An-Najah National University Faculty of Engineering

Computer Engineering Department

Distributed Operation Systems

Lab 2: Turning the Bazar into an Amazon: Replication, Caching and Consistency

Roa Abo Libdeh, 12028262

Noor Sayeh, 12027678

Introduction

To improve performance and reliability under increased load, we containerized the Bazar.com system and added support for replication and caching. Using Docker and Docker Compose, each service (frontend, catalog, order) runs in its own container. The frontend handles caching for read requests and distributes traffic to multiple replicated catalog and order servers to balance the load and ensure fault tolerance.

Materials

To complete this lab, we used the following:

- Docker Desktop
- Docker Compose
- Node.js (Express)
- Postman (Test API)

Architecture

In the dockerized system, **each service runs in its own Docker container**, enabling independent development and deployment. The architecture includes the following services:

---

1. Frontend Container

- Acts as the central entry point for all client interactions.
- Integrates:
    - **In-memory cache** to handle read (query) requests efficiently.
    - **Load balancing logic** (**round robin**)to distribute requests across backend replicas.
- Forwards write requests directly to order/catalog replicas.
- Connects with all replicas over Docker's internal network.

Docker Implementation:

- Dockerfile defines a lightweight Node.js container.
- Exposes port 2000.
- Interacts with backend containers using service names defined in docker-compose.yml.

---

2. Catalog Service Replicas (catalog1, catalog2)

- Each replica serves book-related data (title, price, stock).
- Handles GET (query) and PUT (update) operations.
- Uses a replication protocol to synchronize book data between replicas.
- Sends **cache invalidation messages** to the frontend before database updates.

Docker Implementation:

- Two containers created from the same image.
- Use distinct ports (e.g., 2001, 2003) and environment variables or volumes for isolated behavior.

---

3. Order Service Replicas (order1, order2)

- Handles buy operations and reduces book stock accordingly.
- Communicates with catalog services to verify and update stock.
- Triggers cache invalidation in the frontend upon successful orders.

Docker Implementation:

- Same image, different containers.

Each exposed on a different port (2002, 2004).

Implementation

Each service (frontend, catalog, order) was containerized using a Dockerfile and managed with Docker Compose. Replicas were created by running multiple containers of the same service with different ports. The frontend includes an in-memory cache and uses REST calls for load balancing and cache invalidation. All services were built with Node.js and Express.js, and tested using Postman. Book data remains in a shared DB.js file for simplicity.

How to Run

Start Docker Desktop, then open a terminal in the project folder and run:

docker-compose up –build

Running the command docker-compose up --build will build and start all the services: frontend, catalog replicas, and order replicas within the same Docker network, allowing them to communicate via internal ports

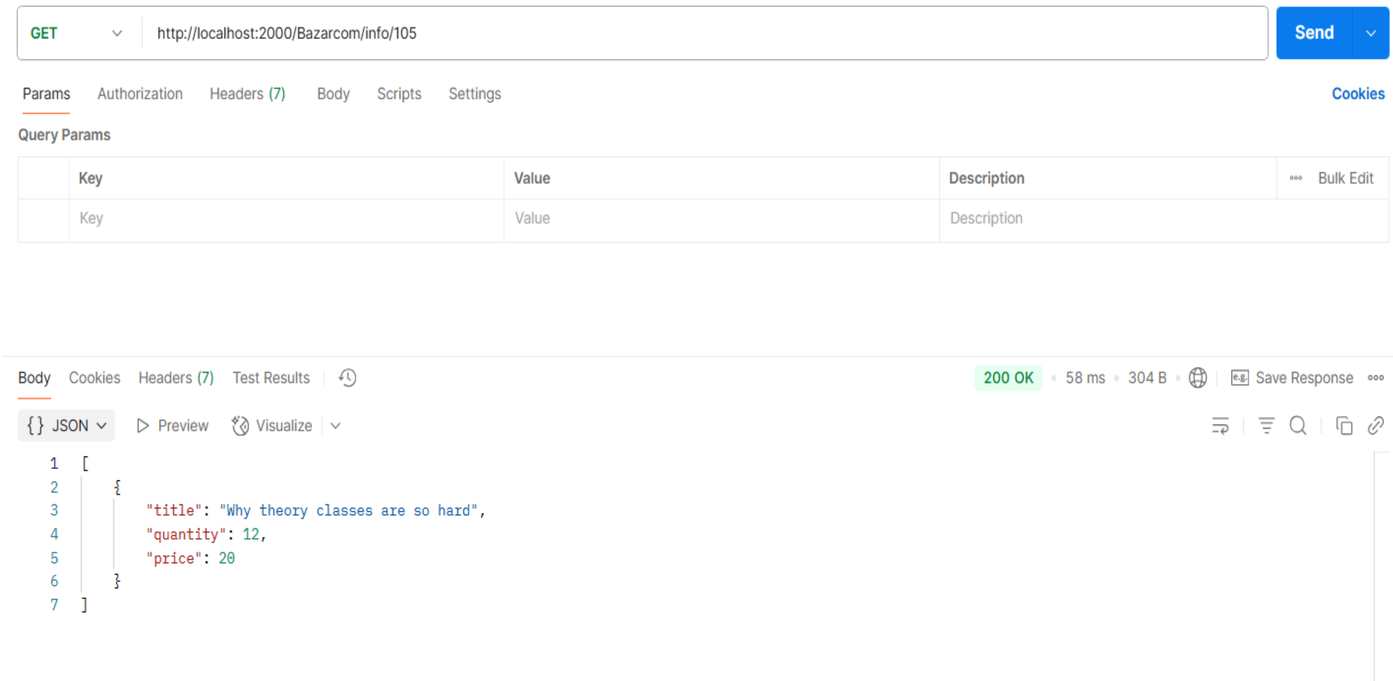
```
[+] Running 10/10
 ✓ catalog1                              Built
 ✓ catalog2                              Built
 ✓ frontend                              Built
 ✓ order1                                Built
 ✓ order2                                Built
 ✓ Container bazar-bookstore-frontend-1  Recreated
 ✓ Container bazar-bookstore-catalog1-1  Recreated
 ✓ Container bazar-bookstore-order2-1    Recreated
 ✓ Container bazar-bookstore-order1-1    Recreated
 ✓ Container bazar-bookstore-catalog2-1  Recreated
Attaching to catalog1-1, catalog2-1, frontend-1, order1-1, order2-1
catalog2-1  | [Catalog] Service running on port 2001
catalog1-1  | [Catalog] Service running on port 2001
order1-1    | Order service running on port 2002
order2-1    | Order service running on port 2002
frontend-1  | [Frontend] Service running on port 2000
```

Testing the APIs using API test tool

We tested APIs using Postman:

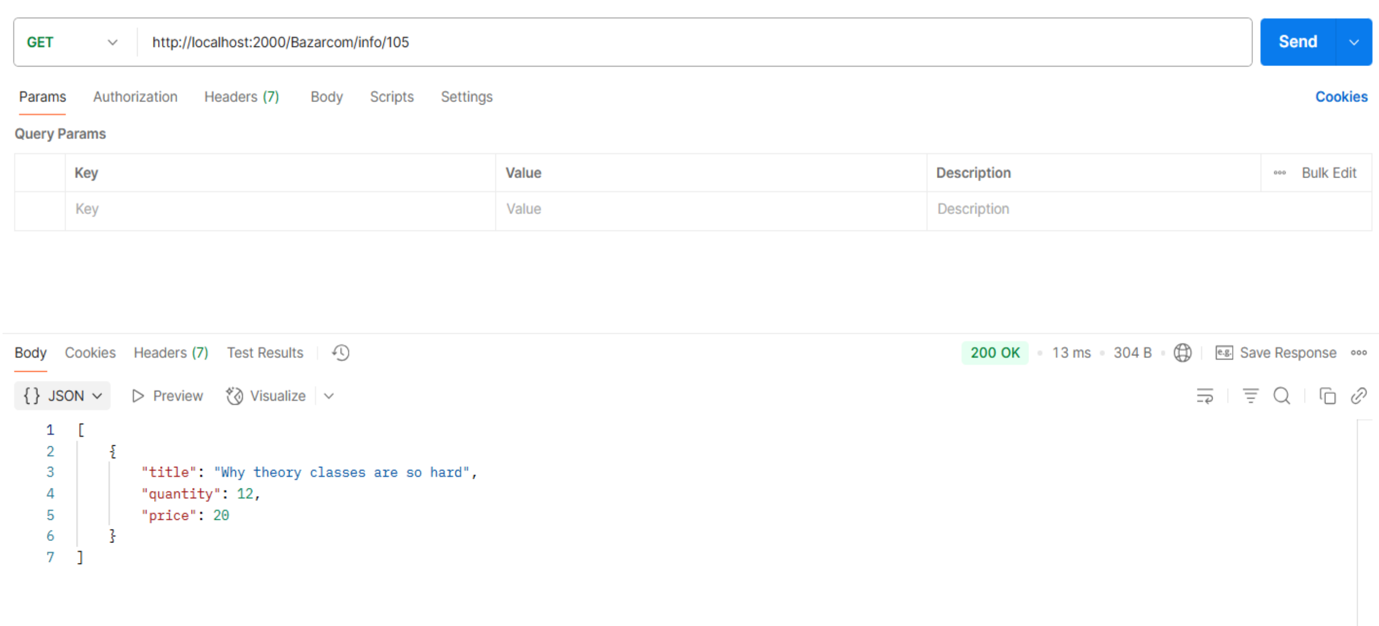
The frontend received a query request for book ID 105. Since it was not found in the cache (cache miss**)it took 58ms to response, the request was forwarded to** Catalog 1 **using a** round-robin load balancing **strategy to retrieve fresh data.**

```
 ✓ Container bazar-bookstore-order1-1    Recreated
Attaching to catalog1-1, catalog2-1, frontend-1, order1-1, order2-1
catalog2-1  | [Catalog] Service running on port 2001
order1-1    | Order service running on port 2002
catalog1-1  | [Catalog] Service running on port 2001
order2-1    | Order service running on port 2002
frontend-1  | [Frontend] Service running on port 2000
frontend-1  | [Frontend] GET /Bazarcom/info/105
frontend-1  | [Frontend] Cache miss
catalog1-1  | [Catalog] GET /CatalogServer/query
```

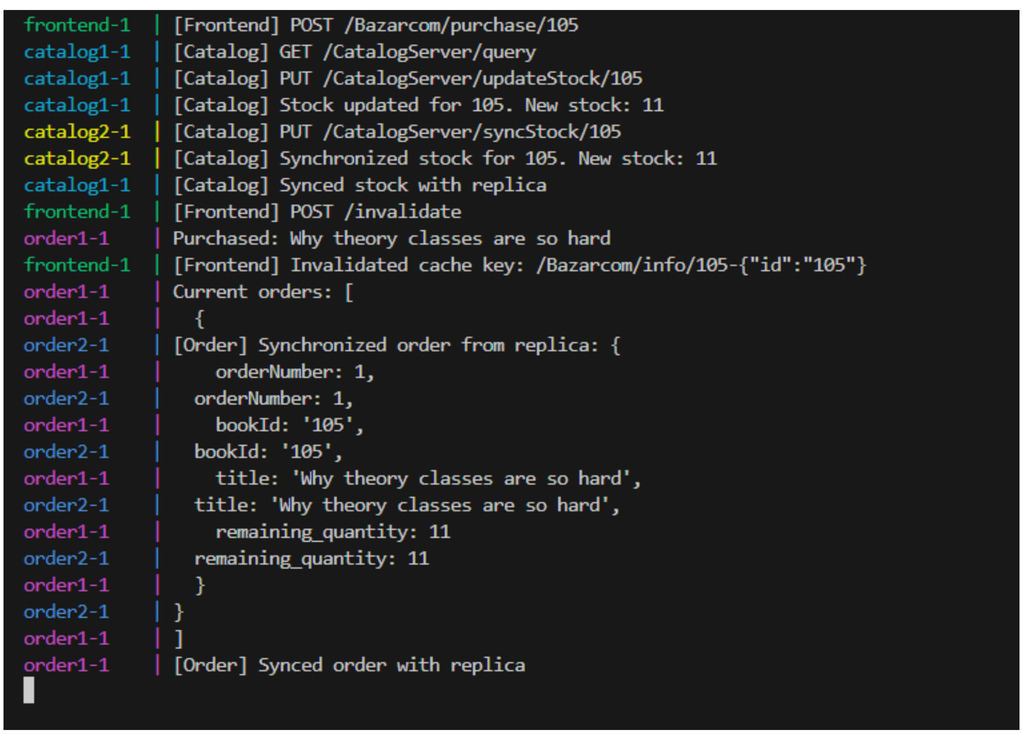


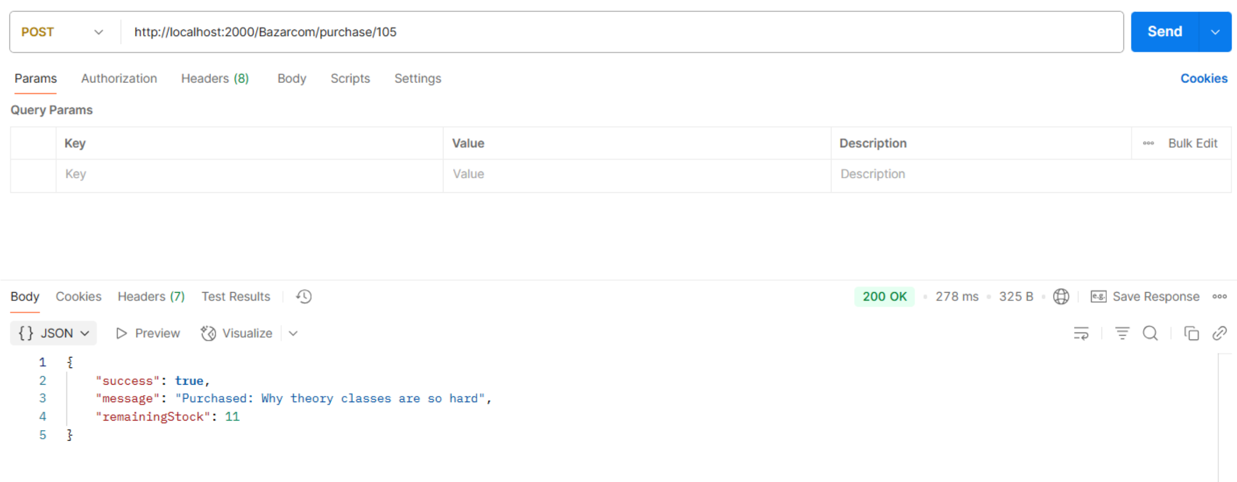
After the initial **cache miss**, the data for book ID 105 was fetched from **Catalog 1** and stored in the cache. On the second request, the frontend returned the result with a **cache hit**, reducing the response time to **13ms**.

```
frontend-1   | [Frontend] GET /Bazarcom/info/105
frontend-1   | [Frontend] Cache miss
catalog1-1   | [Catalog] GET /CatalogServer/query
frontend-1   | [Frontend] GET /Bazarcom/info/105
frontend-1   | [Frontend] Cache hit
```



A **purchase request** was made for book ID 105 ("Why theory classes are so hard"). The catalog service (Catalog 1) handled the stock update and synchronized the new quantity (11) with its replica (Catalog 2). The frontend then sent a **cache invalidation** request to ensure consistency. The order was processed by **Order 1**, synchronized with **Order 2**, and confirmed in the list of current orders. This demonstrates correct handling of write operations, replica synchronization, and cache invalidation.





Following the purchase and successful synchronization between the catalog and order replicas, the frontend invalidated the cached data for book ID 105. When the same book was queried again, the system returned a **cache miss**, demonstrating that the cache was properly cleared after the update. The frontend then forwarded the request to **Catalog 1** to fetch the updated information from the database.

```
frontend-1   | [Frontend] POST /Bazarcom/purchase/105
catalog1-1   | [Catalog] GET /CatalogServer/query
catalog1-1   | [Catalog] PUT /CatalogServer/updateStock/105
catalog1-1   | [Catalog] Stock updated for 105. New stock: 11
catalog2-1   | [Catalog] PUT /CatalogServer/syncStock/105
catalog2-1   | [Catalog] Synchronized stock for 105. New stock: 11
catalog1-1   | [Catalog] Synced stock with replica
frontend-1   | [Frontend] POST /invalidate
order1-1     | Purchased: Why theory classes are so hard
frontend-1   | [Frontend] Invalidated cache key: /Bazarcom/info/105-{"id":"105"}
order1-1     | Current orders: [
order1-1     |   {
order2-1     | [Order] Synchronized order from replica: {
order1-1     |     orderNumber: 1,
order2-1     |   orderNumber: 1,
order1-1     |     bookId: '105',
order2-1     |   bookId: '105',
order1-1     |     title: 'Why theory classes are so hard',
order2-1     |   title: 'Why theory classes are so hard',
order1-1     |     remaining_quantity: 11
order2-1     |   remaining_quantity: 11
order1-1     |   }
order2-1     | }
order1-1     | ]
order1-1     | [Order] Synced order with replica
frontend-1   | [Frontend] GET /Bazarcom/info/105
frontend-1   | [Frontend] Cache miss
catalog1-1   | [Catalog] GET /CatalogServer/query
```

The two queries for book IDs 103 and 102 resulted in **cache misses**, and due to the **round-robin load balancing**, the requests were distributed between **Catalog 2** and **Catalog 1** respectively.

```
frontend-1  | [Frontend] GET /Bazarcom/info/103
frontend-1  | [Frontend] Cache miss
catalog2-1  | [Catalog] GET /CatalogServer/query
frontend-1  | [Frontend] GET /Bazarcom/info/102
frontend-1  | [Frontend] Cache miss
catalog1-1  | [Catalog] GET /CatalogServer/query
```

Note: The rest of outputs are existed on our **GitHub** repository.

Comparison

This table compares the average response time for book query requests with and without caching. It demonstrates how caching significantly reduces latency by serving repeated requests from memory instead of querying the backend catalog server

Request Type	With Cache (ms)	Without Cache (ms)	Improvement (%)
GET /info/103	5	49	89.8%
GET /info/102	6	58	89.7%
Average	5.5	53.5	89.7%

This table shows the effect of a write operation (purchase) on the cache. It includes the time taken to invalidate the cache and the increased latency of the next query, which results in a **cache miss** and must fetch data from the backend again. This confirms that cache consistency is properly enforced after updates.

Step	Time (ms)	Notes
Query (cached)	4	Fast – returned from cache
Purchase operation	300	Includes DB write + replica sync
Query (after invalidate)	21	Cache miss, fetched from Catalog again

Conclusion

This lab successfully improved the performance and scalability of the Bazar.com system by implementing **caching**, **replication**, and **Docker-based deployment**. Caching significantly reduced the average response time for read requests by over **89%**, while replication allowed us to distribute traffic evenly using a **round-robin strategy**, enhancing load balancing and fault tolerance.

We also ensured **cache consistency** by invalidating cached data before database updates, with minimal overhead. Containerizing all services using **Docker** and managing them with **Docker Compose** made the system easier to deploy, test, and scale across different environments.