An-Najah National University

Faculty of Engineering & Information Technology

Dos-Project - Part 2

Abdulqader mohammad – 12112218

Mohammad raddad – 12027855

1. Introduction

In this lab, we extend the Bazar.com online bookstore developed in Lab 1 to handle higher workloads and improve performance. The main objective is to apply key distributed systems concepts including replication, caching, and consistency using a microservices-based architecture and RESTful APIs.

2. System Architecture:

The system consists of the following components:

- **Frontend Server**
  - Receives all client requests
  - Implements load balancing using Round Robin
  - Maintains an in-memory cache with LRU eviction
- **Catalog Service**
  - Stores book information (title, price, quantity)
  - Implemented with SQLite
  - Replicated into two instances running on different ports
- **Order Service**
  - Handles purchase requests
  - Replicated into two instances
  - Updates all catalog replicas to maintain consistency

3. Replication

Replication is implemented for both the Catalog Service and the Order Service.

The frontend server distributes incoming requests among replicas using a Round Robin load-balancing strategy.

For write operations (purchases), the Order Service propagates updates to all catalog replicas, ensuring that all copies remain synchronized.

4. Caching

An in-memory cache is implemented inside the frontend server to store responses for read-only requests (/info/:id).

```
// cached read request
app.get("/info/:id", async (req, res) => {
  const id = req.params.id;
  if (cache.has(id)) {
    console.log("CACHE HIT for book", id);
    return res.json(cache.get(id)); // cache hit
  }
    console.log("CACHE MISS for book", id);
  const catalog = nextCatalog();
  const r = await axios.get(`${catalog}/info/${id}`);
  updateCache(id, r.data);

  res.json(r.data);
});
```

5. Load balancing

is implemented to distribute incoming requests across multiple backend service replicas. The Round Robin algorithm was chosen due to its simplicity and minimal overhead. For each incoming search request, the frontend forwards the request to the next available replica in a cyclic order, ensuring that consecutive requests are handled by different replicas. This strategy evenly distributes the workload, improves system performance, and prevents any single replica from becoming a bottleneck.

```
const CATALOG_REPLICAS = [
  "http://localhost:3001",
  "http://localhost:3003"
];


const ORDER_REPLICAS = [
  "http://localhost:3002",
  "http://localhost:3004"
];


let catalogIndex = 0;
let orderIndex = 0;

// select next Catalog replica
function nextCatalog() {
  const url = CATALOG_REPLICAS[catalogIndex];
  catalogIndex = (catalogIndex + 1) % CATALOG_REPLICAS.length;
  return url;
}

// select next Order replica
function nextOrder() {
  const url = ORDER_REPLICAS[orderIndex];
  orderIndex = (orderIndex + 1) % ORDER_REPLICAS.length;
  return url;
}
```

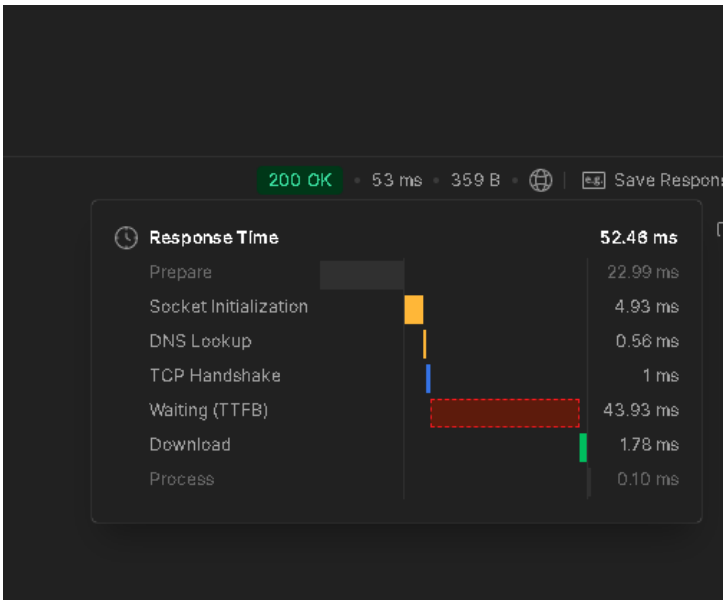
6. Experimental Evaluation

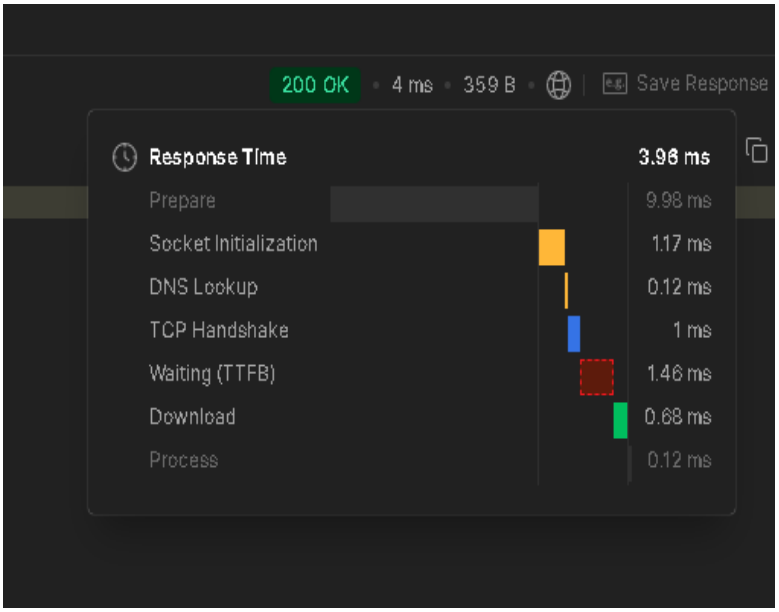
6.1 Response Time Measurement

We measured the average response time for /info/:id requests in two scenarios:

We measured the average response time by issuing 50 consecutive /info/:id requests using a Node.js script. The experiment was performed with caching disabled and enabled.

Scenario	Average Response Time
Without Cache	52.46 ms
With Cache	4 ms





6.2 Cache Invalidation Experiment

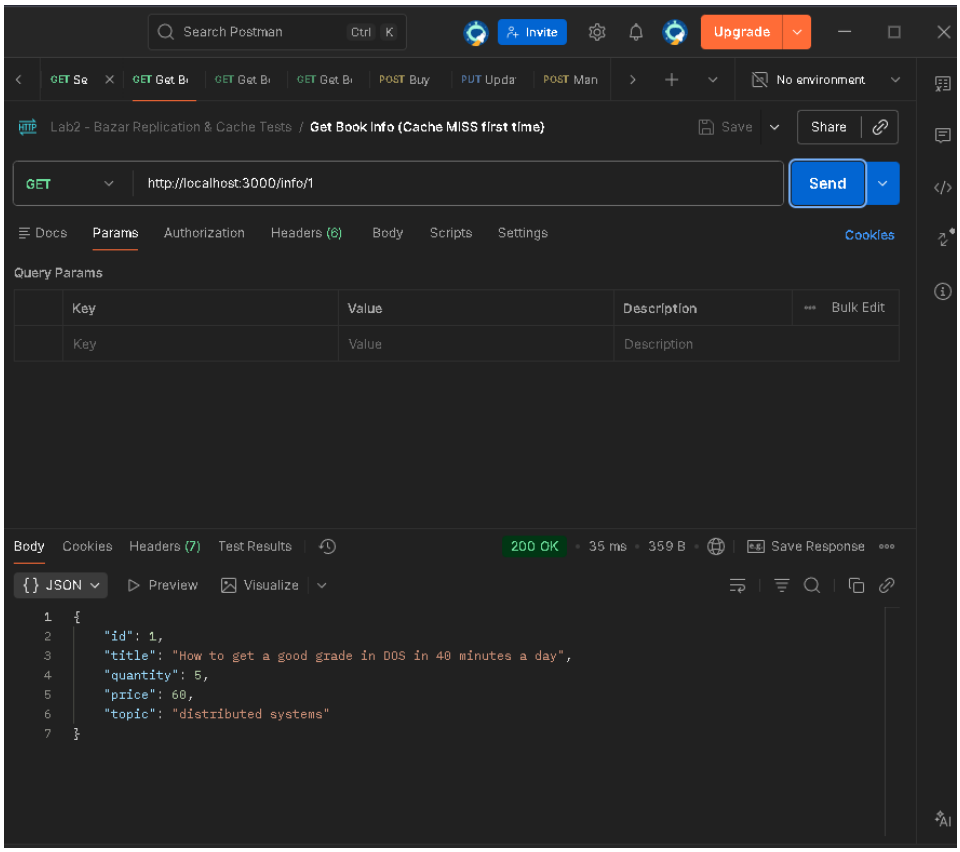
The following experiment was conducted:

1. **Request /info/:id → Cache HIT**
2. **Execute /purchase/:id**
3. **Cache invalidation occurs**
4. **Request /info/:id again → Cache MISS**

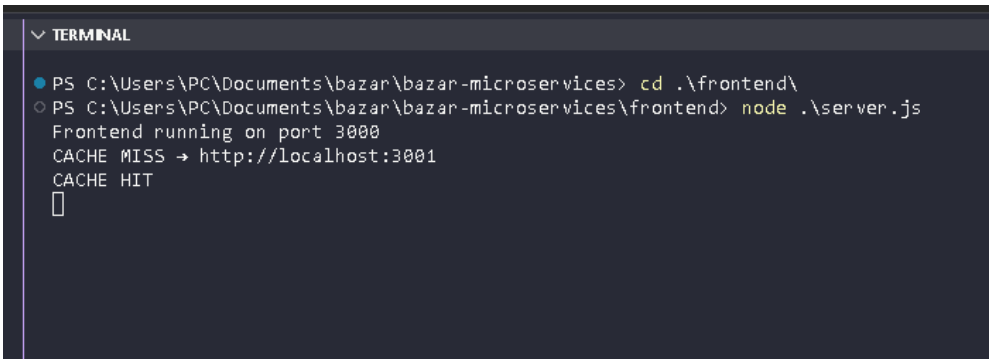
Operation	Time (ms)
Cache Invalidation	80
Request after invalidation (MISS)	7

```
// cache invalidation after write operation
app.post("/invalidate/:id", (req, res) => {
  const id = req.params.id;

  if (cache.has(id)) {
    console.log("CACHE INVALIDATED for book", id);
    cache.delete(id);
  } else {
    console.log("INVALIDATE received, but book not in cache", id);
  }

  res.json({ success: true });
});

// start Frontend service
```

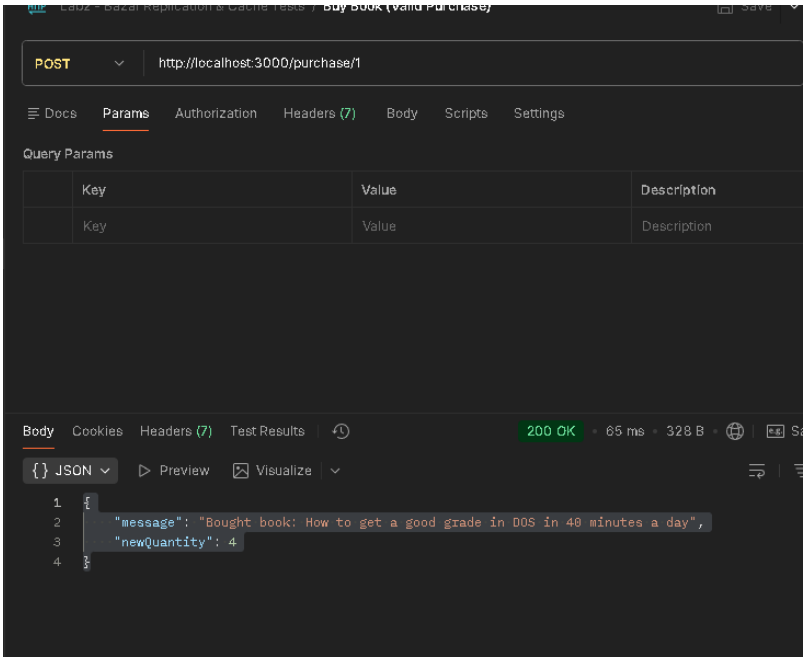
- **first request**



- **second request** **GET http://localhost:3000/info/1**



- **third request**



- **<u>Fourth request</u>** **<u>GET http://localhost:3000/info/1</u>**

7. Design Tradeoffs

- **Replication improves availability and scalability**
- **Caching reduces latency but introduces consistency complexity**
- **Strong consistency is achieved at the cost of additional invalidation overhead**
- **SQLite was chosen for simplicity but limits scalability**

8. Possible Improvements

- **Add health checks for replicas**
- **Dockerize all services**
- **Replace SQLite with a distributed database**

9. How to Run the System

Start Catalog replicas:

- **node server.js**
- **$env:PORT=3003; node server.js**

Start Order replicas:

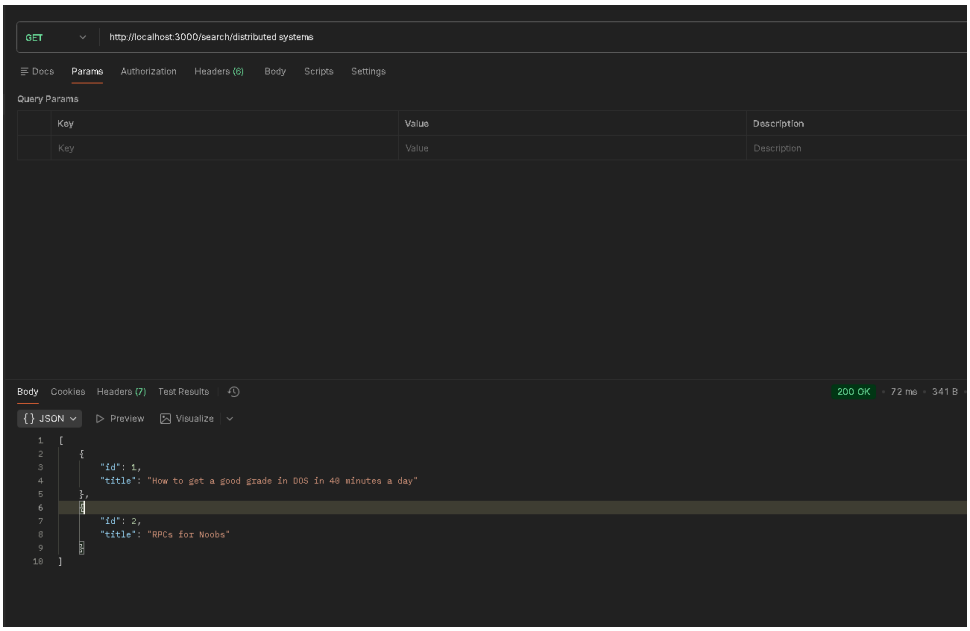
- **node server.js**
- **$env:PORT=3004; node server.js**

Start Frontend server:

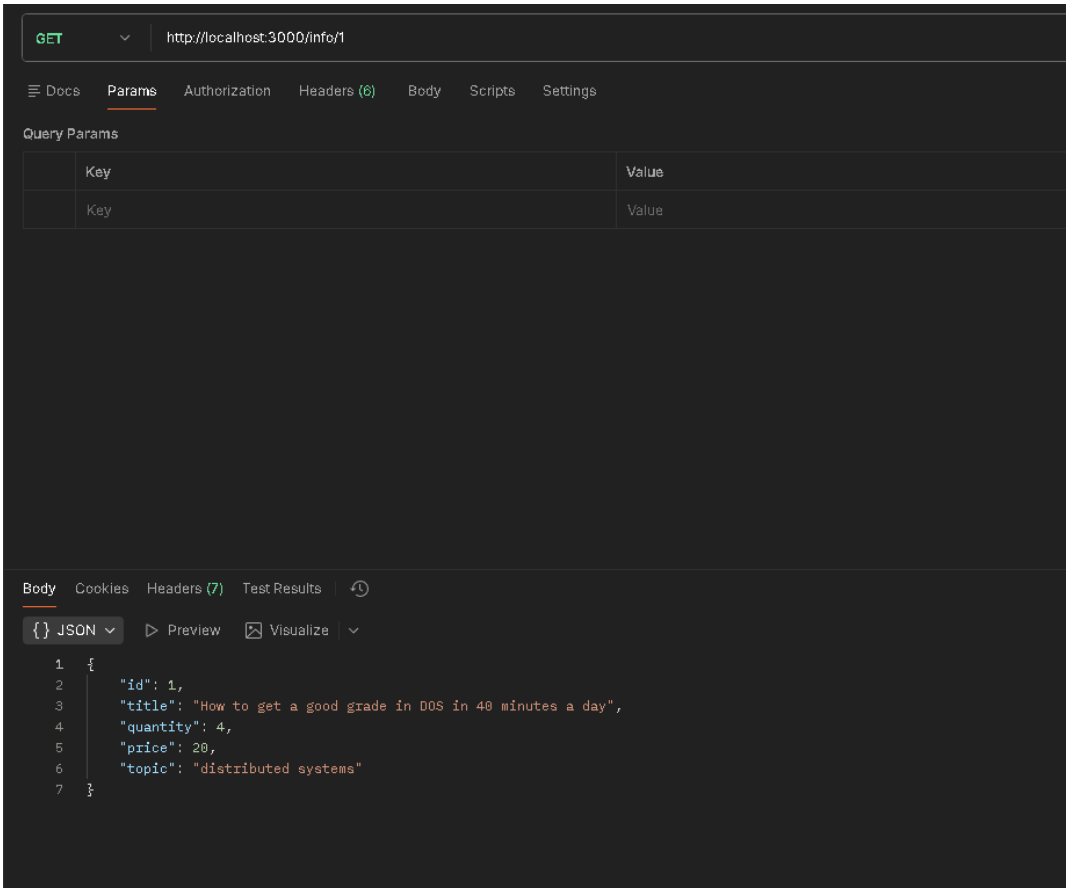
- **node server.js**

10.result :

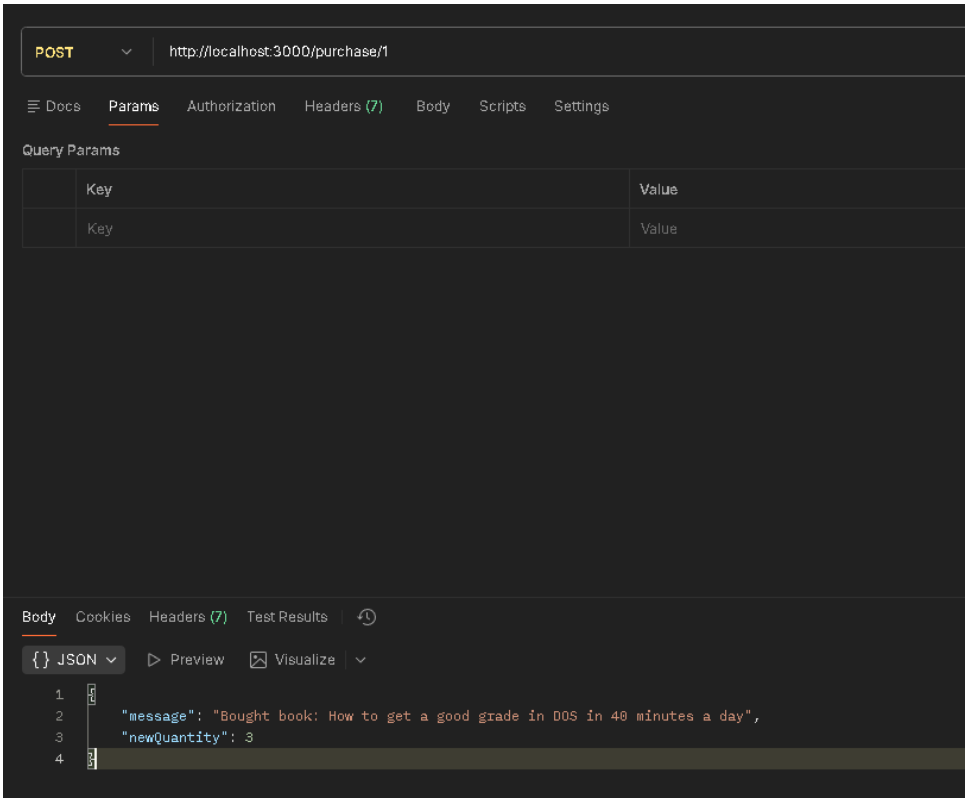
- **http://localhost:3000/search/distributed systems**



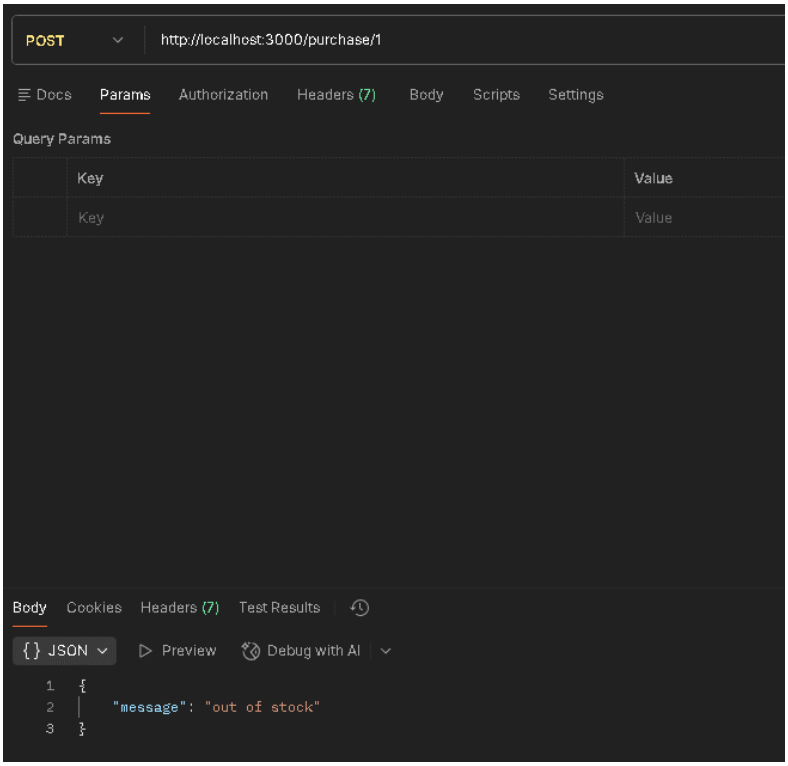
- **http://localhost:3000/info/1**



- **http://localhost:3000/purchase/1**



- **http://localhost:3000/purchase/1 when the out of stock**



11. Conclusion

This lab demonstrates how replication, caching, and consistency mechanisms can be combined to build a scalable and efficient distributed system. The experimental results confirm improved performance while maintaining correct system behavior.