DOS Project

Students: Amr Badran & Mohammed Najeh

Instructor: Dr. Samer Arandi

Introduction

This project implements a distributed microservices system for a bookstore catalog service with replication, caching, and load balancing. The system consists of multiple services communicating via REST APIs, ensuring high availability and performance through replication and caching mechanisms.

System Architecture

This project builds a small online bookstore called Bazar.com. The store sells four books, but it is designed in a way that real websites work. The goal is to learn how to create a system made of different small services that talk to each other over a REST interface.

The system follows a microservices architecture with the following components:

- Gateway Service (Front-end): Node.js service that acts as the entry point
- Catalog Service: .NET Core service managing book inventory (2 replicas)
- Order Service: PHP Lumen service handling purchase operations (2 replicas)

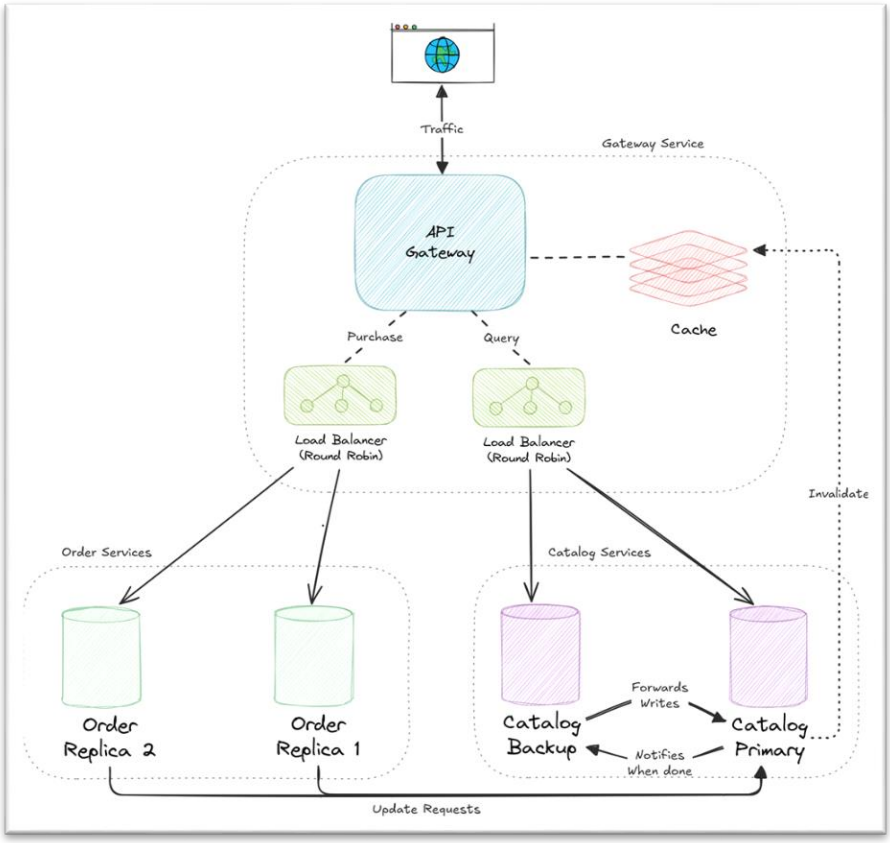
Technology Stack

Services and tools used in the implementation:

- Front-end Gateway: Node.js with Express.js
- Catalog Service: ASP.NET Minimal Web API (.NET 9.0, C#)
- Order Service: PHP 8.2 with Lumen framework
- Containerization: Docker and Docker Compose
- Databases: SQLite for each service replica

Service Communication

All services communicate using HTTP REST APIs, maintaining compatibility with standard web protocols. The gateway routes requests to appropriate backend services using round-robin load balancing.



Replication

Replication Strategy

Each service (Catalog and Order) runs two replicas to ensure high availability and fault tolerance. Each replica maintains its own independent database copy.

When a write operation (create, update, delete) occurs:

- Primary Replica: Receives the write request and processes it locally.
- Replication: The primary replica automatically propagates the write to all other replicas.
- Consistency: All replicas maintain synchronized data.

Implementation Details

Catalog Service (C#):

- Uses ReplicationService to propagate writes.
- Sends HTTP PATCH requests to other replicas with X-Replication header.

Order Service (PHP):

- Uses ReplicationService class for write propagation.
- Sends POST requests to /purchase/replicate endpoint on other replicas.

Key Features:

- Automatic synchronization across replicas.
- Prevention of replication loops using headers.
- Asynchronous replication (non-blocking).

Load Balancing

Round-Robin Algorithm

The gateway implements round-robin load balancing to distribute incoming requests evenly across service replicas.

How It Works

```
Request 1 → Replica 1
Request 2 → Replica 2
Request 3 → Replica 1 (wraps around)
Request 4 → Replica 2
```

This ensures:

- Equal distribution of load.
- Better performance through parallel processing.
- High availability (if one replica fails, requests go to others).

Implementation

- Gateway maintains index counters for each service type.
- Uses modulo arithmetic to cycle through replicas.
- Applies to both read and write operations.

Caching

In-Memory Cache

The gateway implements an in-memory cache to improve read performance for frequently accessed data.

Cached Endpoints

Only read requests are cached:

- GET /books/search/{topic} - Search books by topic
- GET /books/info/{id} - Get book information by ID

Write requests are NOT cached:

- PATCH /books/cost/{id} - Update book cost
- PATCH /books/stock/{id} - Update book stock
- POST /purchase/{id} - Purchase a book

Cache Data Structure

Each cache entry stores only essential book information:

- Book ID
- Title
- Price
- Quantity (stock)

This minimizes memory usage while maintaining necessary data.

Cache Consistency - Server-Push Invalidation

To ensure strong consistency, the system uses server-push invalidation:

- Before write operation: Backend services send invalidation request to gateway.
- Cache invalidation: Gateway removes affected cache entries.
- Write execution: Write operation proceeds.
- Result: Subsequent reads fetch fresh data from backend.

Performance Measurements

Test Methodology

Performance tests measure response times for different scenarios such as searching for books, retrieving book information, and executing write operations (update and purchase).

Test Results

Scenario	Count	Average (ms)	Min (ms)	Max (ms)	Successes	Failures
Search Books by Topic - distributed systems	6	67.68	1.97	379.28	6	0
Search Books by Topic - undergraduate school	6	57.19	2.14	330.81	6	0
Get Book Info - ID 1	6	15.63	1.64	77.86	6	0
Get Book Info - ID 2	6	23.50	1.50	131.60	6	0
Update Book Cost - ID 1	3	155.33	16.81	361.07	3	0
Update Book Stock - Increase ID 1	3	54.38	21.76	84.98	3	0
Purchase Book - ID 1	3	127.34	13.76	238.21	0	3

Across the scenarios measured, the overall average response time was 71.58 ms, with observed per-request minimum of 1.50 ms and maximum of 379.28 ms.

Conclusion

The measured results show that read operations (search and book info) generally have lower average response times than write operations (cost/stock updates and purchase). Cache usage in the gateway helps reduce repeated read request costs, while writes include cache invalidation and replication steps.

API Endpoints

Read Operations (Cached)

```
GET /books/search/{topic}      - Search books by topic
GET /books/info/{id}           - Get book information
```

Write Operations

```
PATCH /books/cost/{id}         - Update book cost
PATCH /books/stock/{id}        - Update book stock (increase/decrease)
POST /purchase/{id}            - Purchase a book
```

Internal Endpoints

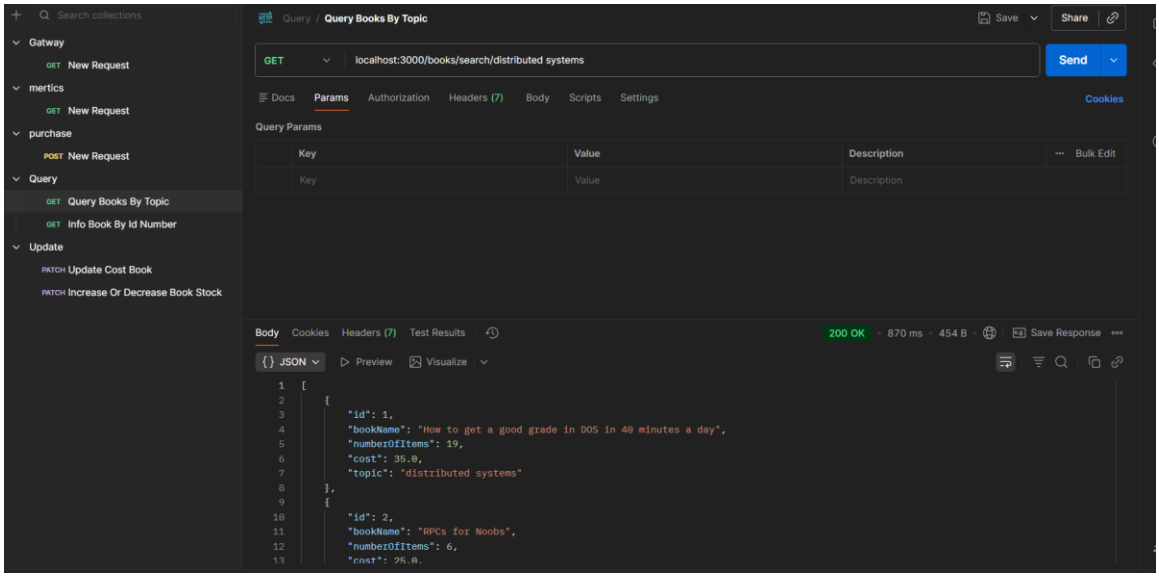
```
POST /cache/invalidate         - Invalidate cache (called by backend)
POST /purchase/replicate       - Replicate order (called by replicas)
```

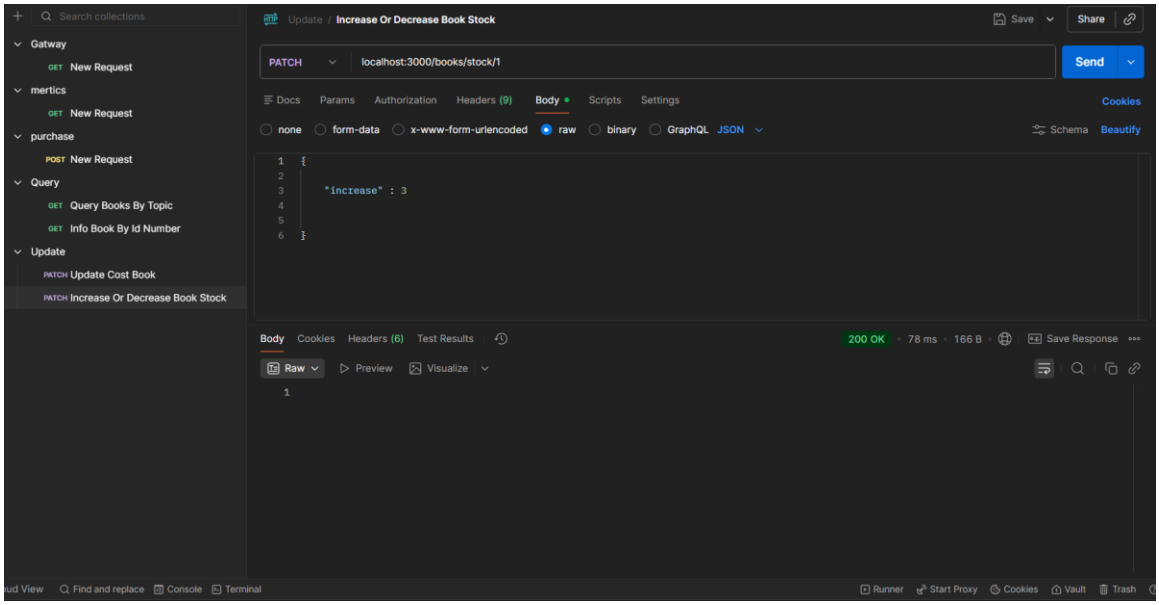
Direct Service Endpoints (as used in testing)

Catalog Service:

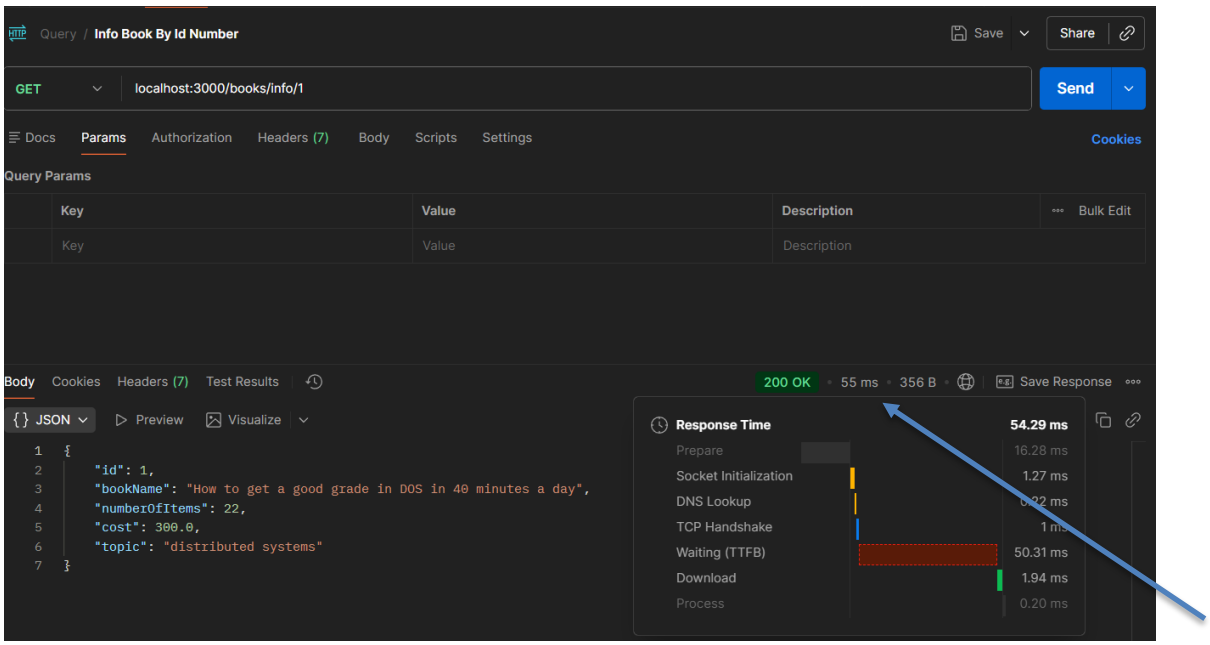
```
GET   CATALOG_IP:PORT/books/search/{topicName}
GET   CATALOG_IP:PORT/books/info/{bookNumber}
PATCH CATALOG_IP:PORT/books/cost/{bookNumber}
PATCH CATALOG_IP:PORT/books/stock/{bookNumber}
```

1) GET: localhost:3000/books/search/distributed systems

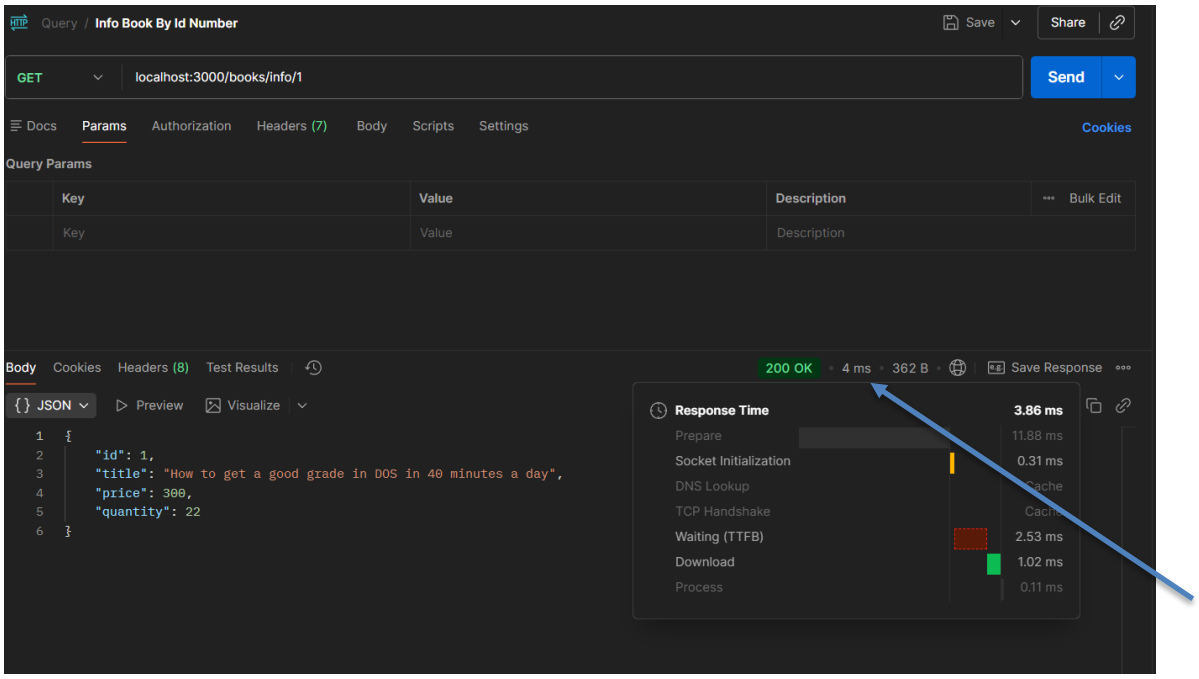




2- **before caching** (cache miss)



3- **after caching** (cache hit)



Order Service:

`POST ORDER_IP:PORT/purchase/{bookNumber}`

Gateway Service:

All external requests can be sent through the gateway on port 3000. The gateway forwards requests to the Catalog and Order services.

`Gateway: http://localhost:3000`

Deployment

Docker Compose

All services are containerized using Docker:

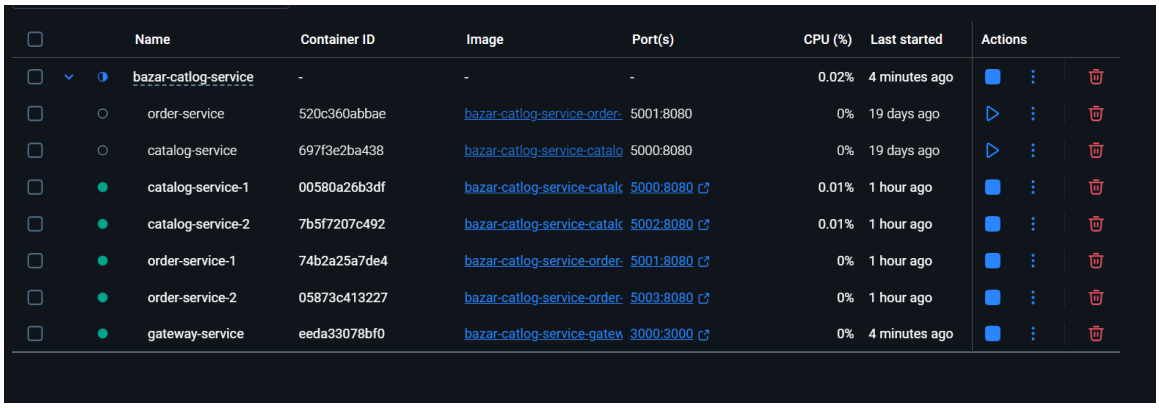
- Gateway: Port 3000
- Catalog Replica 1: Port 5000
- Catalog Replica 2: Port 5002

- Order Replica 1: Port 5001
- Order Replica 2: Port 5003

Running the System

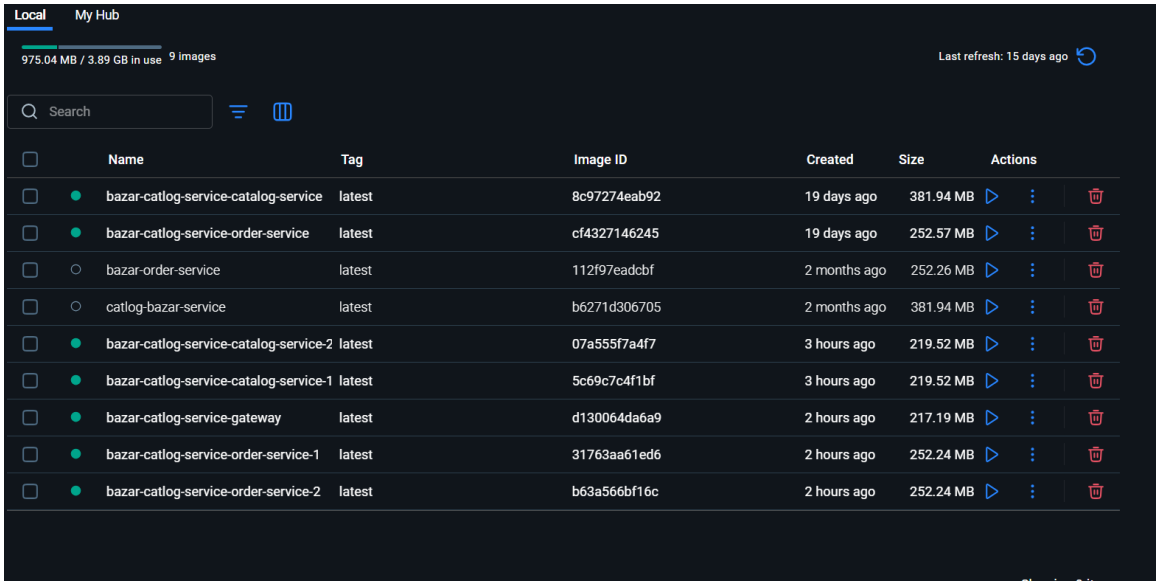
```
docker-compose up --build
```

1-Docker container



Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
bazar-catlog-service	-	-	-	0.02%	4 minutes ago	
order-service	520c360abbae	bazar-catlog-service-order-	5001:8080	0%	19 days ago	
catalog-service	697f3e2ba438	bazar-catlog-service-catalo	5000:8080	0%	19 days ago	
catalog-service-1	00580a26b3df	bazar-catlog-service-catal	5000:8080	0.01%	1 hour ago	
catalog-service-2	7b5f7207c492	bazar-catlog-service-catal	5002:8080	0.01%	1 hour ago	
order-service-1	74b2a25a7de4	bazar-catlog-service-order-	5001:8080	0%	1 hour ago	
order-service-2	05873c413227	bazar-catlog-service-order-	5003:8080	0%	1 hour ago	
gateway-service	eeda33078bf0	bazar-catlog-service-gatew	3000:3000	0%	4 minutes ago	

2-Docker images



Local | My Hub

975.04 MB / 3.89 GB in use | 9 images | Last refresh: 15 days ago

Search

Name	Tag	Image ID	Created	Size	Actions
bazar-catlog-service-catalog-service	latest	8c97274eab92	19 days ago	381.94 MB	
bazar-catlog-service-order-service	latest	cf4327146245	19 days ago	252.57 MB	
bazar-order-service	latest	112f97eadcbf	2 months ago	252.26 MB	
catlog-bazar-service	latest	b6271d306705	2 months ago	381.94 MB	
bazar-catlog-service-catalog-service-2	latest	07a555f7a4f7	3 hours ago	219.52 MB	
bazar-catlog-service-catalog-service-1	latest	5c69c7c4f1bf	3 hours ago	219.52 MB	
bazar-catlog-service-gateway	latest	d130064da6a9	2 hours ago	217.19 MB	
bazar-catlog-service-order-service-1	latest	31763aa61ed6	2 hours ago	252.24 MB	
bazar-catlog-service-order-service-2	latest	b63a566bf16c	2 hours ago	252.24 MB	

Conclusions

This project successfully implemented a two-replica microservices architecture with automatic write propagation, in-memory caching for read endpoints with server-push invalidation.

for load balancing we usesd round-robin to evenly distribute requests, achieving strong consistency through cache invalidation before writes and replicated updates. Key learnings include the flexibility and scalability offered by microservices, the availability benefits and consistency challenges of replication.

the performance gains from caching when paired with correct invalidation strategies, and the effectiveness of load balancing across multiple instances. Future improvements include adding more replicas to increase availability, implementing cache expiration policies.