Duaa shehada - 11716595

DOS - Bazar project - part 2 :

As mentioned in the first part of this project, it is a NodeJs, ExpressJs which is a lightweight web framework used to create a RestFul project. And because this part is based on the first part, it has the same servers but here we have a replica from each back-end server.

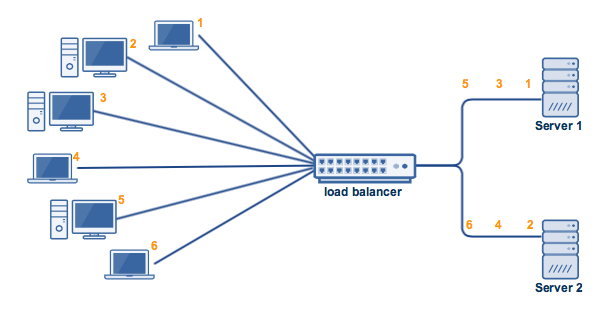
The project Dockerized by using docker desktop and Dockerfile to configure each server and docker-compose that responsible for building the images and creating the containers.

Docker-compose up command used to run all the containers. this is os-level virtualization. the Docker containers communicate with each other on top of a network. Here, in this project, all the servers have different IP addresses on the same network. Also, of course, each server has a different port.

As seen in the Github repository, all the servers are in one repository, and this is called monorepo.

The operations in the servers are the same, where these servers are micro-web servers using a RESTful API and an HTTP protocol with the four verbs (GET, POST, PUT, DELETE ) .so, here the focus will be on the newly added features to the project.

At first, there is a load balancer added to the front-end server. It is used the Round Robin technique, where it is used the concept of sending the request to the first server then the second request to the second server then the third request to the third server and so on . until there are no more servers it is repeater the same process to the first server then second and so on . in our case, there is 2 replica. so, the request is sent for the first time to the first server then the second request to the second server, and so on.



This technique reduces the overhead. because instead to send all the requests to the same server it is to balance the sending between the replica which will increase the performance.

The second feature added to the front-end is the in-memory cache, where it is the case of the result of each read query .so, if the request hits again it will read the result from the cache with no need to send another request. This approach is very useful because it decreases the response time and the latency for the requests and the performance increase.

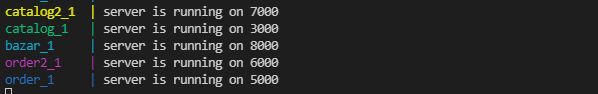
The replication is associated with consistency and for the consistency, I do a lot of work :

* Use invalidate technique when hitting a write request so the database is updated and the result in the cache is invalid. so, in the write request, I send a message to tell the cache that this item is updated and needs to be removed from the cache. as a result, any read request will need to send the request again to the server and not read from the cache.
* For the database consistency, a PUT request was added to update the database. so if the first server updates his database, it sends this PUT request to the second server to tell him to update his database too.

**How my project work :**

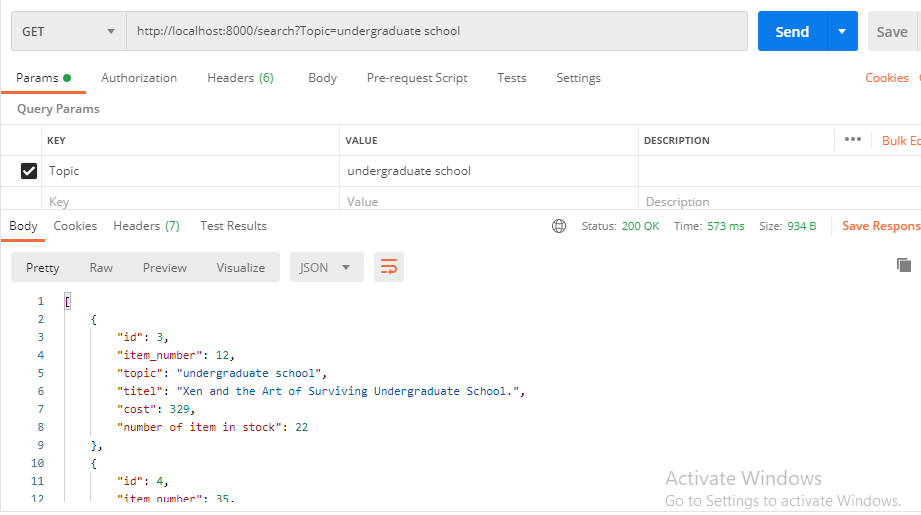
Run a command : docker-compose up

Make sure all the containers run





Then use the postman as a client talk to the front-end server and hit different request:

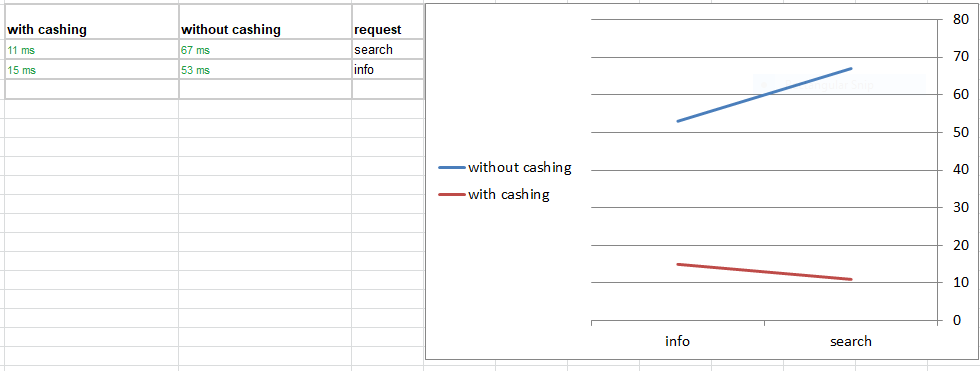


**Possible improvements :**

* Use different techniques which may be more accurate for load balancing. With better results. and, separate the load balancing instead to be built in the front-end itself.
* Use technique different than the invalidate technique in the cache like the update technique.

**Experimental Evaluation and Measurements:**

**1-**

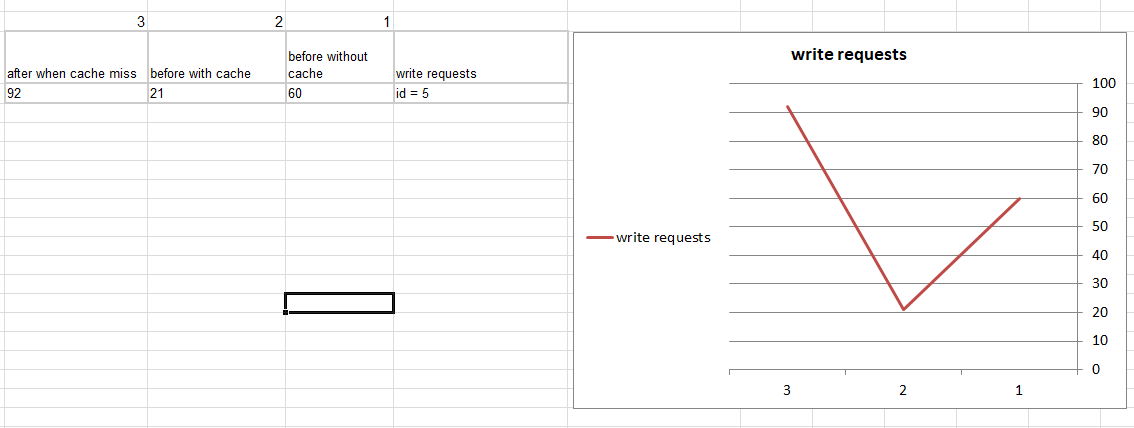
****

**average response time without caching =** 67+53 / 2 = 60ms

**average response time with caching =** 11+15 / 2 = 13ms

As seen the caching help to reduce the response time so the system become faster

**2-**

****

**What are the overhead of cache consistency operations ?** the update request send a invalidate message to the cache to tell the cache that this item is modified and need to be removed from the cache

**What is the latency of a subsequent request that sees a cache m**iss? As seen the latency increase when there is a cache miss.