 계속 달라질 것이다. (상기했듯이 해시 함수의 특징이다.) 이러한 방식으로 나온 hash값에서 일정한 조건을 만족하면 (여기서는 앞자리 4개가 0일때로 가정하였다) 이는 true를 반환하며, 블록이 완성되어서 체인에 추가되게 된다.

이 과정이 성공적으로 이루어지면 채굴자는 보상(예를 들어 1BTC라던가)을 받고, 블록에 체인에 추가되게 된다.

현실적 비트코인에서는 이러한 문제의 난이도가 점점 상승하지만 (상술하였듯 경제학적인 문제 때문이다.) , 이 코드에서는 고정적인 난이도를 사용했다. (만약 난이도를 올리고 싶으면 맨 마지막의 return 함수의 조건을 바꾸면 된다. Hash 앞자리를 0이 5개로 바꾼다던지…)

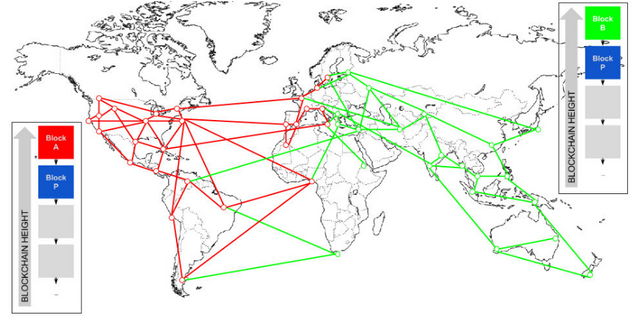
합의 알고리즘 : P2P 네트워크(분산시스템)의 신뢰도를 보장하기 위해서 나온 알고리즘. 탈중앙화된 app는 다음과 같은 특징을 지니게 되는데,

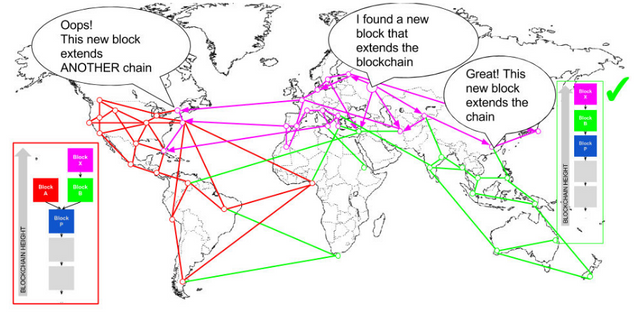
1. 백엔드가 탈중앙화된 피어 투 피어(Peer-to-Peer) 네트워크에서 동작.
2. 소스가 오픈 소스인 일종의 인터넷 애플리케이션.
3. 어떠한 단일 노드도 탈중앙화 애플리케이션(DApp)에 대한 제어권을 가질 수 없음.

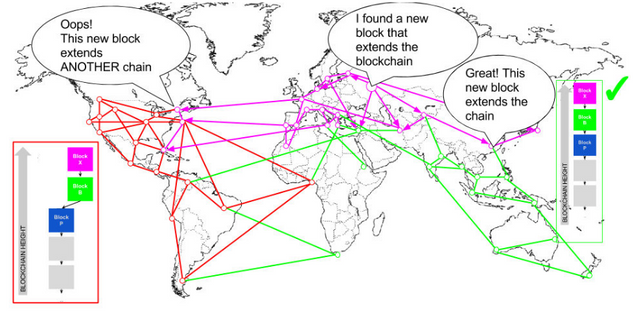
이때, 인터넷에 연결된 어떤 컴퓨터라도 피어가 될 수 있으므로 애플리케이션 데이터를 잘못된 값으로 변경하거나 다른 피어들과 올바르지 않은 정보를 공유하는 피어를 찾아내고 방지하기 위해 피어가 발행한 정보가 올바른지 검증하기 위한 방법이다.

합의 문제 : 정보 도달에 시간차가 있는 네트워크에서 참가자가 하나의 결과에 대한 합의를 얻기 위한 알고리즘. 합의 문제는 블록체인이 나오기 전부터 존재하던 개념이다.

만약 중앙집권적인 기관이었다면(즉, 탈중앙화가 아니라면) 전부 통제가 가능하기 때문에 두 개의 블록이 생성되지 않았을 뿐만 아니라 설사 생성된다 하더라도, 그것을 운영하는 중앙기구에서 어떤 블록이 진짜인지 선택하면 되지만, 탈중앙화된 블록체인에서는 해당 의사결정을 해 줄 사람이 없기 때문에 이를 결정할 과정이 바로 합의 문제를 이용한 합의 알고리즘이다.

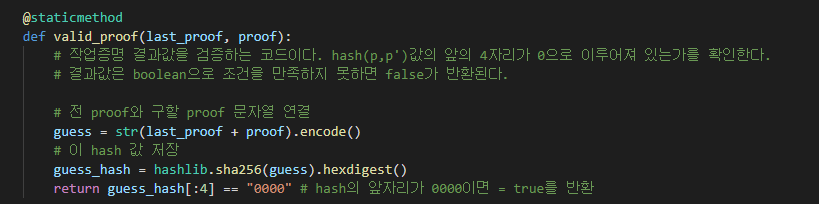


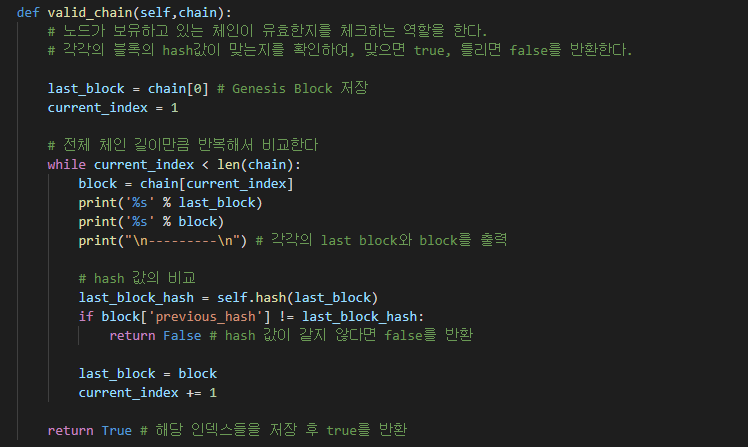


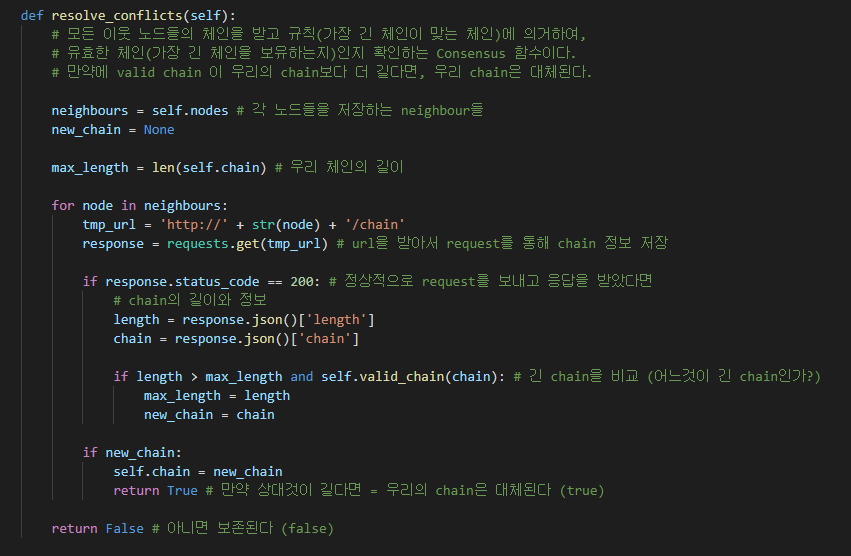


상기한 이미지로 예시를 들어보자. 각각 마지막 block p에서, 미국과 호주에서 각각 block p에 대한 다음 block인 block a와 block b를 생성하였다 하자. 이 과정 중에서 block a와 block b가 공존하는 fork가 발생하게 된다.

이 두개의 체인은 언젠가 네트워크 상에서 충돌을 일으키게 되는데, 이때, 만약 미국에서는 block a에 대해 연결한 블록이 없고, 호주에서 block b에 대한 block x를 연결하게 된다면, 보편적으로 많이 사용되는 합의 문제인 ‘더 긴 체인이 선택된다’는 알고리즘에 따라 호주의 블록인 block b가 선택되며, block a는 고아블록이 된다.



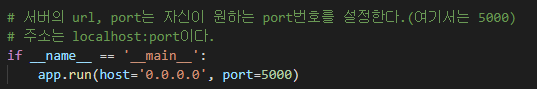




상기한 합의 알고리즘을 세 함수에 나눠서 구현한다.

맨 위의 코드는 새 노드를 nodelist에 저장하는 함수다. 이를 이용하여 우리는 이웃 노드와의 합의 알고리즘을 테스트하는데 사용할 이웃 노드를 추가할 수 있다.

이웃 노드를 하나 만들때는 현재 포트와 다른 포트로 지정해야함을 유의하자. (코드 맨 하단 부분의 port 번호를 바꾼 뒤에 blockchain1.py로 하나 더 저장해두자)



두번째 코드는 이웃 노드가 가지고 있는 체인의 유효함을 검증하는 함수이다. 각각의 hash값이 옳은지 판단 후, 같지 않다면 false를 반환할 것이다. 같다면 저장하고 true를 반환한다.

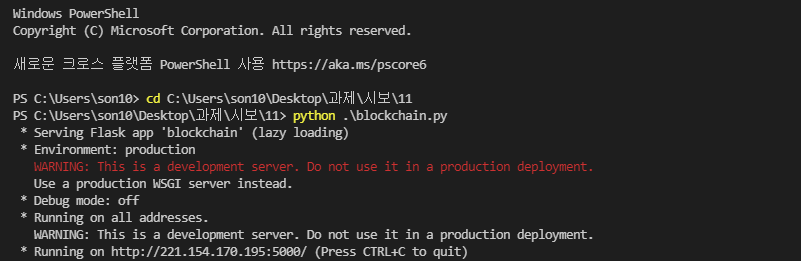
세번째 코드는 두번째 코드의 연장선상으로, 합의 알고리즘의 핵심이다. 우리는 상단에서 설명하였듯이, 두 블록이 충돌하였을 때를 대비하여, 하나의 결과만을 내기 위한 알고리즘을 만들어야 한다. 여기서는 합의 알고리즘으로 ‘가장 긴 체인만이 유효한 체인이다’라는 합의 문제를 내세웠다. 이웃 노드와 정상적으로 연결되었다는 가정하에, chain의 정보와 길이를 받아온다. 이후 chain의 길이를 나의 체인하고 비교하여, 만약 우리 체인이 더 짧다면 이웃의 체인으로 대체되고, true 값을 반환한다 (이는 이후 서버에서 메시지를 보내기 위함이다.) . 아니면, false를 반환한다. 이 과정을 통해 (이웃 노드도 이 과정을 거쳐야 한다) 우리는 하나의 체인 (상기한 문제에 의거한 가장 긴 체인 = 유효한 체인)만을 얻게 되며, 충돌을 방지할 수 있게 된다.

서버 구현에 관해서

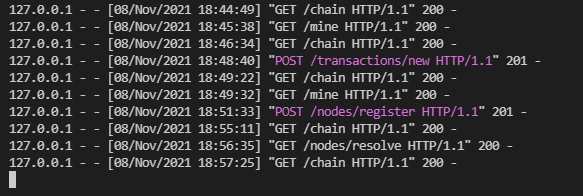
Flask로 간단하게 블록체인 시현용 서버를 구성해본다. 이 서버는 http 요청을 사용하여 웹 기반으로 우리가 짠 블록체인에 명령을 내릴 수 있게 해준다.

이에 더불어 내용확인이나 명령에 좀더 편의성을 확보하기 위해 postman을 사용할 것이다.

실행방식은 간단하다. (그 이전에 flask 라이브러리가 설치되어 있지 않다면 flask 라이브러리를 설치하자. 구동에는 python 또한 필요하다.)



(소스코드가 있는 디렉터리에서) python blockchain.py를 입력한다. Flask app를 통해 서버가 구현될 것이다.



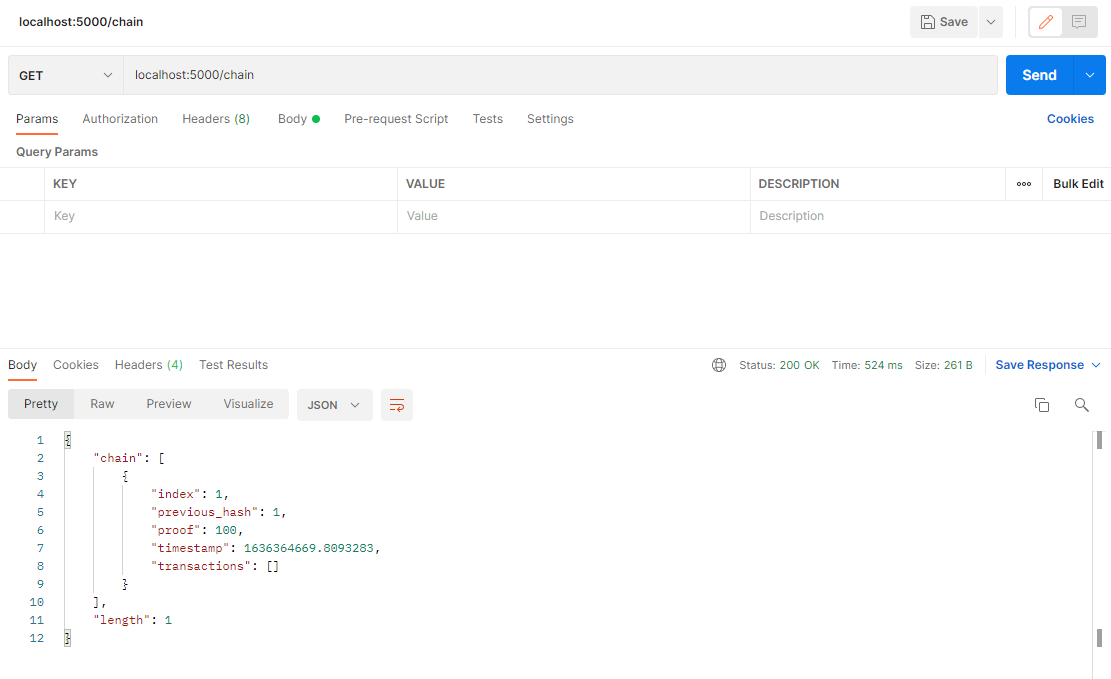
여러 명령어들이 정상적으로 작동하는 모습



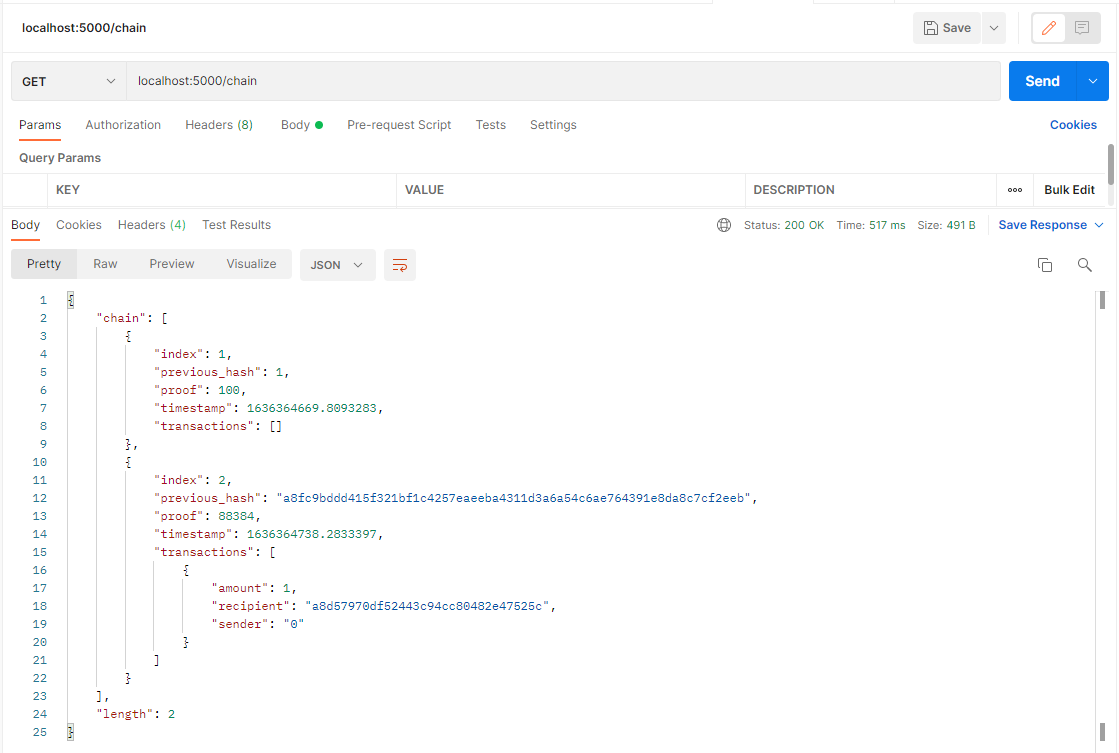
이 코드에 대한 return 반응이다.

이후 postman에서 주소를 입력하여 정상적으로 동작하는지 테스트한다. 하술 할 것은 테스트하는 과정들 이자, 서버 코드에 관한 설명이다.

/chain : 현재 노드의 블록체인 전체를 출력한다.

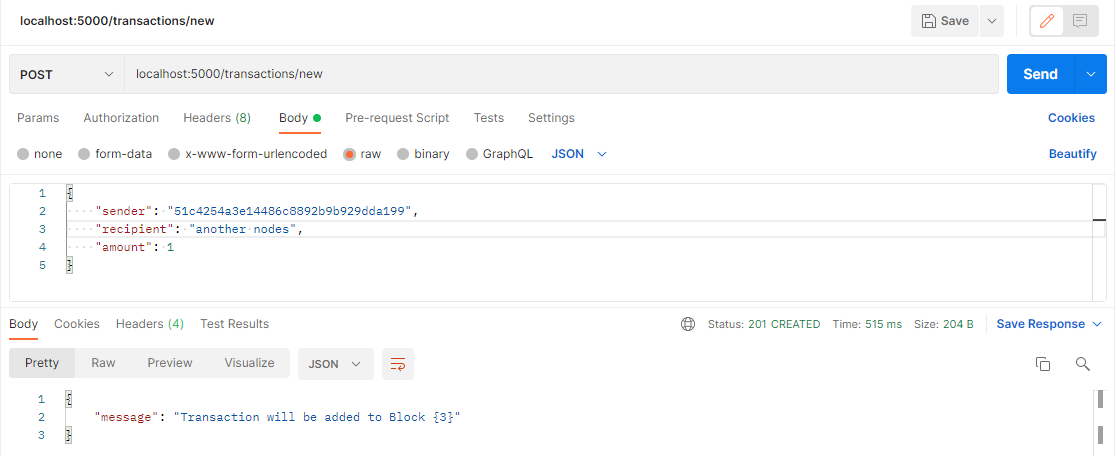


아무것도 mine 하지 않고 입력했을 경우 (제네시스 블록은 언제나 존재하기에, 현재는 제네시스 블록 하나만 표시되어있다.)

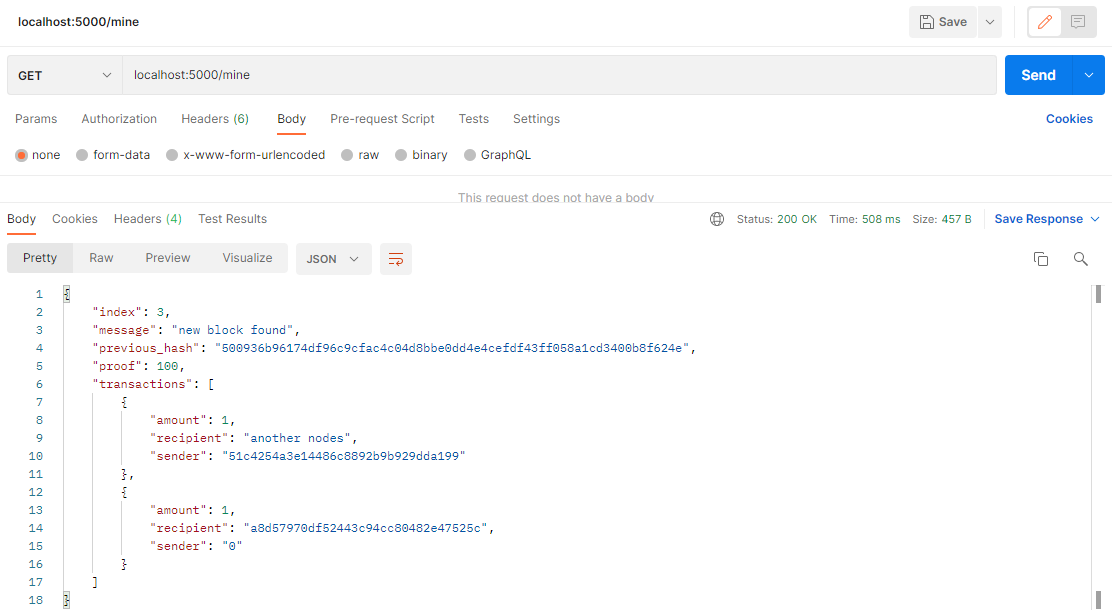


한번 mine 뒤 입력해본 모습 (mine한 block이 chain에 연결되어 있음을 볼 수 있다. 이것이 blockchain인 셈이다.)

/transactions/new : 새로운 거래를 추가하기 위한 주소다. Post 형식으로 보내지며, 보내진 거래는 다음 채굴 시 (지금 체인 길이가 6이면 7번째 블록을 mine할 때) 추가된다.

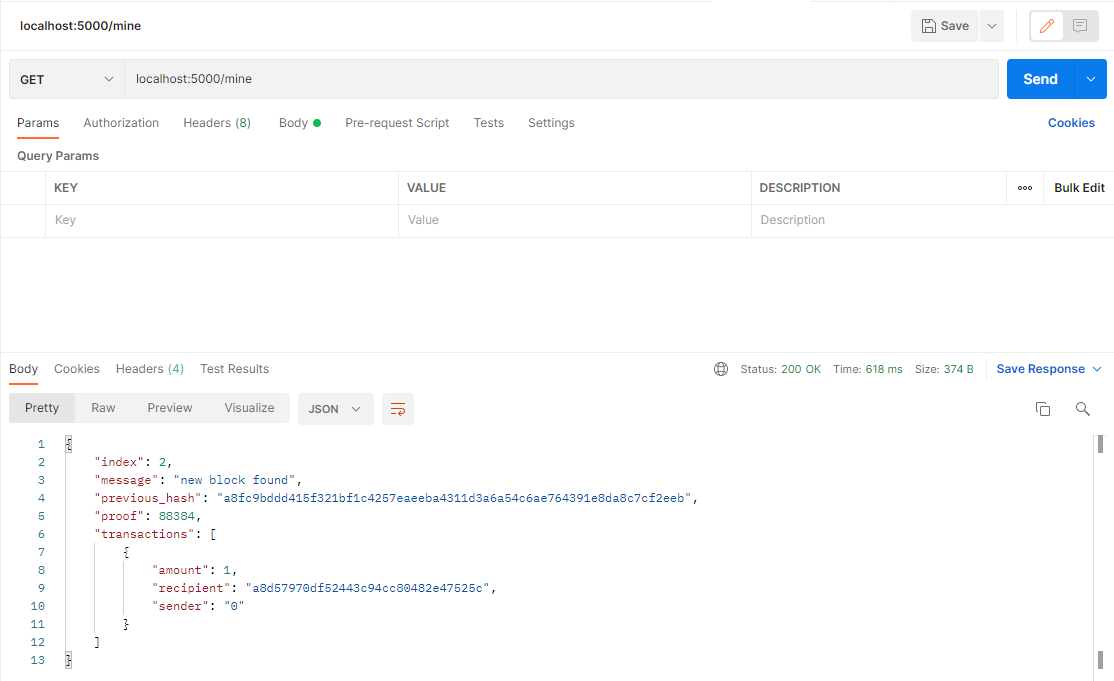


Post 방식으로 json을 이용하여 추가하는 모습



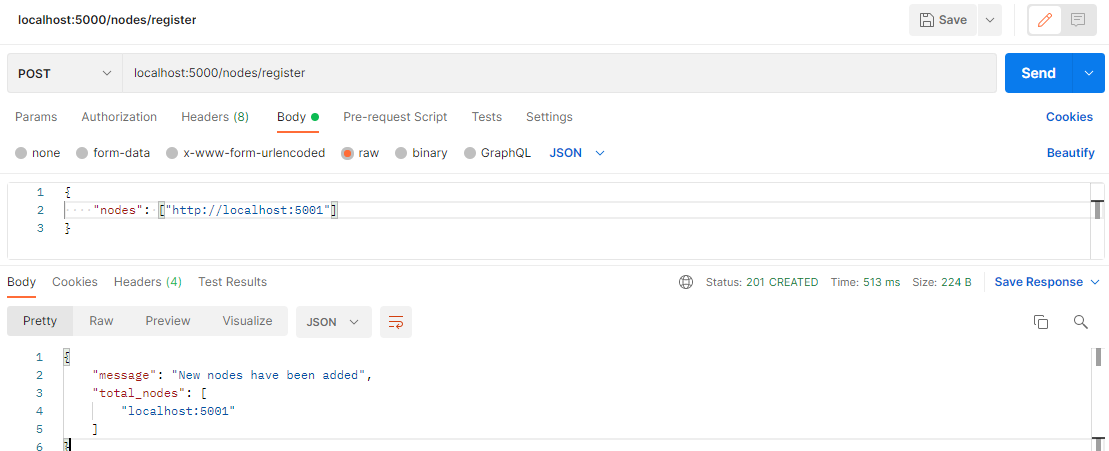
이후 mine시 거래 정보가 추가됨을 볼 수 있다.

/mine : 상기한 채굴을 위한 주소다. 블록 구조가 만들어지며, pow 알고리즘을 통해 nonce값을 생성 후 그 값을 블록에 넣어서 블록을 생성시키고 체인에 연결한다.

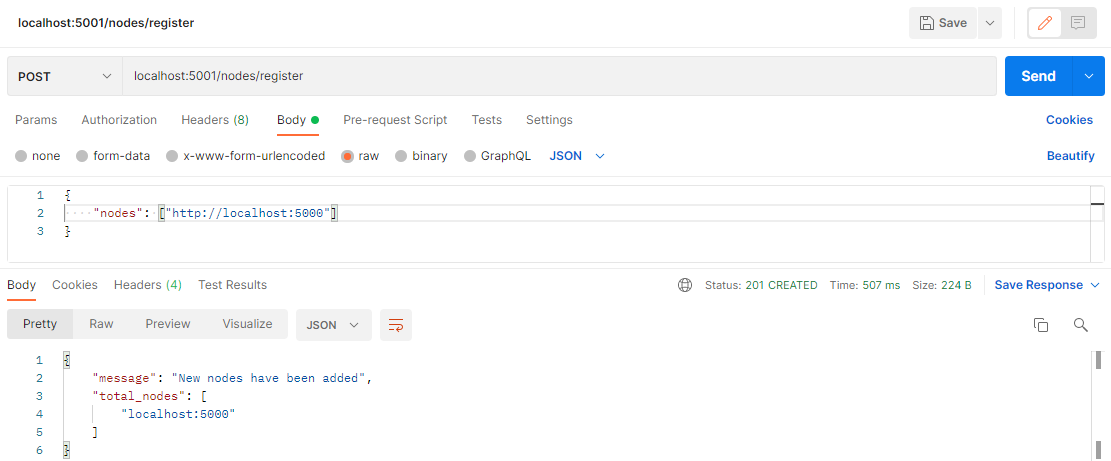


Mine을 하는 모습. 블록 구조에 따라서 정보가 나오며, 트랜젝션에 1만큼의 보상이 나오는 것을 볼 수 있다.

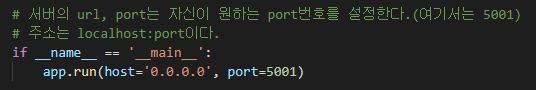
/nodes/register : 새로운 노드를 추가하기 위한 주소다. Post 형식으로 보내지며, 이웃 노드를 추가해서 합의 알고리즘을 테스트 해보기로 한다. (포트 번호는 달라져야 하며, 다른 노드에서도 이웃 노드를 추가해야 한다. 예를 들어 5000번에서는 5001번을 추가한다면, 5001번에서는 5000번을 추가해야 한다.)



Localhost:5000에서 localhost:5001을 추가하는 모습

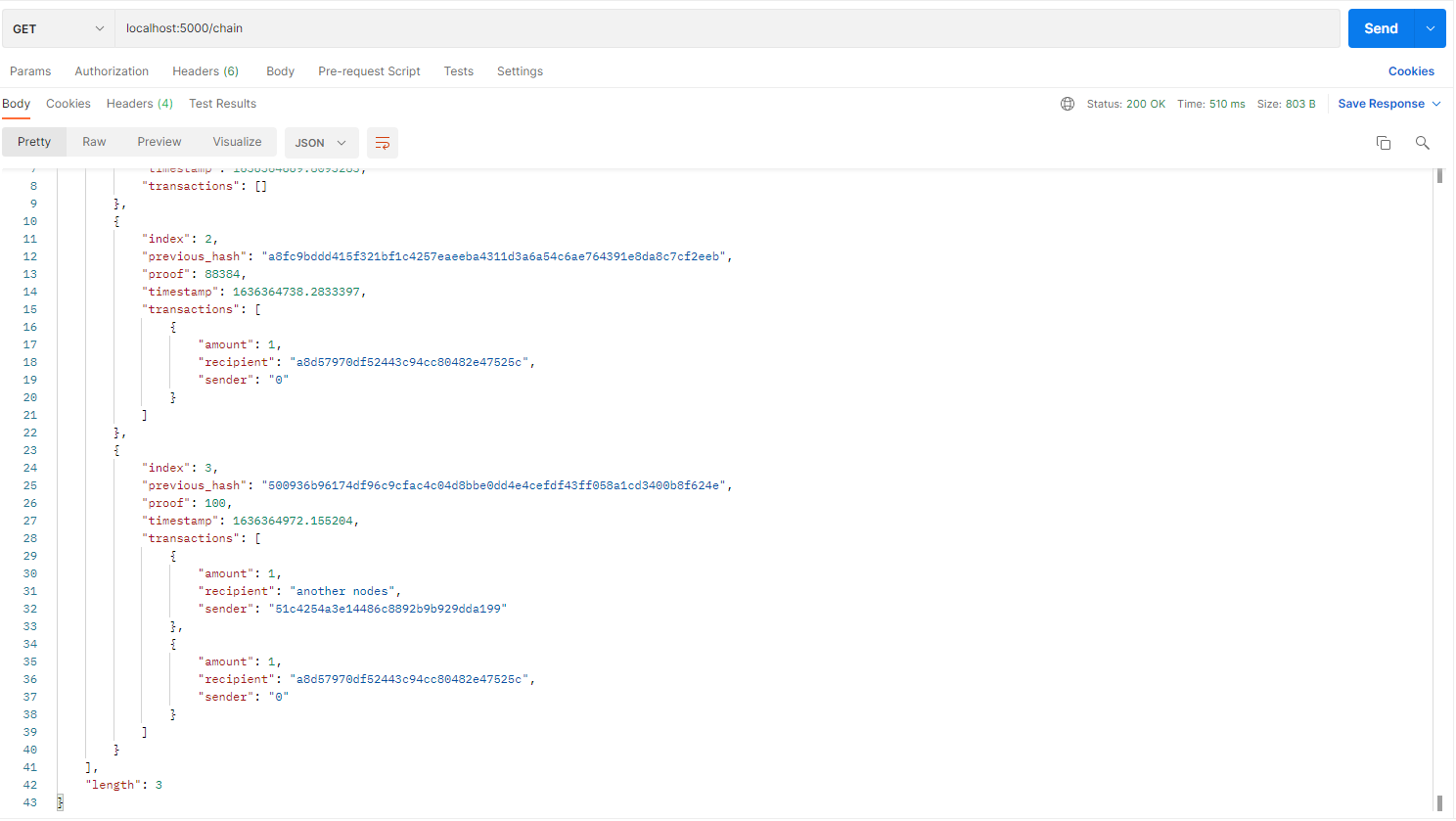


물론 쌍방향으로 연결되어야 하기 때문에 5001번도 동일하게 해준다.

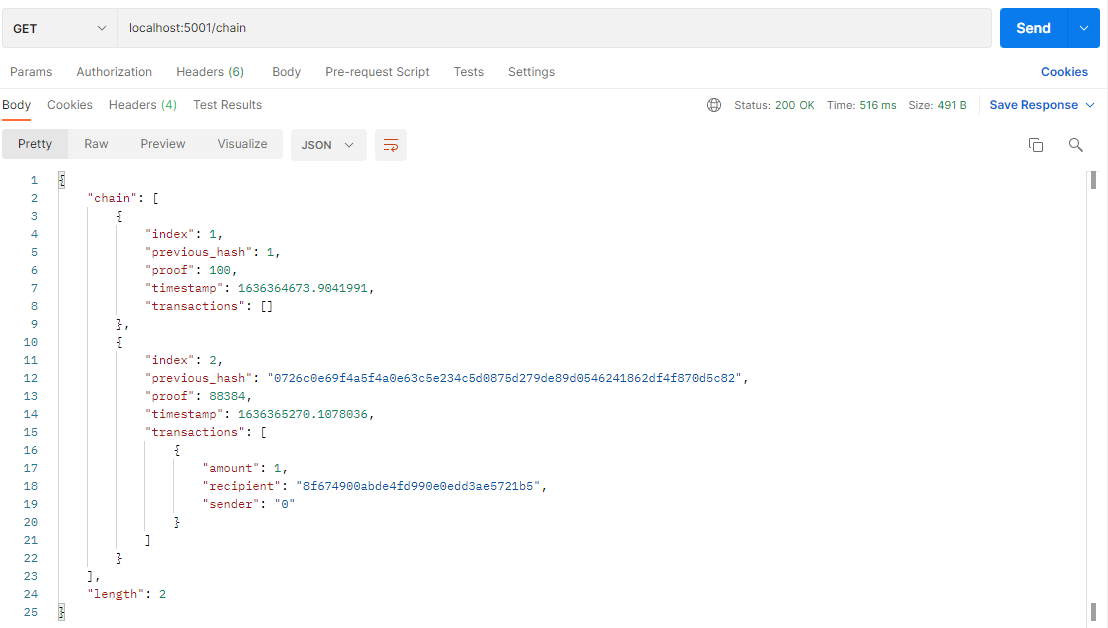


이를 위해서 맨 밑에 서버 port 번호를 바꾼 코드 하나를 더 생성하는 것을 추천한다. (blockchain1.py로 저장해두자)

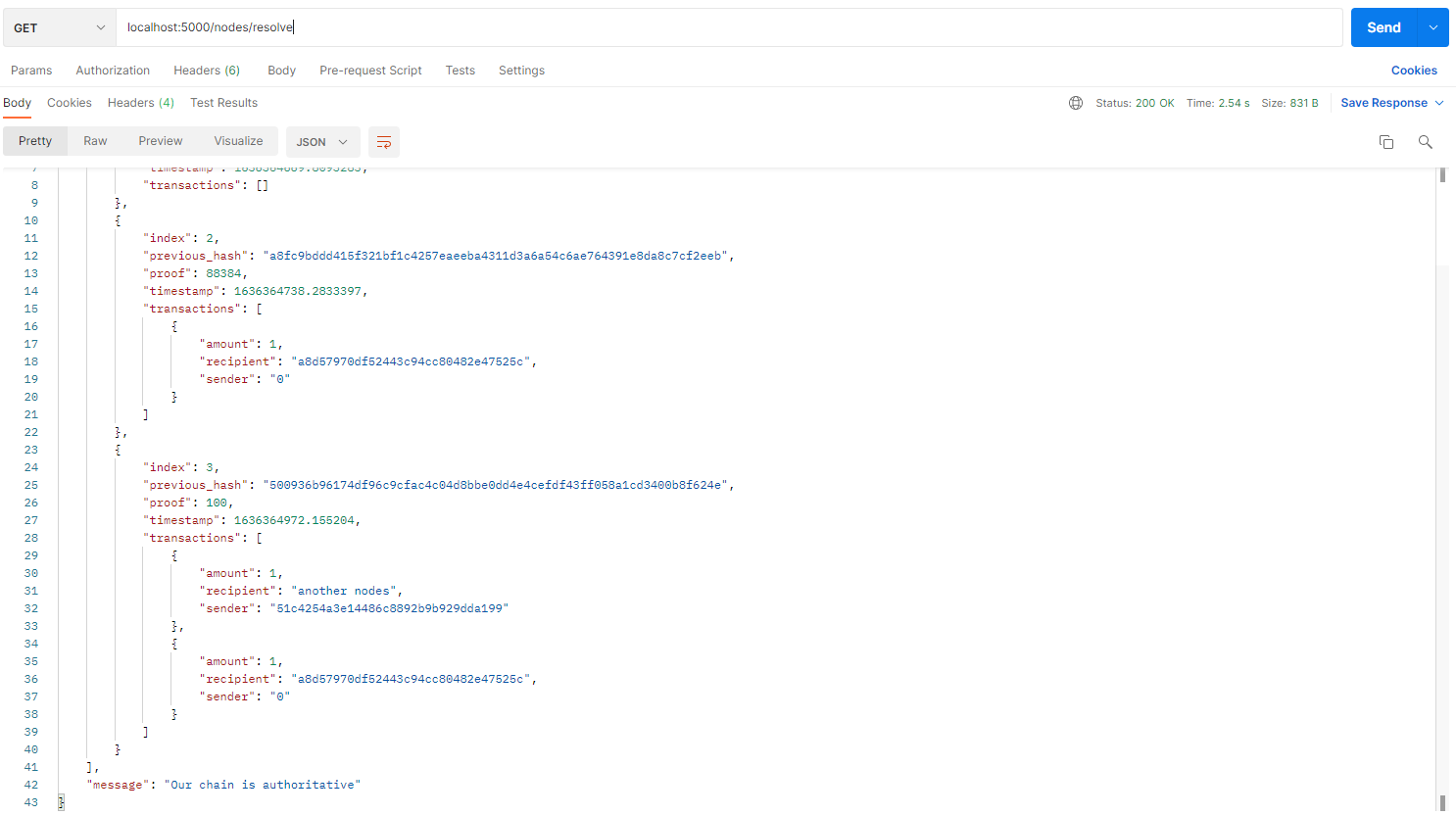
/nodes/resolve : 합의 알고리즘을 위한 주소다. 두 노드가 정상적으로 연결되었다는 가정하에, 합의 알고리즘인 ‘가장 긴 체인이 유효한 체인’을 적용시킨다. (두 노드에서 돌려봐야 한다.) 두 노드의 길이가 다를 때, 가장 긴 노드는 보존되고. 더 짧은 노드는 대체된다. 이렇게 충돌에서 벗어나게 된다.



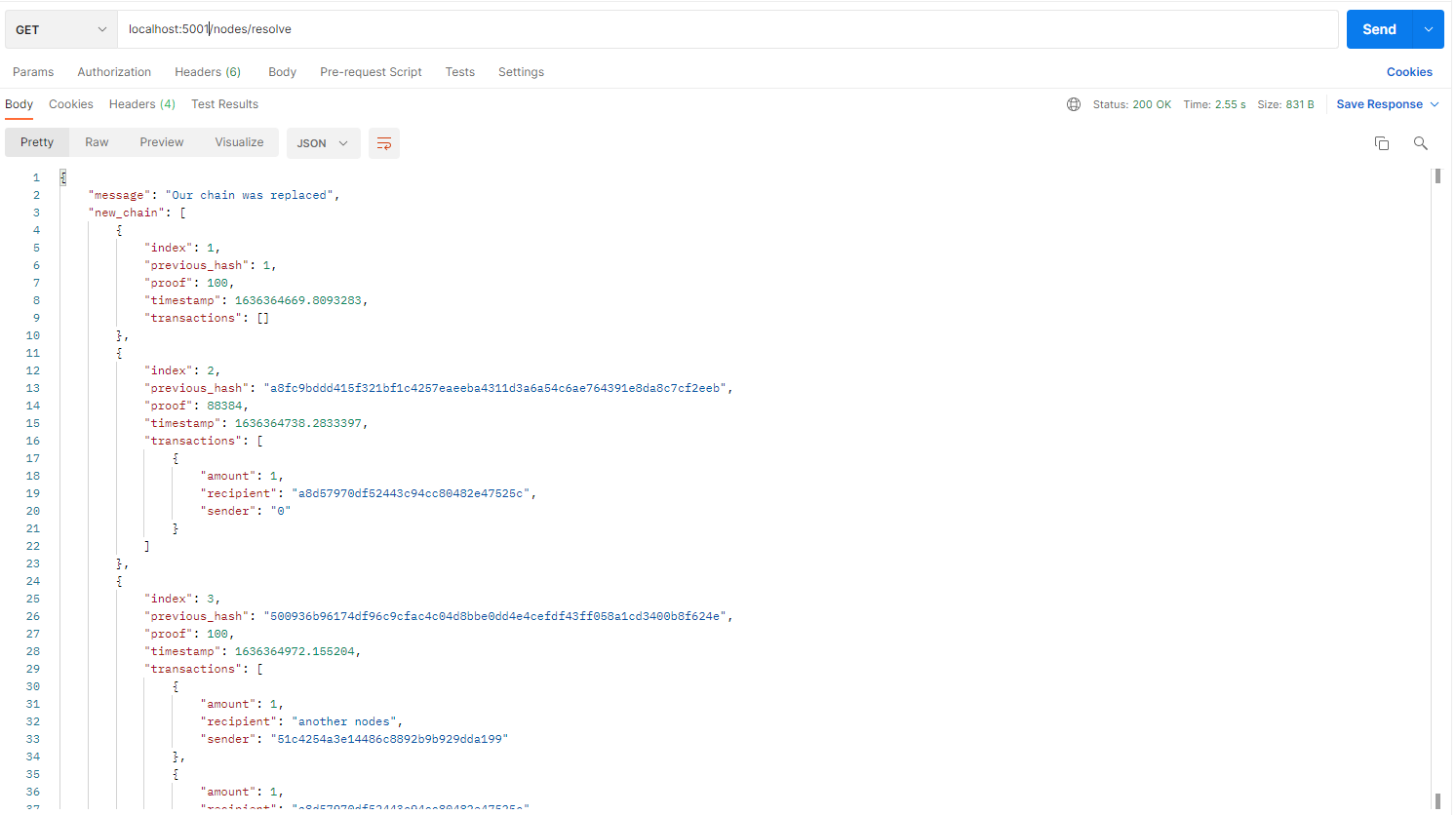
5000번 포트의 체인 (짤렸다) – 길이는 3이다.



5001번 포트의 체인 – 길이는 2다.



5000번에서 실행한 모습. 길이가 이웃 노드보다 더 길기에 보존됨을 볼 수 있다.



5001번에서 실행한 모습. 길이가 이웃 노드보다 짧았기에 5000번 노드의 체인으로 대체되었다. (3번 블록을 잘 보면 아까 추가했던 트랜젝션이 보일것이다.)

참고 링크

<https://steemit.com/kr/@kblock/44-1-pow-pos>

<https://steemit.com/kr/@yahweh87/1-consensus-problem>

<https://steemit.com/kr/@yahweh87/3>

<https://medium.com/caulink/%ED%8C%8C%EC%9D%B4%EC%8D%AC%EC%9C%BC%EB%A1%9C-%EB%B8%94%EB%A1%9D%EC%B2%B4%EC%9D%B8-%EB%A7%8C%EB%93%A4%EA%B8%B0-part-1-4386dbc735e>

<https://medium.com/caulink/%ED%8C%8C%EC%9D%B4%EC%8D%AC%EC%9C%BC%EB%A1%9C-%EB%B8%94%EB%A1%9D%EC%B2%B4%EC%9D%B8-%EB%A7%8C%EB%93%A4%EA%B8%B0-part-2-633bb0555221>

<https://www.banksalad.com/contents/%EC%89%BD%EA%B2%8C-%EC%84%A4%EB%AA%85%ED%95%98%EB%8A%94-%EB%B8%94%EB%A1%9D%EC%B2%B4%EC%9D%B8-%EB%B8%94%EB%A1%9D%EC%B2%B4%EC%9D%B8%EC%9D%98-%EC%9B%90%EB%A6%AC-%EC%B1%84%EA%B5%B4-%ED%95%B4%EC%8B%9C-%EA%B7%B8%EB%A6%AC%EA%B3%A0-%EC%9E%91%EC%97%85%EC%A6%9D%EB%AA%85-qvCud>

<https://meetup.toast.com/posts/107>

<https://flask.palletsprojects.com/en/2.0.x/>