



An-Najah National University
Computer Engineering Department
Distributed Systems - DOS

Project – Part1 Report

Shams Abd Al-Aziz
Yara Daraghmeh

Dr. Samer Arandi

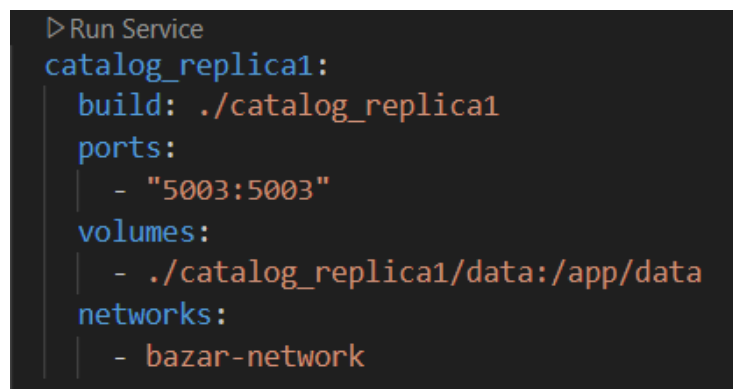
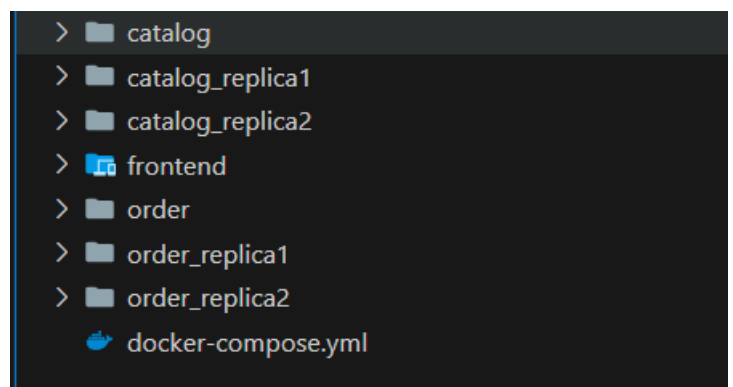
Introduction:

In the previous lab we learned how to make microservices, dealing with REST APIs and how to use docker, and we make online book shop, but in this part, we add an in-memory cache mechanism, a load balancing algorithm, server replicas, and providing synchronization between these replicas to improve full application, improve request processing latency and increase the availability and remove single point of failure.

Discussion:

- 1- Create Replicas of Catalog and Order Servers: we create 2 replicas for each service (server in docker), by the command:
`cp -r catalog catalog_replica1` (and do the same for the others)
So, everything inside catalog server (code and data) is copied, then we added some functionalities in all services.

Then we update the docker compose file, and add all new services, each one connected on different port.



- 2- Frontend: Load Balancer + Cache:

Load balancing is done with a simple round-robin mechanism to spread requests across replicas. The Round Robin algorithm has been chosen for its simplicity and effectiveness.

At the first we defined:

```
# Round-robin index tracking
catalog_index = 0
order_index = 0
```

Then, we defined this function to pick the next server URL, and do the same for order URLs.

```
def get_next_catalog_url():
    global catalog_index
    url = catalog_replicas[catalog_index]
    catalog_index = (catalog_index + 1) % len(catalog_replicas)
    return url
```

after that we use it like this inside search function, get info function for catalog service and purchase a book using order service:

```
catalog_url = get_next_catalog_url()
try:
    response = requests.get(f"{catalog_url}/search/{topic}")
    result = parse_response(response.json())
```

then, we add **an in-memory cache** to improve request processing latency. An in-memory cache object (cache) has been added to the application. This object stores data keyed by the search books of topic and get info of item queries. When a search or get info is performed, the cache is checked first, if the data is present, it is directly used; otherwise, a new API call fetches the data and updates the cache to add the new response of the request.

We defined:

```
search_topic_cache = {}
info_item_cache = {}
```

Then we check inside app.py in frontend server:

And do the same for info query.

```
@app.route('/api/search/<topic>', methods=['GET'])
def search(topic):
    start = time.time()

    if topic in search_topic_cache:
        duration = time.time() - start
        resp = make_response(jsonify({
            "results": search_topic_cache[topic],
            "duration": duration
        }))
        resp.headers['X-Cache-Hit'] = 'true'
        return resp
```

Then we add other caching features such as a limit on the number of items in the cache and use LRU as cache replacement policy to replace older items with newer ones.

So, we define this function for search cache and do the same steps for info cache:

to check if the cache after added item is reach the max size or not, if it does then pop the first item added (oldest).

```
# Caches
SEARCH_CACHE_LIMIT = 10
def search_enforce_cache_limit(cache):
    while len(cache) > SEARCH_CACHE_LIMIT:
        # Remove the first inserted (oldest) item
        cache.pop(next(iter(cache)))
```

After doing the query, and get the results, just add it to cache and check cache size.

```
catalog_url = get_next_catalog_url()
try:
    response = requests.get(f"{catalog_url}/search/{topic}")
    result = response.json()
    if response.status_code == 200:
        search_topic_cache[topic] = result
        search_enforce_cache_limit(search_topic_cache)
```

3- Data Constancy between replicas :

Strong consistency is enforced using a server-push invalidation mechanism. When a replica handles a write operation—such as a buy—it first sends an invalidation request to the front-end cache and before it send it we checked the book quantity on the data base . The front-end server immediately removes the relevant cache entries, ensuring all future reads retrieve up-to-date data.

```
def purchase(item_id):
    catalog_url = get_next_catalog_url()
    response = requests.get(f"{catalog_url}/info/{item_id}")

    if response.status_code != 200:
        return jsonify({'error': 'Book not found'}), 404

    book_info = response.json()
    if book_info['quantity'] <= 0:
        return jsonify({'error': 'Book out of stock'}), 400
```

```
# send invalidate request to frontend
invalidate_response = requests.post(f"http://frontend:5002/invalidate/{item_id}")
if invalidate_response.status_code != 200:
    return jsonify({'error': 'Failed to invalidate cache'}), 500

#send update req to catalog
update_url= f"http://catalog:5000/update/{item_id}"
update_response = requests.put(
    update_url,
    json={'quantity_change': -1},
    headers={'Content-Type': 'application/json'})
)
```

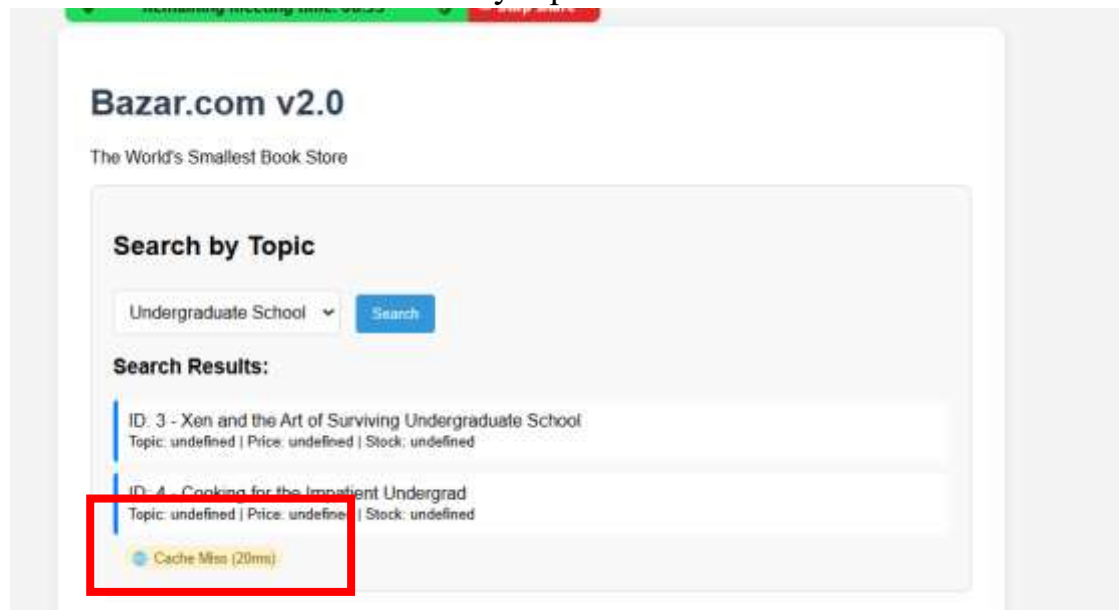
Then, it sends update request to catalog server to change the quantity in the stock on there database and catalog it self sends update requests to their replicas

```
#send update req to catalog replica 1
update_url1 = f"http://catalog_replica1:5003/update/{item_id}"
update_response1 = requests.put(
    update_url1,
    json={'quantity_change': -1},
    headers={'Content-Type': 'application/json'})
)
if update_response1.status_code != 200:
    return jsonify({'error': 'Failed to update catalog replica 1'}), 500
```

Then it updates its local database, replicates the changes to other replicas, and confirms the transaction.

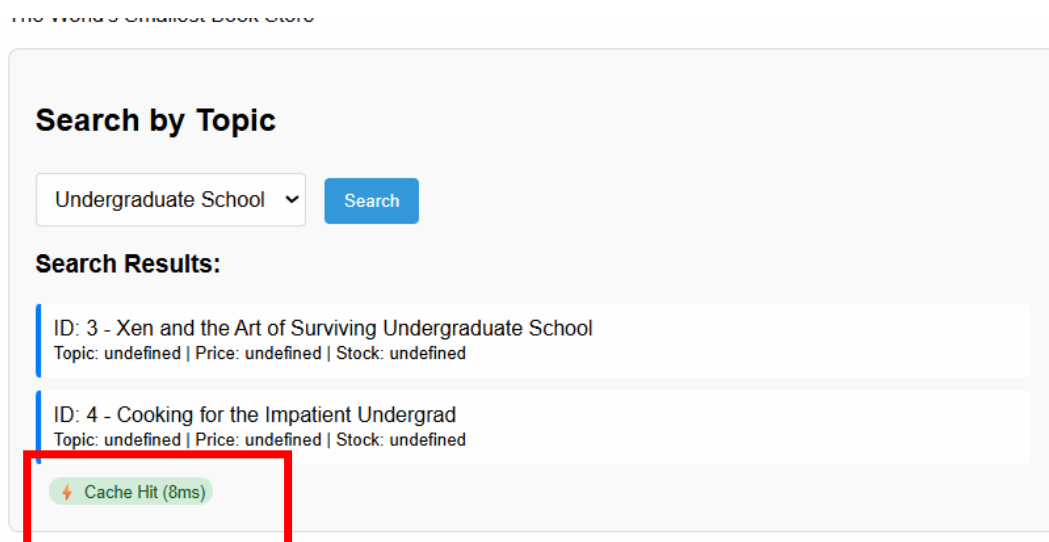
Running the App :

- 1- The first Query using the frontend service
 - We searched about the books by topic



As expected since this was the first query it must be a cache miss and it took a bit longer time to be brought 128ms since it wasn't in the frontend cache .

Now we going to try it again and it must be a hit with much less time



We are talking about **2.5 times improvement**.

Now What we will do is to inform about a book and then invalidate it in the cache by changing on its data for example by purchasing a book so if we requested again about the book it must be a cache miss since its invalidated in the frontend service cache .

1-


Book Information

Get Info

How to get a good grade in DOS in 40 minutes a day

Price: \$40

In Stock: 8

 Cache Miss (12ms)

2-


Book Information

Get Info

How to get a good grade in DOS in 40 minutes a day

Price: \$40

In Stock: 8

 Cache Hit (7ms)

till now its valid in the cache in the next step we will invalidate it by purchasing it so its data will be changed by other replicas and this replica will send the front a invalidation request

3-

Purchase Book

Purchase

Successfully purchased: How to get a good grade in DOS in 40 minutes a day. Order ID: 14

4-


Book Information

Get Info

How to get a good grade in DOS in 40 minutes a day

Price: \$40

In Stock: 7

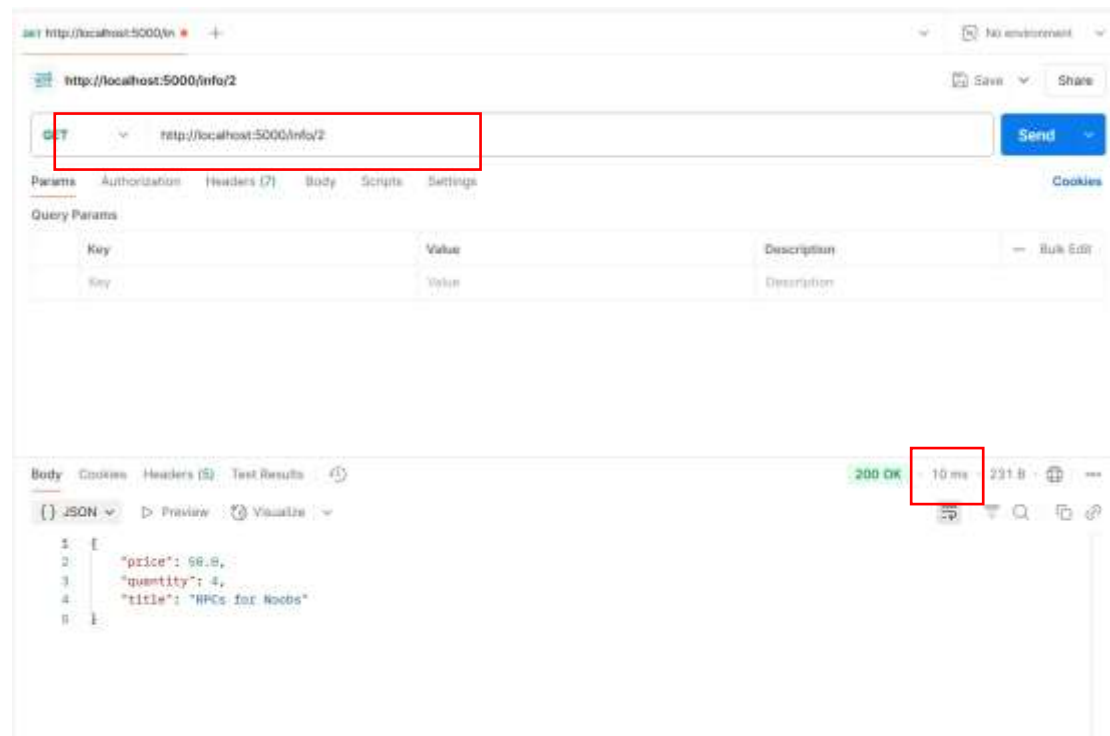
 Cache Miss (18ms)

Now lets go more deep and ask the for the same service from the front and we will get reply from different replicas depending on the load balancing algorithm we used we will focus in one service but we also applied it for other services

```
frontend-1 | * Debugger PIN: 114-631-598
catalog-1   | 172.21.0.8 - - [27/May/2025 20:25:53] "GET /search/distributed%20systems HTTP/1.1" 200 -
frontend-1  | 172.21.0.1 - - [27/May/2025 20:25:53] "GET /api/search/distributed%20systems HTTP/1.1" 200 -
catalog_replica1-1 | 172.21.0.8 - - [27/May/2025 20:26:03] "GET /search/undergraduate%20school HTTP/1.1" 200 -
frontend-1   | 172.21.0.1 - - [27/May/2025 20:26:03] "GET /api/search/undergraduate%20school HTTP/1.1" 200 -
catalog_replica2-1 | 172.21.0.8 - - [27/May/2025 20:26:12] "GET /search/artificial%20intelligence HTTP/1.1" 200 -
frontend-1   | 172.21.0.1 - - [27/May/2025 20:26:12] "GET /api/search/artificial%20intelligence HTTP/1.1" 200 -
frontend-1   | 172.21.0.1 - - [27/May/2025 20:26:26] "GET /api/search/artificial%20intelligence HTTP/1.1" 200 -
```

For the first search request the frontend sent for catalog-1 server then for the second different search request the front end sent for catalogreplica1 server then we repeated it and it asked from replica two , and for cache testing the last request was stored in the cache so when we queried it was returned from the front end , and the same logic is applied for order service .

Also we included a backend test we used postman to test our servers :



Question's Answers :

1- What are the overhead of cache consistency operations?

Cache invalidation required a separate HTTP POST request to `/invalidate/<item_id>`, adding more time for extra requesting thus it makes congestion on the network and it also needs more memory for caching

2- What is the latency of a subsequent request that sees a cache miss?

A cache miss resulted in a latency of the request. The higher delay comes from contacting backend services. And when it hits, time is much lower

Project Dockerization :

I built my project from scratch using Docker. Each service runs in its own container with a dedicated port, and I use volumes to share data between the host OS and the Docker containers. To simplify deployment, I created a **docker-compose.yml** file that spins up all services at once with a single command. You can find it in the repo link .

For running the project :

```
docker-compose build --no-cache
```

```
docker-compose up
```

after finishing we down the containers :

```
docker-compose down
```