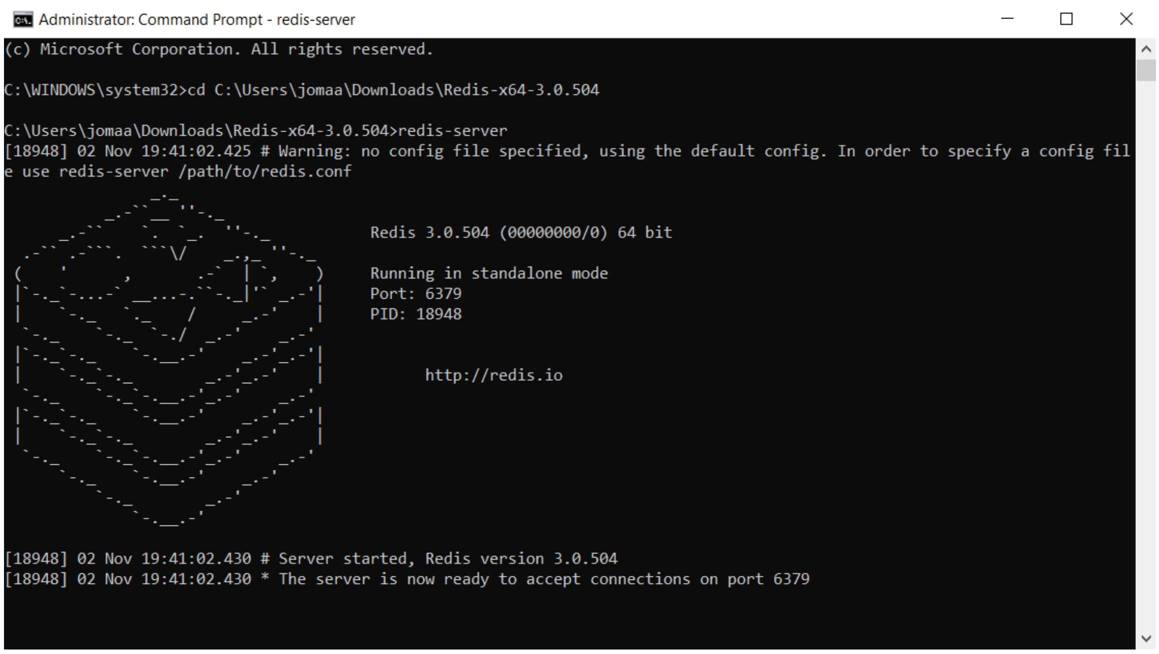
BAZAR.COM

Part2

Batool juma && marwa banifadel

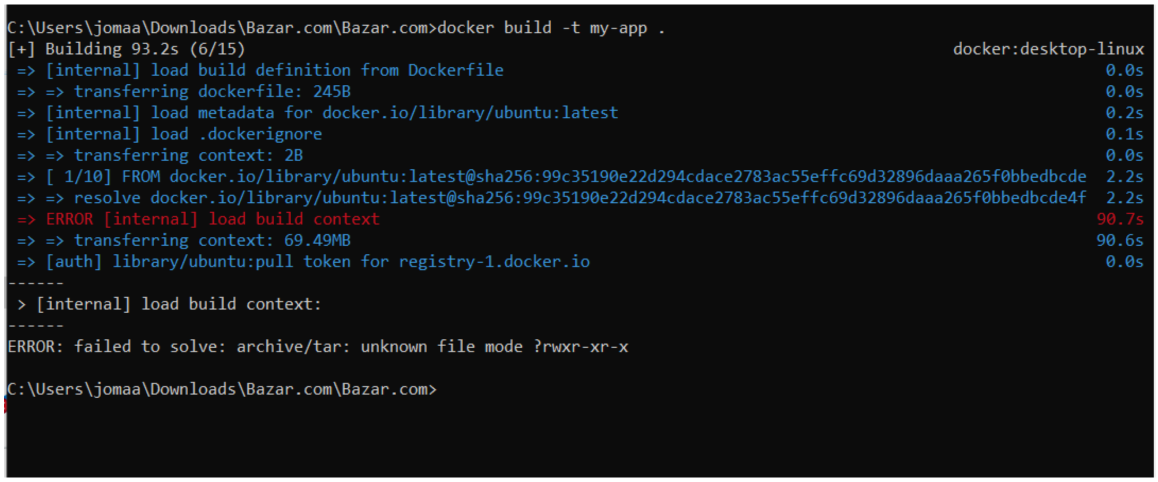
first :

1- run to redis-server at port 6379 as Administrator to save request in cache



```
(c) Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>cd C:\Users\jomaa\Downloads\Redis-x64-3.0.504

C:\Users\jomaa\Downloads\Redis-x64-3.0.504>redis-server
[18948] 02 Nov 19:41:02.425 # Warning: no config file specified, using the default config. In order to specify a config fil
e use redis-server /path/to/redis.conf
                _._
           _.-``__ ''-._
      _.-``    `.  `_.  ''-._           Redis 3.0.504 (00000000/0) 64 bit
  .-`` .-```.  ```\/    _.,_ ''-._
 (    '      ,       .-`  | `,    )     Running in standalone mode
 |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
 |    `-._   `._    /     _.-'    |     PID: 18948
  `-._    `-._  `-./  _.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |           http://redis.io
  `-._    `-._`-.__.-'_.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |
  `-._    `-._`-.__.-'_.-'    _.-'
      `-._    `-.__.-'    _.-'
          `-._        _.-'
              `-.__.-'

[18948] 02 Nov 19:41:02.430 # Server started, Redis version 3.0.504
[18948] 02 Nov 19:41:02.430 * The server is now ready to accept connections on port 6379
```

2- Build image in docker



```
C:\Users\jomaa\Downloads\Bazar.com\Bazar.com>docker build -t my-app .
[+] Building 93.2s (6/15)                                                                          docker:desktop-linux
 => [internal] load build definition from Dockerfile                                                                0.0s
 => => transferring dockerfile: 245B                                                                                0.0s
 => [internal] load metadata for docker.io/library/ubuntu:latest                                                    0.2s
 => [internal] load .dockerignore                                                                                   0.1s
 => => transferring context: 2B                                                                                     0.0s
 => [ 1/10] FROM docker.io/library/ubuntu:latest@sha256:99c35190e22d294cdace2783ac55effc69d32896daaa265f0bbedbcde  2.2s
 => => resolve docker.io/library/ubuntu:latest@sha256:99c35190e22d294cdace2783ac55effc69d32896daaa265f0bbedbcde4f  2.2s
 => ERROR [internal] load build context                                                                            90.7s
 => => transferring context: 69.49MB                                                                               90.6s
 => [auth] library/ubuntu:pull token for registry-1.docker.io                                                       0.0s
------
 > [internal] load build context:
------
ERROR: failed to solve: archive/tar: unknown file mode ?rwxr-xr-x

C:\Users\jomaa\Downloads\Bazar.com\Bazar.com>
```

3- docker network create projectPart1-net –to connect all server in same network

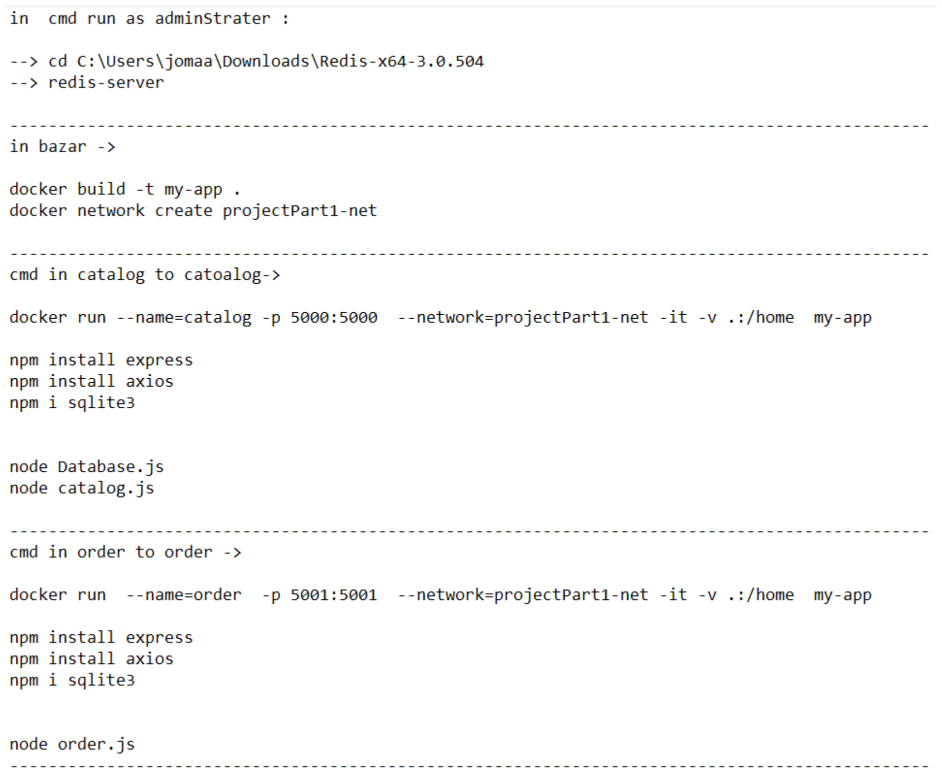
4- run to 5 dockers in this command , each run in a separate cmd.

Ps : The meaning of each line is explained in the submission for the first part.

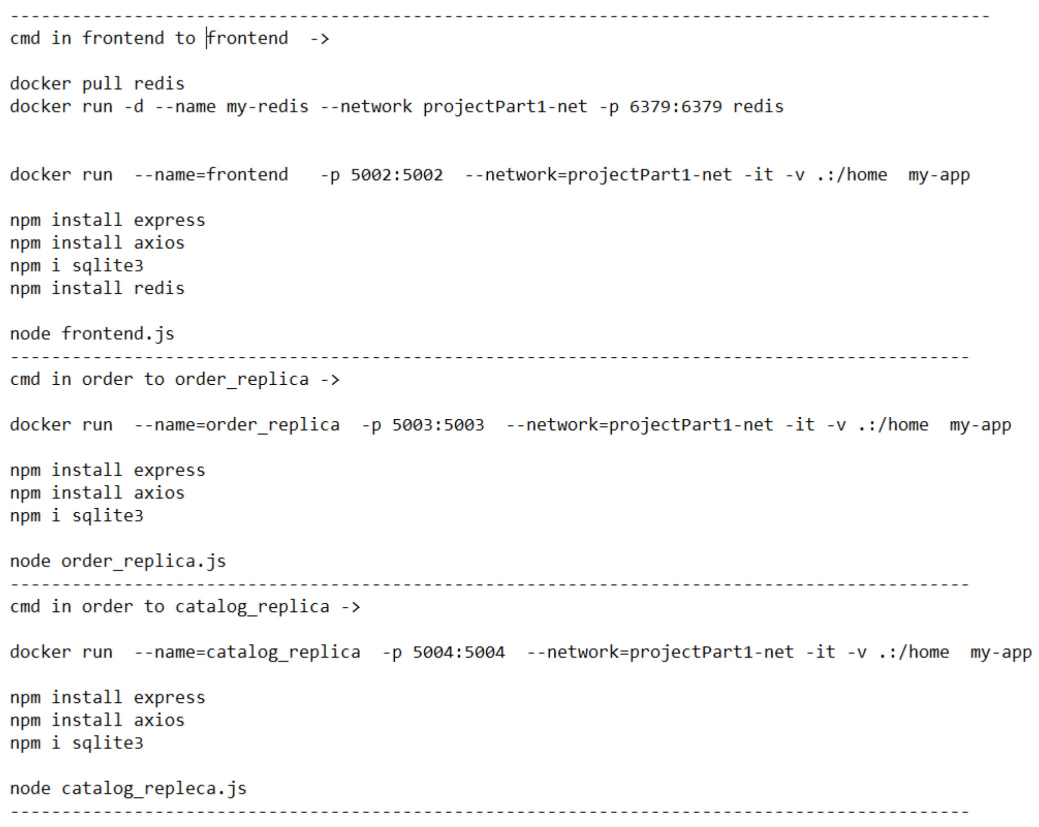
But here 2 servers have been added due to the replica.

- Catalog at port 5000 - order at port 5001 - frontend at port 5002

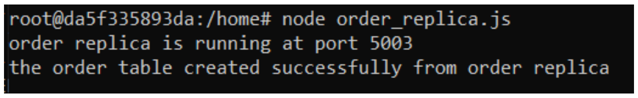
- order_replica at port 5003 - catalog_replica at port 5004



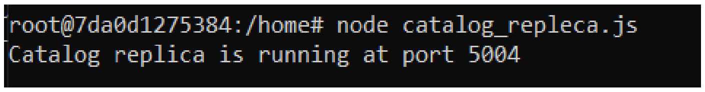
```
in  cmd run as adminStrater :

--> cd C:\Users\jomaa\Downloads\Redis-x64-3.0.504
--> redis-server

-----------------------------------------------------------------------------------------------
in bazar ->

docker build -t my-app .
docker network create projectPart1-net

-----------------------------------------------------------------------------------------------
cmd in catalog to catoalog->

docker run --name=catalog -p 5000:5000  --network=projectPart1-net -it -v .:/home  my-app

npm install express
npm install axios
npm i sqlite3


node Database.js
node catalog.js

-----------------------------------------------------------------------------------------------
cmd in order to order ->

docker run  --name=order  -p 5001:5001  --network=projectPart1-net -it -v .:/home  my-app

npm install express
npm install axios
npm i sqlite3


node order.js
-----------------------------------------------------------------------------------------------
```



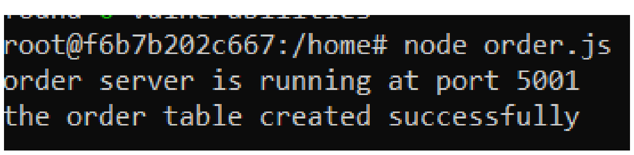
```
-----------------------------------------------------------------------------------------------
cmd in frontend to frontend  ->

docker pull redis
docker run -d --name my-redis --network projectPart1-net -p 6379:6379 redis


docker run  --name=frontend   -p 5002:5002  --network=projectPart1-net -it -v .:/home  my-app

npm install express
npm install axios
npm i sqlite3
npm install redis

node frontend.js
-----------------------------------------------------------------------------------------------
cmd in order to order_replica ->

docker run  --name=order_replica  -p 5003:5003  --network=projectPart1-net -it -v .:/home  my-app

npm install express
npm install axios
npm i sqlite3

node order_replica.js
-----------------------------------------------------------------------------------------------
cmd in order to catalog_replica ->

docker run  --name=catalog_replica  -p 5004:5004  --network=projectPart1-net -it -v .:/home  my-app

npm install express
npm install axios
npm i sqlite3

node catalog_repleca.js
-----------------------------------------------------------------------------------------------
```



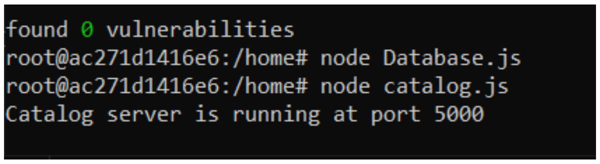
```
root@da5f335893da:/home# node order_replica.js
order replica is running at port 5003
the order table created successfully from order replica
```



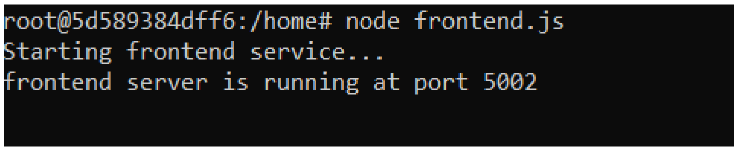
```
root@7da0d1275384:/home# node catalog_repleca.js
Catalog replica is running at port 5004
```



```
root@f6b7b202c667:/home# node order.js
order server is running at port 5001
the order table created successfully
```



```
found 0 vulnerabilities
root@ac271d1416e6:/home# node Database.js
root@ac271d1416e6:/home# node catalog.js
Catalog server is running at port 5000
```



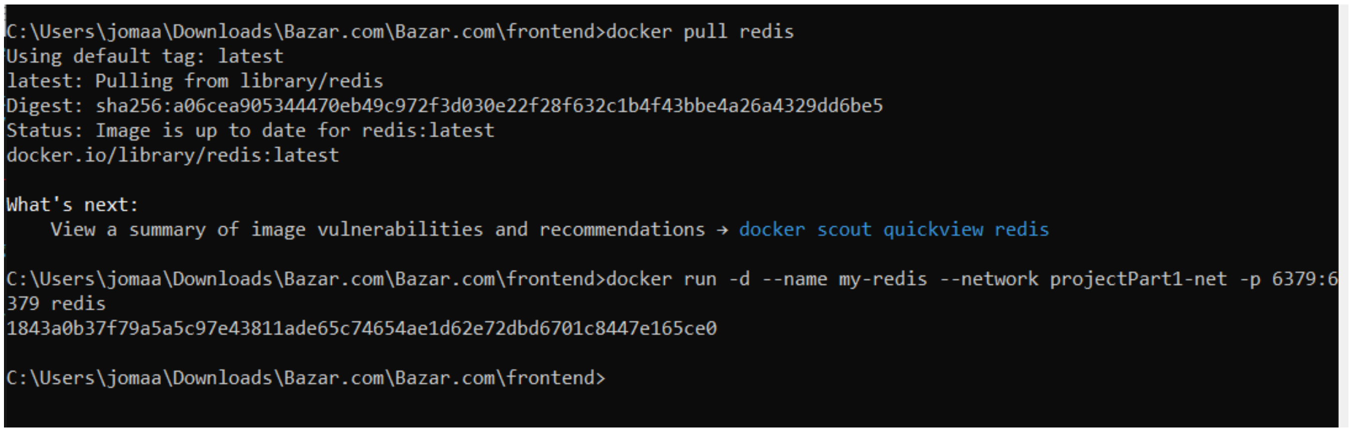
```
root@5d589384dff6:/home# node frontend.js
Starting frontend service...
frontend server is running at port 5002
```

docker pull redis

- This command downloads the Redis image from Docker Hub to your local system if it isn't already downloaded. Redis is an in-memory database, and Docker allows you to run it as a containerized service.

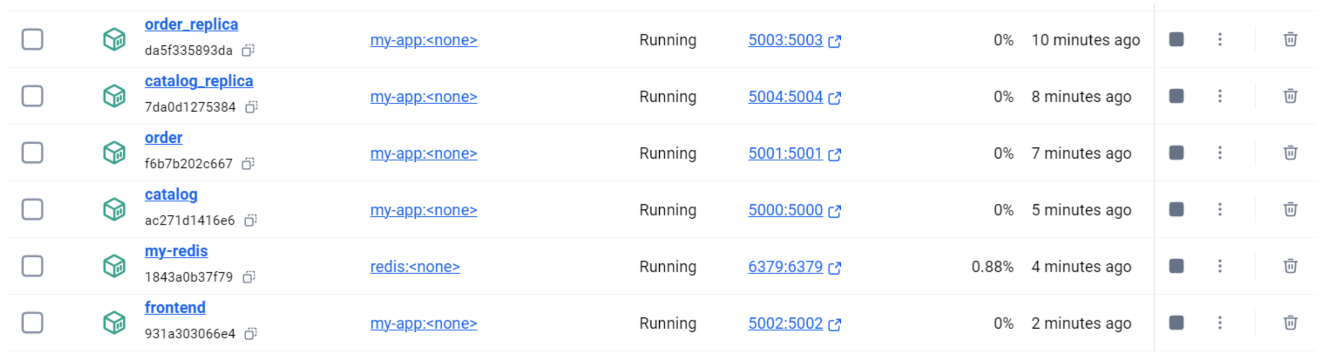
docker run -d --name my-redis --network projectPart1-net -p 6379:6379 redis

- docker run: Starts a new container from a Docker image.



```
C:\Users\jomaa\Downloads\Bazar.com\Bazar.com\frontend>docker pull redis
Using default tag: latest
latest: Pulling from library/redis
Digest: sha256:a06cea905344470eb49c972f3d030e22f28f632c1b4f43bbe4a26a4329dd6be5
Status: Image is up to date for redis:latest
docker.io/library/redis:latest

What's next:
    View a summary of image vulnerabilities and recommendations → docker scout quickview redis

C:\Users\jomaa\Downloads\Bazar.com\Bazar.com\frontend>docker run -d --name my-redis --network projectPart1-net -p 6379:6
379 redis
1843a0b37f79a5a5c97e43811ade65c74654ae1d62e72dbd6701c8447e165ce0

C:\Users\jomaa\Downloads\Bazar.com\Bazar.com\frontend>
```

In docker :



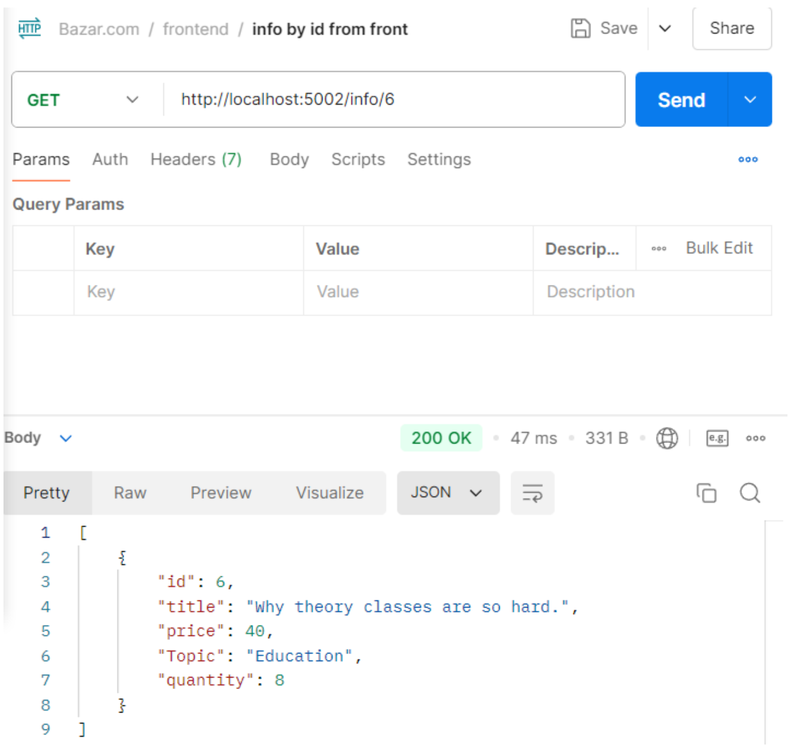
order_replica da5f335893da	my-app:<none>	Running	5003:5003	0%	10 minutes ago	
catalog_replica 7da0d1275384	my-app:<none>	Running	5004:5004	0%	8 minutes ago	
order f6b7b202c667	my-app:<none>	Running	5001:5001	0%	7 minutes ago	
catalog ac271d1416e6	my-app:<none>	Running	5000:5000	0%	5 minutes ago	
my-redis 1843a0b37f79	redis:<none>	Running	6379:6379	0.88%	4 minutes ago	
frontend 931a303066e4	my-app:<none>	Running	5002:5002	0%	2 minutes ago	

Second in Postman :

In the first request for the book, the information will come from the main server. It was confirmed that the book number is not in the cache, then the request is sent and the request is stored in the cache. As for the second request, the search will be done. If it is in the cache, it will come directly from the cache, and this will be done.

As for the copies (replicas), the Round Robin method was used: distributing requests alternately on all servers.

The first request goes to the catalog, and the second request generally goes to the catalog replica if there is information in the cache about the request.

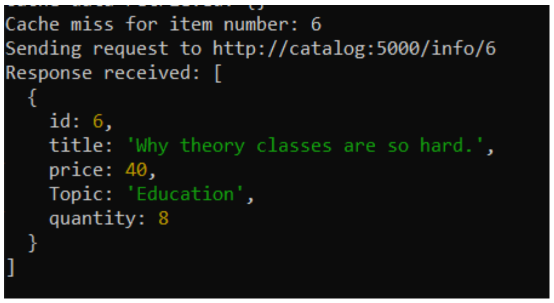


Bazar.com / frontend / info by id from front

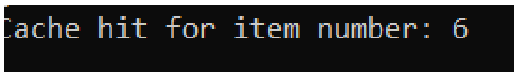
GET http://localhost:5002/info/6

Body 200 OK · 47 ms · 331 B

```
[
    {
        "id": 6,
        "title": "Why theory classes are so hard.",
        "price": 40,
        "Topic": "Education",
        "quantity": 8
    }
]
```

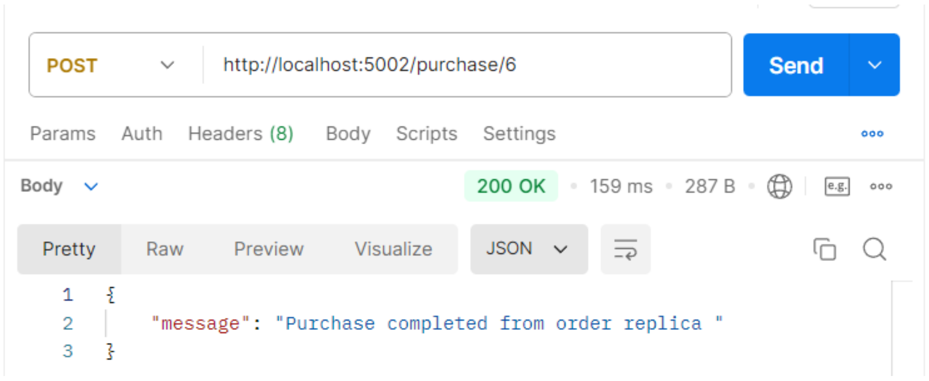


```
Cache miss for item number: 6
Sending request to http://catalog:5000/info/6
Response received: [
  {
    id: 6,
    title: 'Why theory classes are so hard.',
    price: 40,
    Topic: 'Education',
    quantity: 8
  }
]
```

in second request in same URI  :

```
Cache hit for item number: 6
```

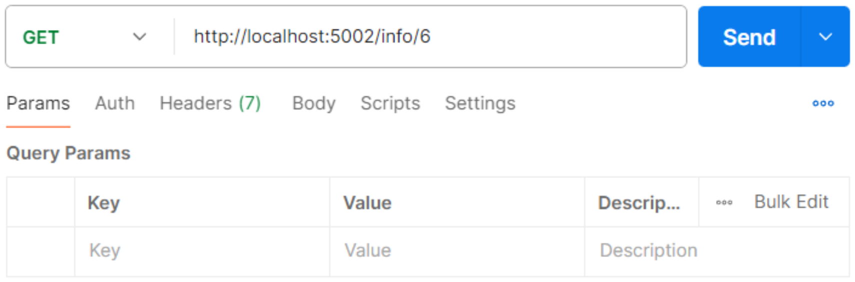
When I make a purchase, the information will go to the order server and will go to the cash to search for this book. If it exists, the cash will be modified so that if the order is re-ordered, the information will come directly from the cash. I have made an automatic deletion of the cash after an hour of storage in it.



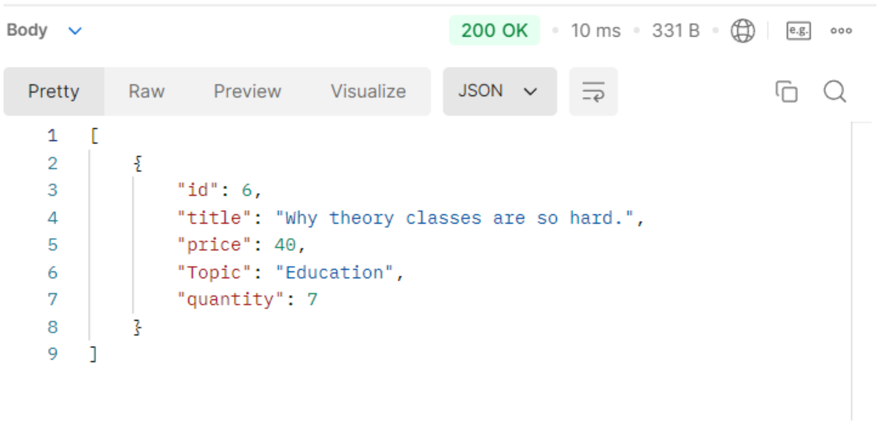
POST http://localhost:5002/purchase/6

200 OK • 159 ms • 287 B

```
{
    "message": "Purchase completed from order replica "
}
```

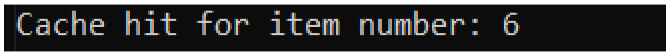


GET http://localhost:5002/info/6



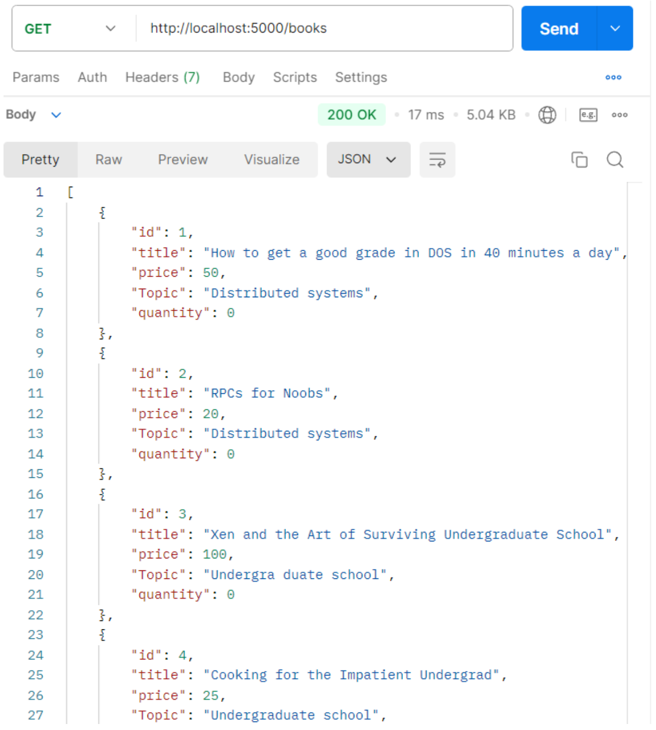
200 OK • 10 ms • 331 B

```
[
    {
        "id": 6,
        "title": "Why theory classes are so hard.",
        "price": 40,
        "Topic": "Education",
        "quantity": 7
    }
]
```



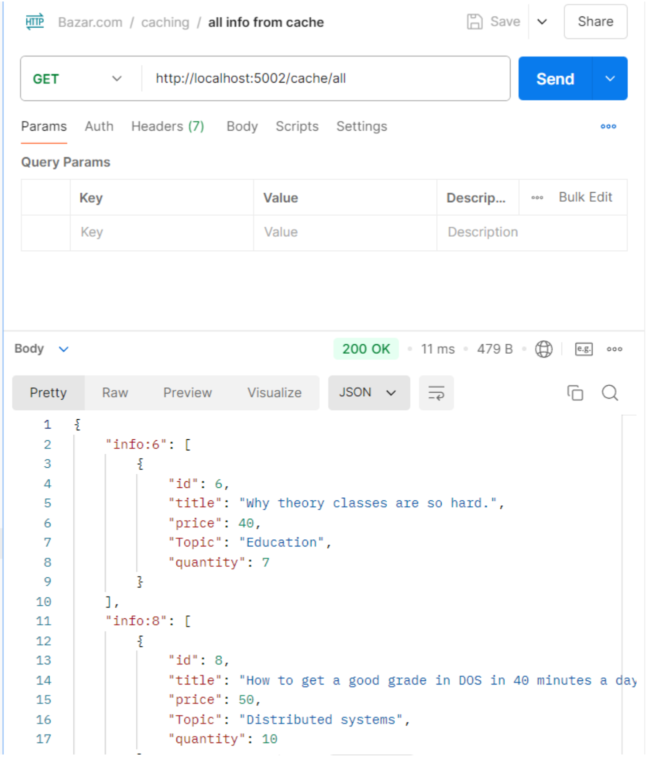
```
Cache hit for item number: 6
```

So I made sure of the basic requirements, now I made an additional URI for the first to empty the cache, the second to see everything in the cache, and the third in the catalog to see all the books



GET http://localhost:5000/books

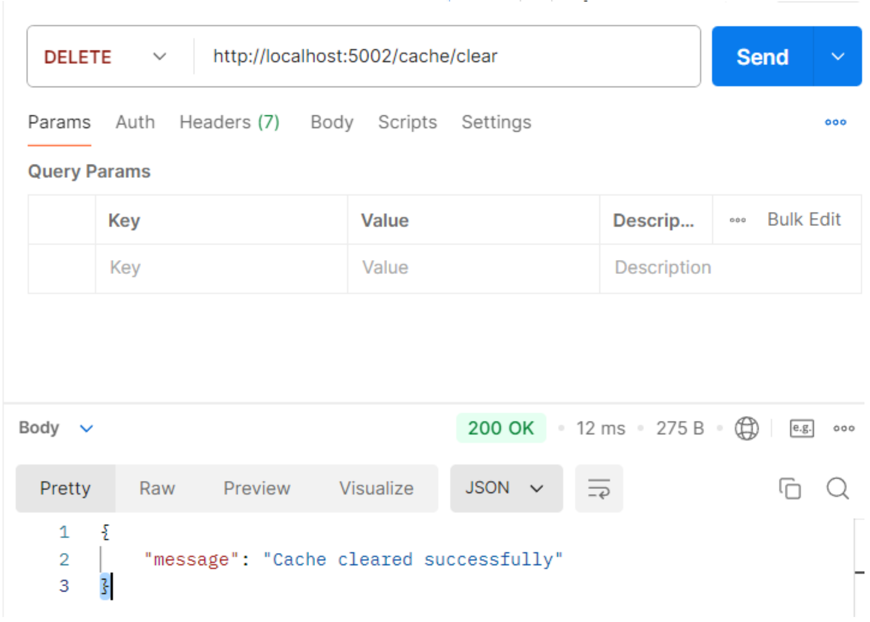
```
[
    {
        "id": 1,
        "title": "How to get a good grade in DOS in 40 minutes a day",
        "price": 50,
        "Topic": "Distributed systems",
        "quantity": 0
    },
    {
        "id": 2,
        "title": "RPCs for Noobs",
        "price": 20,
        "Topic": "Distributed systems",
        "quantity": 0
    },
    {
        "id": 3,
        "title": "Xen and the Art of Surviving Undergraduate School",
        "price": 100,
        "Topic": "Undergra duate school",
        "quantity": 0
    },
    {
        "id": 4,
        "title": "Cooking for the Impatient Undergrad",
        "price": 25,
        "Topic": "Undergraduate school",
```



Bazar.com / caching / all info from cache

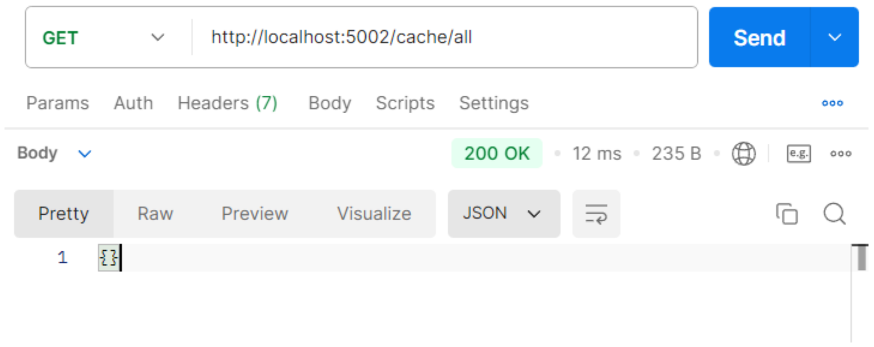
GET http://localhost:5002/cache/all

```
{
    "info:6": [
        {
            "id": 6,
            "title": "Why theory classes are so hard.",
            "price": 40,
            "Topic": "Education",
            "quantity": 7
        }
    ],
    "info:8": [
        {
            "id": 8,
            "title": "How to get a good grade in DOS in 40 minutes a day
            "price": 50,
            "Topic": "Distributed systems",
            "quantity": 10
```



DELETE http://localhost:5002/cache/clear

```
{
    "message": "Cache cleared successfully"
}
```



http://localhost:5002/cache/all

200 OK • 12 ms • 235 B

```
{}
```

About the libraries I used:

1. **Express**

- **Purpose**: Express is a web application framework for Node.js that helps you build web applications and services quickly and easily.
- **Uses**:
  - **Creating HTTP Servers**: You can quickly set up HTTP servers with custom settings.
  - **Route Management**: It allows easy management of routes, letting you define specific responses for each route.
  - **Middleware**: Supports the use of middleware to process requests.
  - **Request Distribution**: Can be used with features like request routing and load balancing.

2. **Axios**

- **Purpose**: Axios is a promise-based HTTP client for the browser and Node.js, used for making HTTP requests.
- **Uses**:
  - **Making GET and POST Requests**: Easily make requests to APIs.
  - **Handling Responses**: Provides a simple interface to deal with responses, including data and errors.
  - **Custom Settings**: Allows customization of request settings such as headers and data.

3. **SQLite3**

- **Purpose**: SQLite3 is a library that provides an interface to interact with SQLite databases, which are lightweight and self-contained.
- **Uses**:
  - **Creating Local Databases**: Easily create local databases to store data.
  - **Executing SQL Queries**: Use SQL queries to interact with the data, such as inserting, reading, updating, and deleting.
  - **Flexibility**: SQLite is a good option for small to medium-sized applications where a separate database server is not needed.

4. **Redis**

- **Purpose**: Redis is an in-memory data structure store used as a database, cache, and message broker.
- **Uses**:
  - **Caching**: Use Redis to temporarily store data to improve application performance.
  - **Session Storage**: Suitable for storing user session information.
  - **Message Queuing**: Can be used to create queuing systems and inter-service communication.
  - **Support for Advanced Data Structures**: Redis supports advanced data types like lists, sets, and hashes.

The Round Robin method is a simple and effective technique for distributing loads among servers. It gives each server an equal chance to handle requests, preventing any single server from becoming overwhelmed while others remain underutilized. I chose to use Round Robin for my project because my requests are few and straightforward, making it a better fit for my needs. This method is easy to implement and configure, which also helps improve performance and reduce waiting times. Overall, Round Robin is an ideal option for balancing loads efficiently and fairly in my project.

Here's an overview of the general architecture of your system, including its various components such as the frontend server, replication, and caching:

1. Frontend Server

The frontend server acts as the main interface for users. It is built using **Express.js** and runs on port 5002. This server handles incoming requests and coordinates communication between the client and the backend services (catalog and order services).

- **Request Handling**: It processes requests for searching items, retrieving item information, and handling purchases.
- **Load Balancing**: It implements round-robin load balancing to distribute requests across multiple replicas of the catalog and order services.
- **Caching**: Utilizes **Redis** for caching responses to minimize database hits and improve performance. The cached data is stored for one hour.

2. Catalog Service

The catalog service is responsible for managing the catalog of items available for purchase. It runs on port 5000 and handles requests related to searching for items and retrieving item details.

- **Database Operations**: Interacts with an SQLite database to perform CRUD operations on the catalog.
- **Endpoints**: Exposes endpoints for searching by topic, fetching all books, and retrieving item details.

3. Order Service

The order service handles purchase requests and manages order data. It operates on port 5001 and communicates with the catalog service to update item quantities upon successful purchases.

- **Order Management**: Inserts new orders into the database and retrieves existing orders.
- **Integration with Catalog**: Sends requests to the catalog service to check item availability and update quantities after a purchase.

4. Replication

Both the catalog and order services have replicas to enhance reliability and availability. This setup ensures that if one service instance fails, others can still handle requests.

5. Database

The SQLite database stores the catalog information, including item titles, prices, topics, and quantities. The database schema includes tables for catalog items and orders, supporting the functionalities of both the catalog and order services.

6. Caching

Redis is used as a caching layer to store frequently accessed data. This reduces the need for repeated database queries and speeds up response times. The cache can be cleared through a dedicated endpoint.

How the system works:

When a request is made by the user, the Friday request is directed to the server. The client relies on directing the requests to the original copy of the catalog service or the order service using the Round Robin method and also partially distributing the load. After the request, the data is stored in the cache to speed up retrieval in future requests.

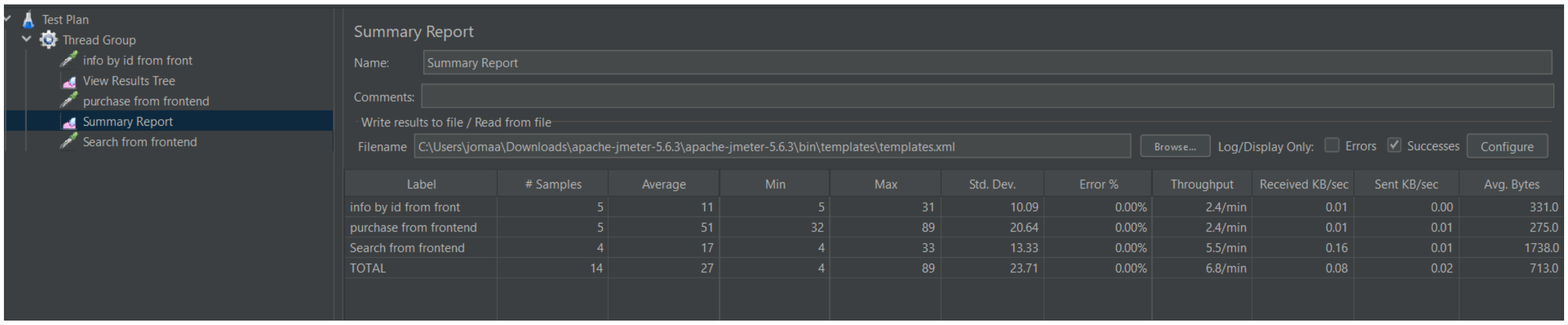
Possible improvements:

1- Improved user interface
2- Increased security
3- Increased information
4- Increased replica

Performance Results

I did a test on **jmeter** and these were the readings:

Ps: download JMETER from : https://jmeter.apache.org/download_jmeter.cgi in first run :



Summary Report

Name: Summary Report

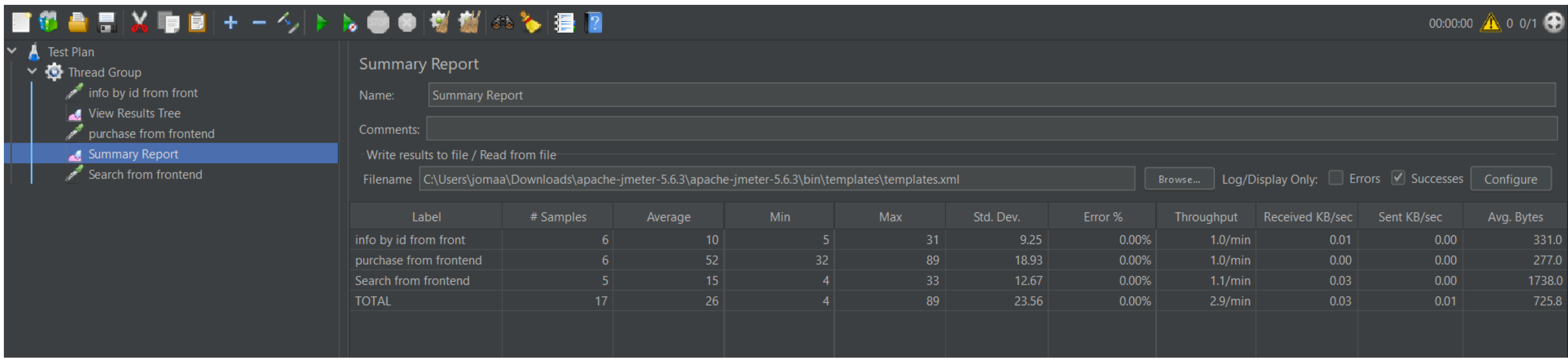
Comments:

Write results to file / Read from file

Filename C:\Users\jomaa\Downloads\apache-jmeter-5.6.3\apache-jmeter-5.6.3\bin\templates\templates.xml Browse... Log/Display Only: Errors ✓ Successes Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
info by id from front	5	11	5	31	10.09	0.00%	2.4/min	0.01	0.00	331.0
purchase from frontend	5	51	32	89	20.64	0.00%	2.4/min	0.01	0.01	275.0
Search from frontend	4	17	4	33	13.33	0.00%	5.5/min	0.16	0.01	1738.0
TOTAL	14	27	4	89	23.71	0.00%	6.8/min	0.08	0.02	713.0

in second run :



Summary Report

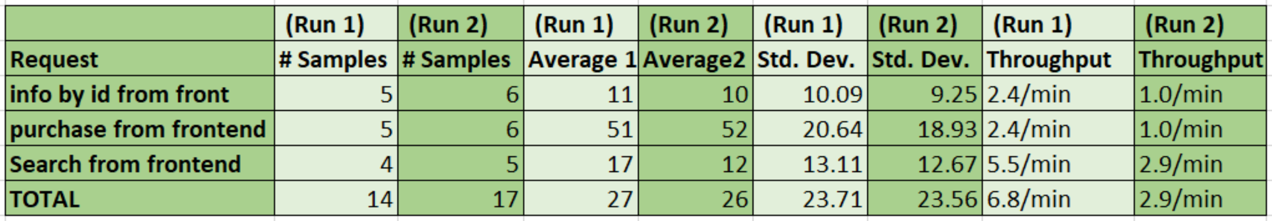
Name: Summary Report

Comments:

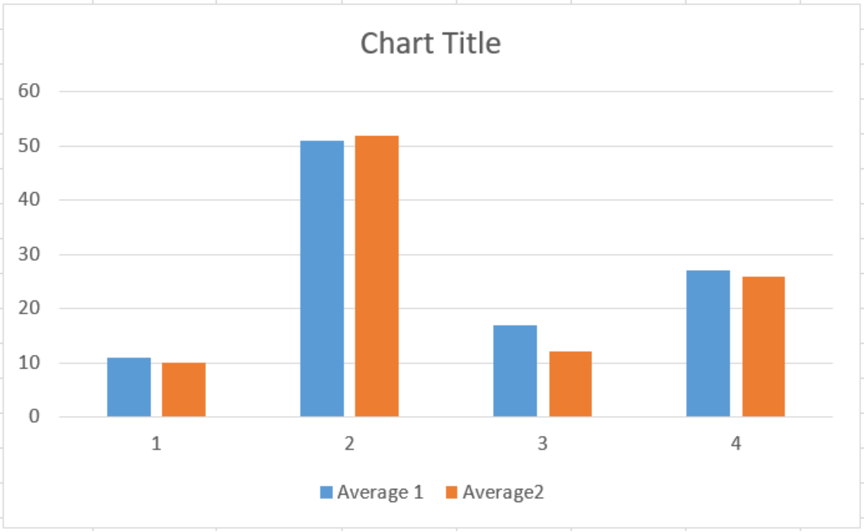
Write results to file / Read from file

Filename C:\Users\jomaa\Downloads\apache-jmeter-5.6.3\apache-jmeter-5.6.3\bin\templates\templates.xml Browse... Log/Display Only: Errors ✓ Successes Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
info by id from front	6	10	5	31	9.25	0.00%	1.0/min	0.01	0.00	331.0
purchase from frontend	6	52	32	89	18.93	0.00%	1.0/min	0.00	0.00	277.0
Search from frontend	5	15	4	33	12.67	0.00%	1.1/min	0.03	0.00	1738.0
TOTAL	17	26	4	89	23.56	0.00%	2.9/min	0.03	0.01	725.8



	(Run 1)	(Run 2)	(Run 1)	(Run 2)	(Run 1)	(Run 2)	(Run 1)	(Run 2)
Request	**# Samples**	**# Samples**	**Average 1**	**Average2**	**Std. Dev.**	**Std. Dev.**	**Throughput**	**Throughput**
info by id from front	5	6	11	10	10.09	9.25	2.4/min	1.0/min
purchase from frontend	5	6	51	52	20.64	18.93	2.4/min	1.0/min
Search from frontend	4	5	17	12	13.11	12.67	5.5/min	2.9/min
TOTAL	14	17	27	26	23.71	23.56	6.8/min	2.9/min



Interpretation of the results:

1. Number of samples (# Samples): The number of samples increased slightly in the second run, which means that the number of targeted operations was slightly larger.

2. Average response time (Average): The average response time did not change significantly between the two runs. However, it decreased slightly in the second run for the “info by id from front” and “Search from frontend” requests, indicating a slight improvement in performance for these requests.

3. Throughput: The Throughput for each request appears to have decreased in the second run, which could be due to the increased load or the number of samples.

4. Standard error (Std. Dev.): The values decreased slightly, which could indicate a relative stability in performance.

Overall, performance appears to be fairly stable but slightly slower in the second run, which may require optimizing Throughput to ensure that requests are fulfilled faster, I expected the second run to have better performance.