An-Najah National University
Computer Engineering Department
Distributed Operating Systems - 10636456

Microservices Lab2-Bazar.com

Dr.Samer Arandi
Aya Farhan Suwadeh-11717238

- **System setting up and overview:**

This part of the lab focuses on replication, consistency and caching which are typically used in distributed applications, as a proof of concept I set up a new machine as a replication of both backend and backend servers, already have two existing machines from part one which re front and back servers separately, front runs alone on a standalone machine, backend origin servers run on another machine and the replica servers run on a third machine, screenshots below demonstrate how apps run on machines:

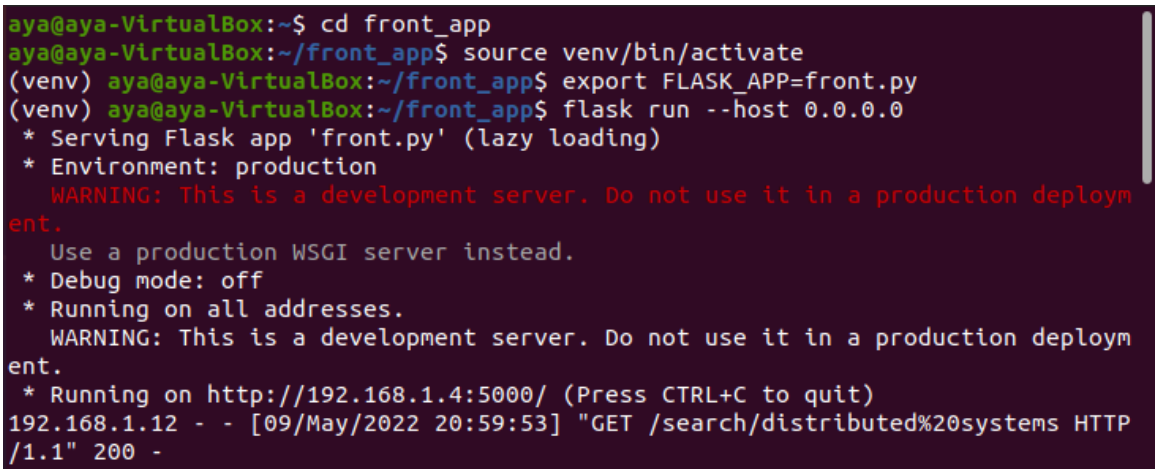


figure 1: front app setting up

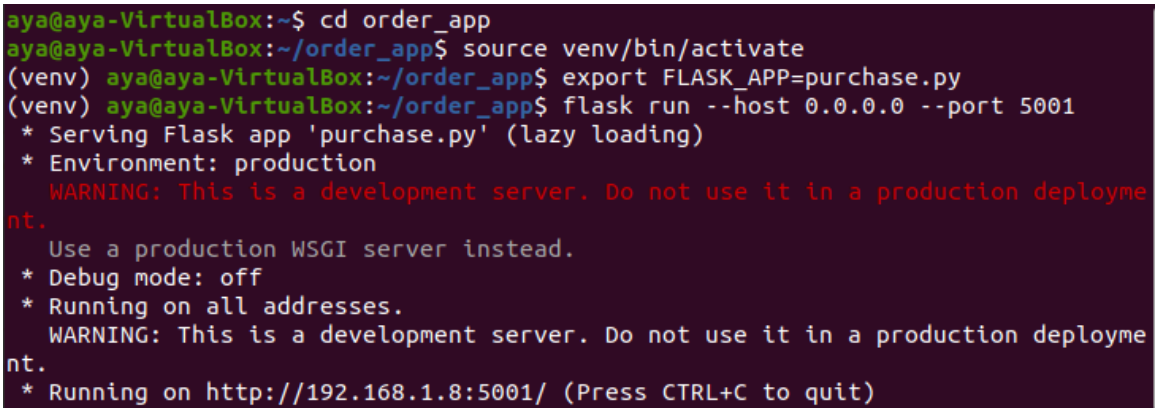


figure 2: origin order app setting up

```
aya@aya-VirtualBox:~/order_app$ cd ..
aya@aya-VirtualBox:~$ cd catalog_app
aya@aya-VirtualBox:~/catalog_app$ source venv/bin/activate
(venv) aya@aya-VirtualBox:~/catalog_app$ export FLASK_APP=purchase.py
(venv) aya@aya-VirtualBox:~/catalog_app$ export FLASK_APP=query_update.py
(venv) aya@aya-VirtualBox:~/catalog_app$ flask run --host 0.0.0.0 --port 5001
 * Serving Flask app 'query_update.py' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployme
nt.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployme
nt.
 * Running on http://192.168.1.8:5001/ (Press CTRL+C to quit)
```

figure 3: origin catalog app setting up

```
ayaf@ayaf-VirtualBox:~$ cd first_replica_order
ayaf@ayaf-VirtualBox:~/first_replica_order$ source venv/bin/activate
(venv) ayaf@ayaf-VirtualBox:~/first_replica_order$ export FLASK_APP=purchase.py
(venv) ayaf@ayaf-VirtualBox:~/first_replica_order$ flask run --host 0.0.0.0
 * Serving Flask app 'purchase.py' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deploym
ent.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses (0.0.0.0)
   WARNING: This is a development server. Do not use it in a production deploym
ent.
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.1.10:5000 (Press CTRL+C to quit)
```

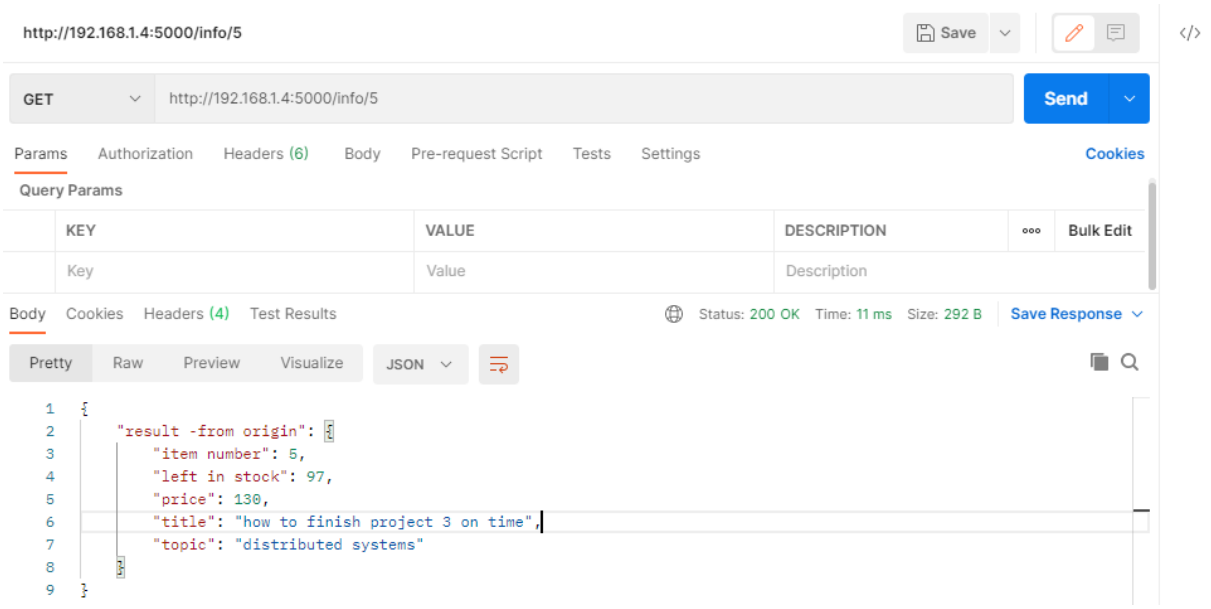
figure 4: replica order app setting up

```
ayaf@ayaf-VirtualBox:~/first_replica_order$ cd ..
ayaf@ayaf-VirtualBox:~$ cd first_replica_catalog
ayaf@ayaf-VirtualBox:~/first_replica_catalog$ source venv/bin/activate
(venv) ayaf@ayaf-VirtualBox:~/first_replica_catalog$ export FLASK_APP=query_upd
ate
(venv) ayaf@ayaf-VirtualBox:~/first_replica_catalog$ flask run --host 0.0.0.0 -
-port 5001
 * Serving Flask app 'query_update' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deploym
ent.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses (0.0.0.0)
   WARNING: This is a development server. Do not use it in a production deploym
ent.
 * Running on http://127.0.0.1:5001
 * Running on http://192.168.1.10:5001 (Press CTRL+C to quit)
```

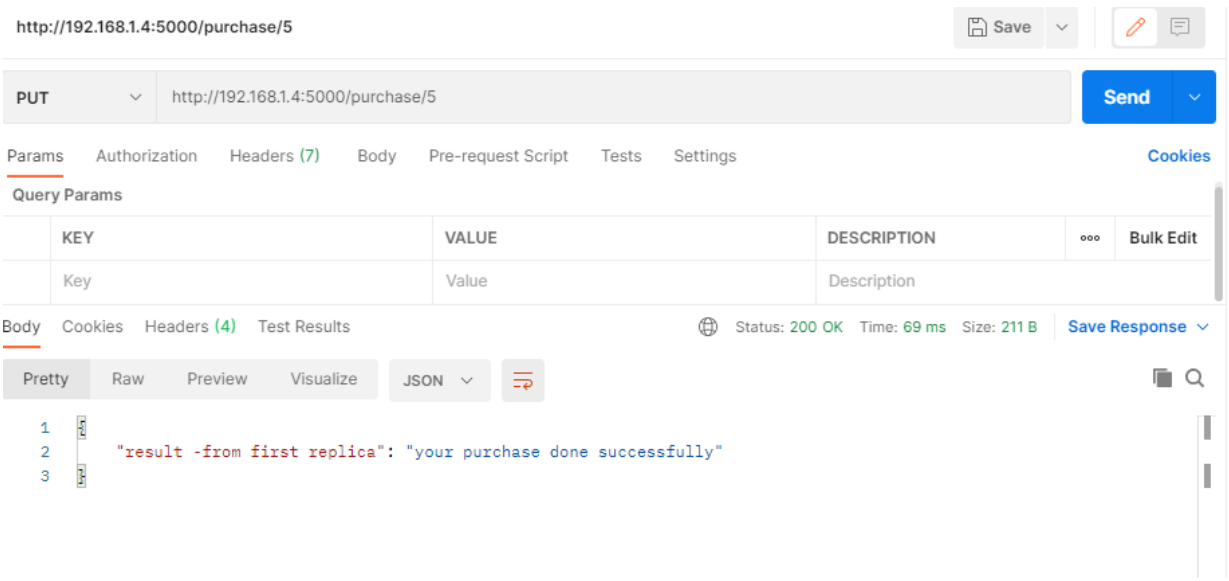
figure 5: replica catalog app setting up

- **How the system works? (outputs):**

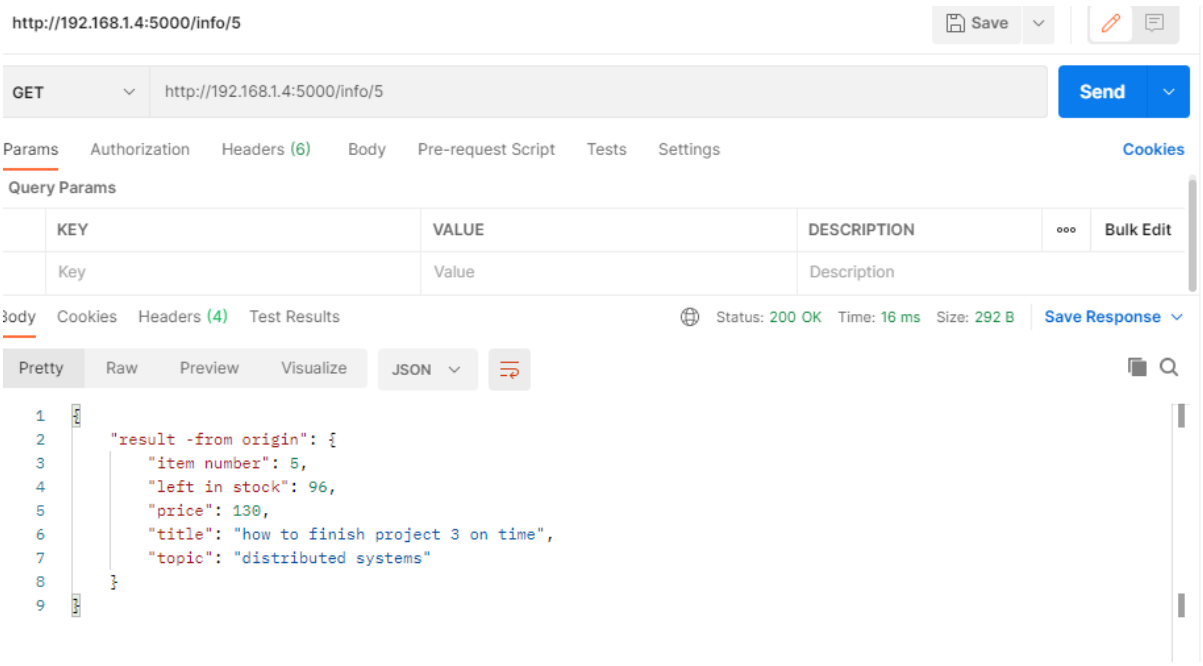
  When we hit a query against the whole system the front-server forward the coming request either to the origin-server or to the replica-server based on round-robin load balancing algorithm, and if it's a read operation, the query and its result will be cached if it's not already cached for the next time, if it's a write operation for example purchase query-the one I implemented, it will be forwarded to one of the backend servers, and then invalidate the item if it's in cache to keep data for user up-to-date, the following screenshots shows some of the workflow:



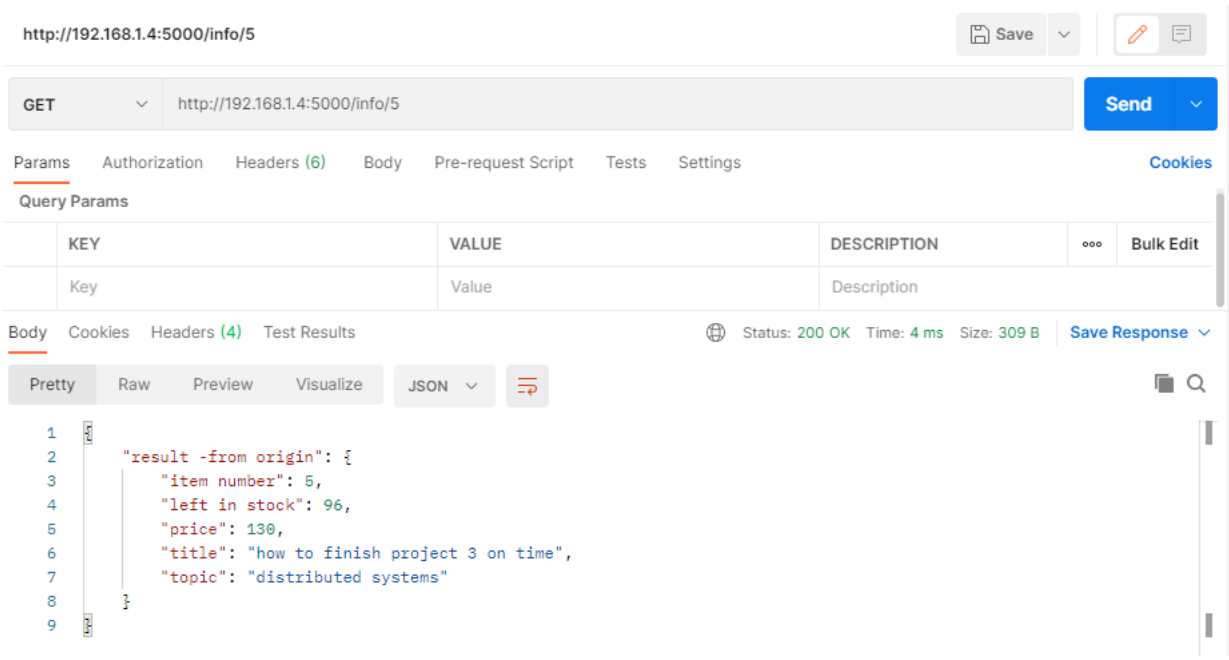
In the first screenshot, the first info request forwarded to the origin server, and took 11ms



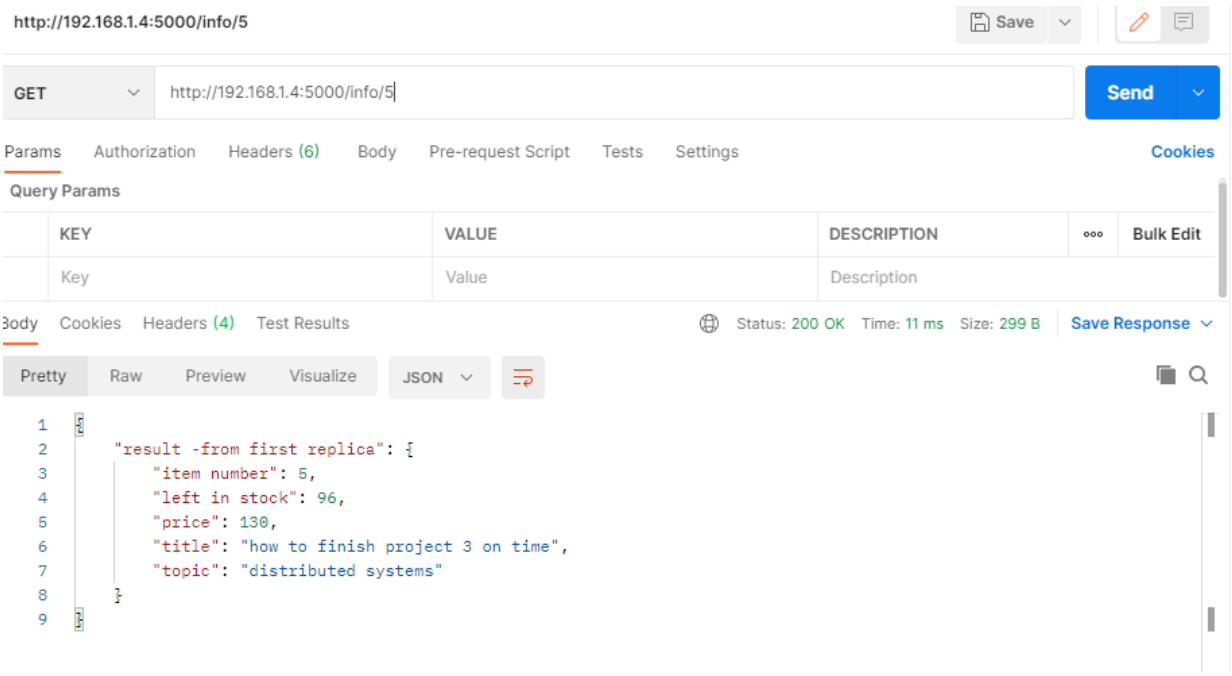
In the second screenshot, I made a rite operation (purchase), and it was forwarded to the replica server-that's what round robin does, and since it's a rite operation , directly an invalidate request hit the front server to stale the cached content corresponding to the updated item.



In the above screenshot, I queried again the item 5, and as we can notice that this is not a cached content, since it took 16ms, and we can see that the data (number of items) is updated, even that the purchase happens from a server but the effect reflected on both servers



And this screenshots, shows how the time went down to the ¼ when it was cached (16ms without caching, 4ms with caching)



For the cache I used LRU policy replacement because the cache memory in the real world has a relatively small storage comparing to the disks servers, so as we can see that the request is not cached this time because I made 3 queries, so they replaced the very first cached query, so this time it took 11ms which means it wasn't cached.

- **System tradeoffs and possible improvements and extensions:**

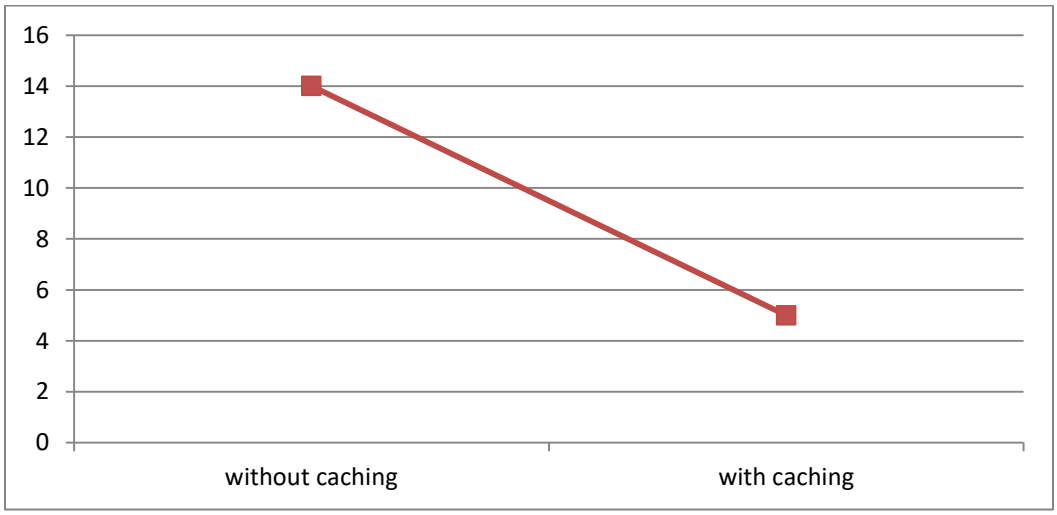
This system is a proof of concept of distributed applications, and how we can satisfy its requirements like consistency, reliability and etc., it works fine and smoothly for the current situation and queries, but for more write and complex queries the mission will be harder, and guaranteeing high level of consistency and reliability would be 10 times harder, so the need for frameworks or tools or services would be a must to make the developers effort more focused on the business logic code rather than on the coordination and orchestrating between the distributed fragments of the app , and for the current design we might applied the concept of replication but it requires extra work to keep all data consistent and the same, and for caching it needed extra work also to keep data updated, this adds on some latency on the query time, and another possible improvement rather than the services, for the current design it would be more real if each micro service (origin and replicated) were on a separated machine, this would make the system closer to the reality.

- **Experimental Evaluation and Measurements**

  Average measurements:

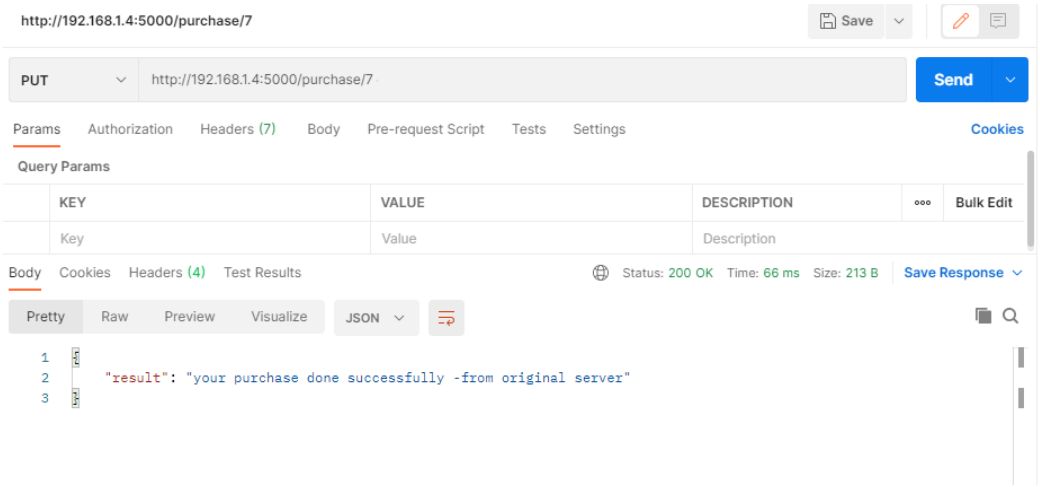
  - Info request without caching = 1st request took 11ms +2nd request took 16ms+3rd request took 15ms = 14ms in average.
  - Info request with caching = 1st request took 6ms +2nd request took 4ms+3rd request took 5ms = 5ms in average.

    ➔ Then we can find out that with caching approximately the query time decreased to the 1/3 which highly affects the performance positively.

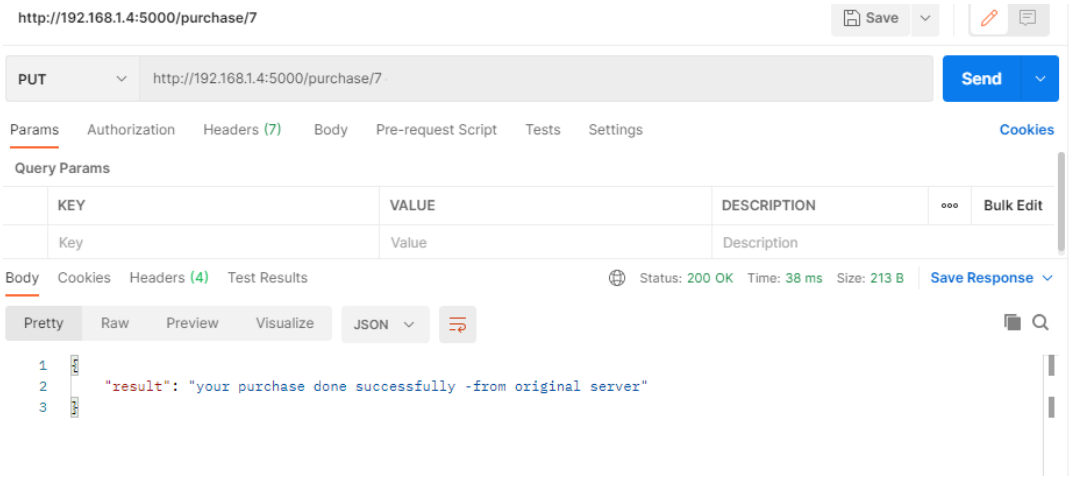


With consistency vs. without:

As I said and suggested that with consistency latency would be longer, and that's right for most of the time and it would be more obvious in bigger apis:



With consistency it took 66ms as shown above



and here without consistency it took 38ms,

With consistency	Without consistency
66ms	38ms

That's because, with consistency we need to contact to the involved servers (via http requests) which requires some time.

Cache miss case:

When we have cache miss an extra latency would be added on the overall time, because it will require going through the cache before heading off to the server directly.

Github repository:

https://github.com/aya-farhan/dos-microservices-part2