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

Raya Hindi 12027720 Masa Lubbadeh 12027847

Contents

Introduction

This project aims to enhance the performance of Bazar.com, a simulated online bookstore. As Bazar.com became more popular, users will complain about slow request processing, by introducing replication, caching, and consistency mechanisms. These techniques are commonly used to improve the scalability.

Objectives

The main goals of this project are:

1. **Learn replication, caching, and consistency**: These techniques are very important for handling large-scale web applications with traffic.
2. **Learn to use Docker**

Project overview

Bazar.com consists of:

- **Front-end** server: Handles user requests and perform initial processing, also it coordinates the communication between the user interface with the backend.
- **Catalog Server**: Manages book inventory with stock counts, pricing, and topic categorization.
- **Order Server**: Processes purchase orders and manages inventory updates.
- **Catalog server & order server replicas:** as we mentioned before, we need them to handle more requests.

Implementation

As mentioned in Part 1, Bazar.com is built using **Node.js** and **Express.js**. Node.js was chosen for its non-blocking, event-driven architecture, which is ideal for building scalable

applications that need to handle numerous simultaneous connections, such as an online bookstore. Additionally, **Express.js**, a minimal and flexible web application framework for Node.js, was used to streamline the development process, offering powerful tools for creating RESTful APIs.

In the initial setup, the application consisted of separate components like the **Catalog Server** and the **Order Server**, which handled book-related queries and purchase transactions. These components communicated with the frontend through **Axios**, a promise-based HTTP client for making asynchronous requests between the frontend and backend services.

New Changes and Enhancements:

In Part 2, we introduced several optimizations to handle higher traffic and improve performance:

1. **Replication:** Both the Catalog and Order services are now replicated, which means multiple instances of each service run to handle incoming requests. The frontend server is responsible for distributing incoming requests across these replicas using a load-balancing algorithm. we implemented a **round-robin** strategy, where each incoming request is routed to the next available replica in a cyclic manner.

2. **Caching:** To improve response times, we added an in-memory cache on the frontend server to store frequently accessed data, such as book details. This helps reduce redundant queries to the Catalog Service. The cache functions as a simple key-value store. Before forwarding requests to the backend, the cache is checked to see if the data is already available. Whenever the data is updated, the cache is cleared to ensure the information remains current.

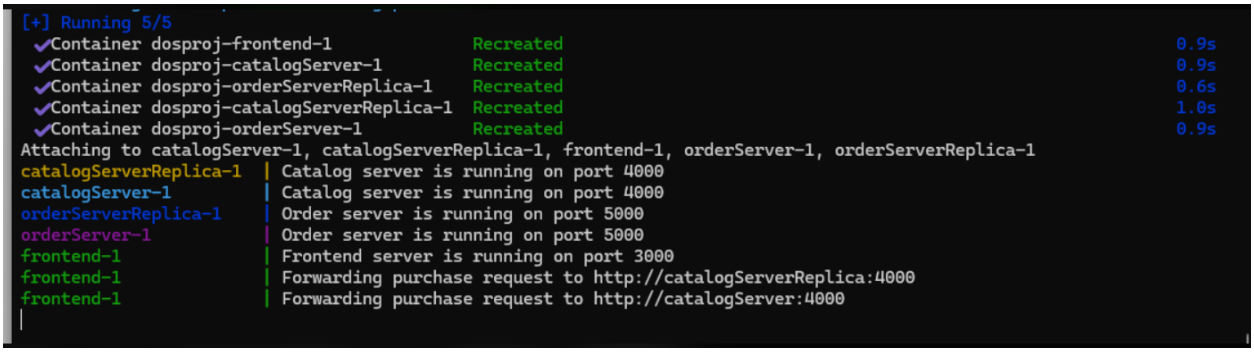
3. **Consistency:** To ensure consistency across replicas, the system synchronizes updates through a server-push mechanism. When a purchase is made, the replicas are notified to update the quantity of the book to keep them in sync.

Setup and Installation

"Using Docker"

1. Clone the repository to your local machine:
   bash git clone **https://github.com/MasaLubbadeh/DOSProj.git** cd DOSProj
2. Build the Docker images and start the containers:

   "bash docker-compose up –build"

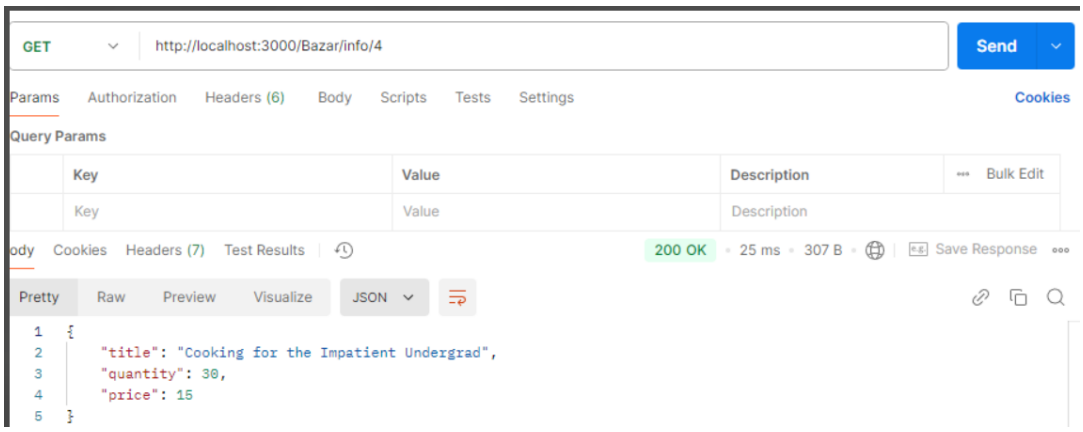


The system should now be running on Docker containers.

Testing APIs

Get book's information by id:

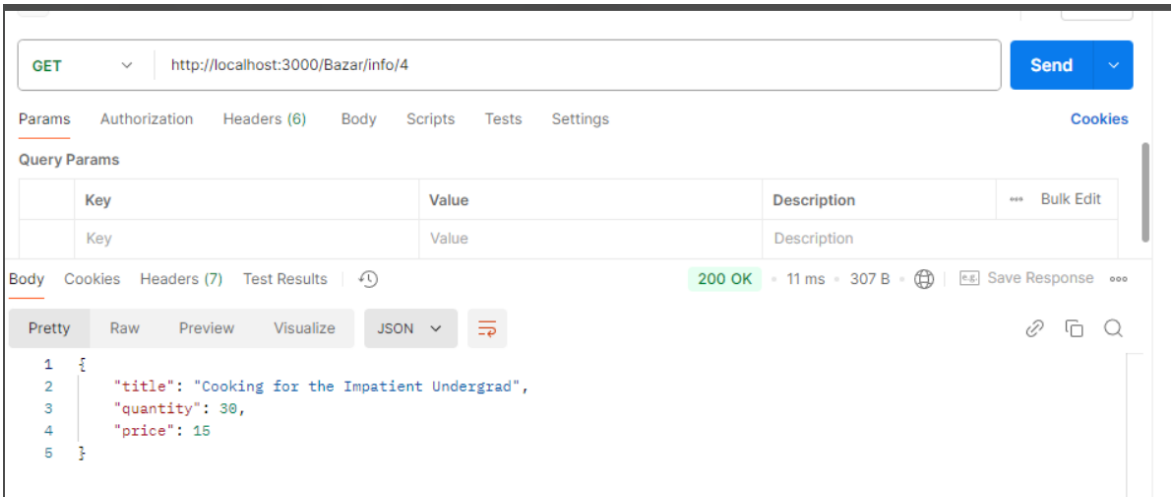
http://localhost:3000/Bazar/info/:id



Frontend server receives the request, first it checks the book is already in cache so the result will be taken from cache. Else, the request is forwarded to the catalog server (the chosen replica), then add it to cache.

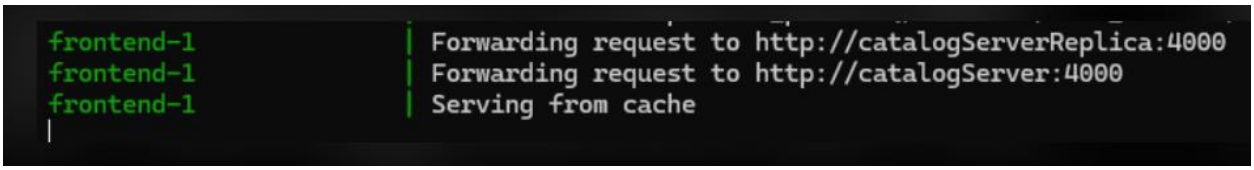
In the case above, book wasn't in cache. Time:25ms

Now when we perform the request again, it will be in cache.



We can see the time:11ms which is less and better that the case before.

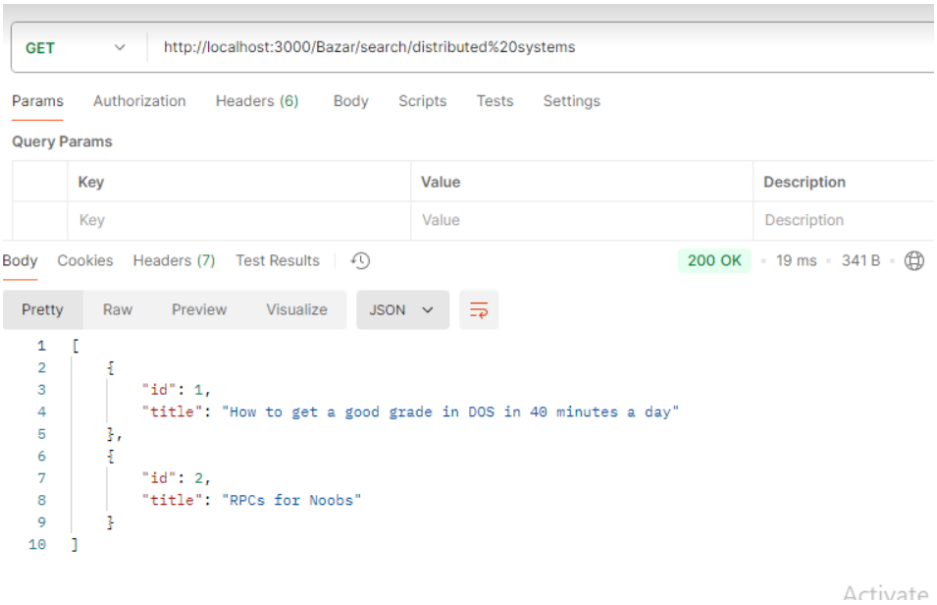
We can see the first request is given to catalogServerReplica and the second to catalogServer.



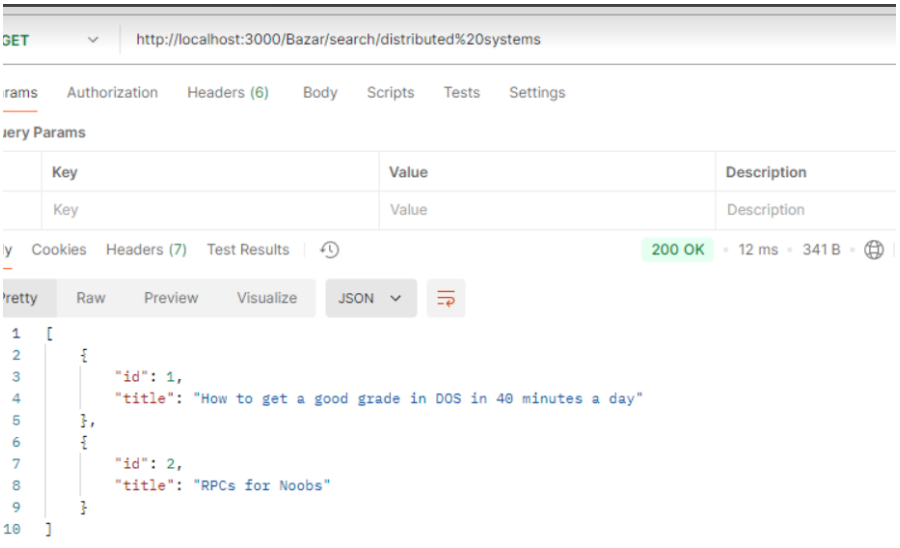
Search books by topic:

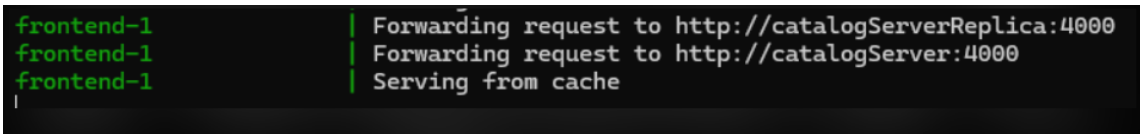
http://localhost:3000/Bazar/Search/:topic

the same process as before. (chosen replica forward request to catalog server after checking the cache). In this case the books are not yet in cache.



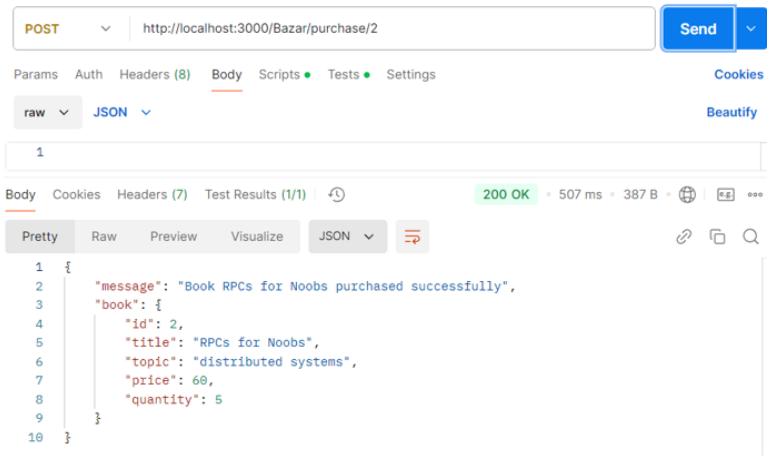
Now when the books are already in cache, there's no need to forward it to catalog server, so time is less than the previous case. (time:14)





Purchase books by id:

http://localhost:3000/Bazar/purchase/:id



Frontend server will forward the request to order server replica, the chosen order server replica will forward the request to the corresponded Catalog Server. If the book is updated successfully.

- the server will notify the in-memory cache with an invalidate request if the item in cache, which causes book's data to be removed from the cache. It the book is not in cache then no invalidation is needed.

- this replica will notify the other replica that the stock is changed for this item. (forwarding a request to update book's quantity)

Replica setup

Replicas were created by using Docker's multi-container setup, with the server and its replica running on different ports:

- The **Catalog Server** runs on port 4000, while the **Catalog Server Replica** runs on port 4001. Both instances are built from the same Docker image, ensuring consistency between them.
- Similarly, the **Order Server** operates on port 5000, and the **Order Server Replica** runs on port 5001.

The Docker containers are managed using **Docker Compose.**

conclusion

In this project, we successfully implemented several techniques aimed at improving the performance and scalability of Bazar.com, transforming it into a more efficient and responsive online bookstore system.

The use of Docker containers developed an easy deployment process whereby each service can be deployed and managed independently.

Future Work:

- **Cache Eviction Strategies**: Implementing a more sophisticated cache eviction policy, such as Least Recently Used (LRU).
- **Horizontal Scaling**: allowing for more replicas to handle traffic.