Distributed Operating Systems

Bazar.com: A Multi-tier Online Book Store

Lama Ibrahim Hala Jabi

➔ Introduction:

This project involved designing and implementing **Bazar.com**, a minimalistic online bookstore with a two-tier microservices architecture. The store, offering four book titles across *Distributed Systems and Undergraduate Life categories*, features a front-end for user requests and a back-end with two microservices: a catalog server for inventory management and an order server for processing purchases. Built with a RESTful interface, the system emphasizes modularity, scalability, and lightweight distributed processing.

➔ System Architecture and Design

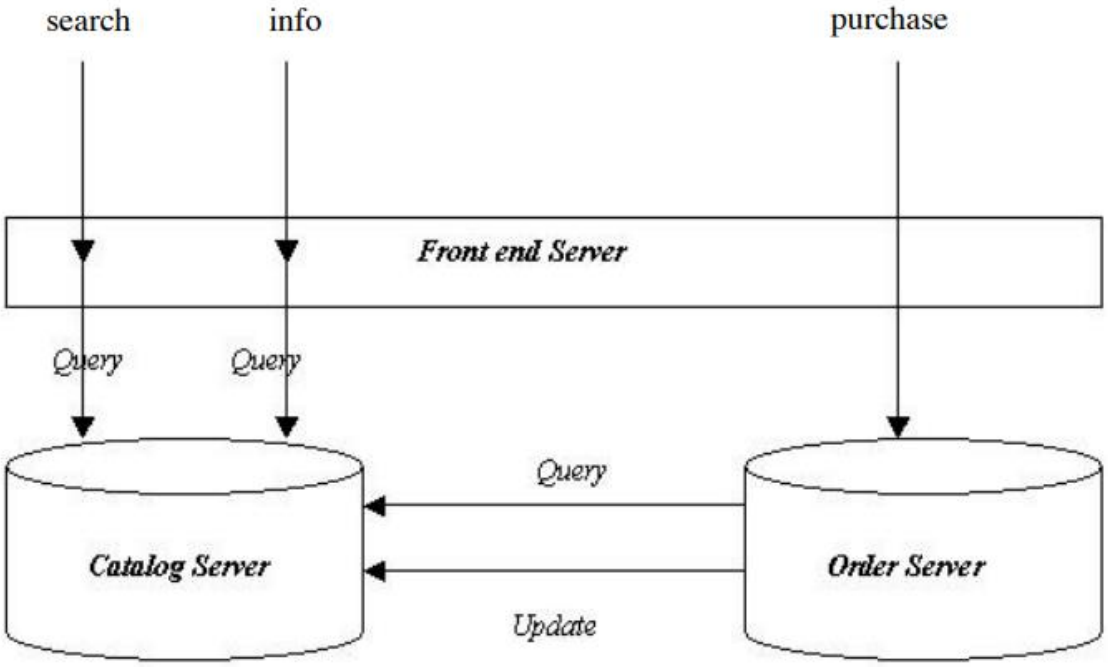
1. Project Goals and Requirements

The primary goals for this project included:

- Building a distributed online bookstore that functions with minimal hardware resources and microservices.
- Ensuring ease of use through a REST API that supports basic operations: search, info, and purchase.
- Developing a lightweight, scalable system suitable for future expansion.

2. System Components:

- **Front-End Microservice**: The front-end service is responsible for handling incoming user requests, initiating API calls to the catalog and order services as needed. It manages three core operations:
  - **search(topic)**: Accepts a topic (either Distributed Systems or Undergraduate Life) and returns book titles within that category.
  - **info(item_number)**: Retrieves details about a specific book, such as stock level and price.
  - **purchase(item_number)**: Initiates a purchase request for a given book, handled by the order server.



- **Catalog Server**: Manages the book inventory, including stock levels and prices. It supports two query operations (by subject or item) and an update operation for stock management.

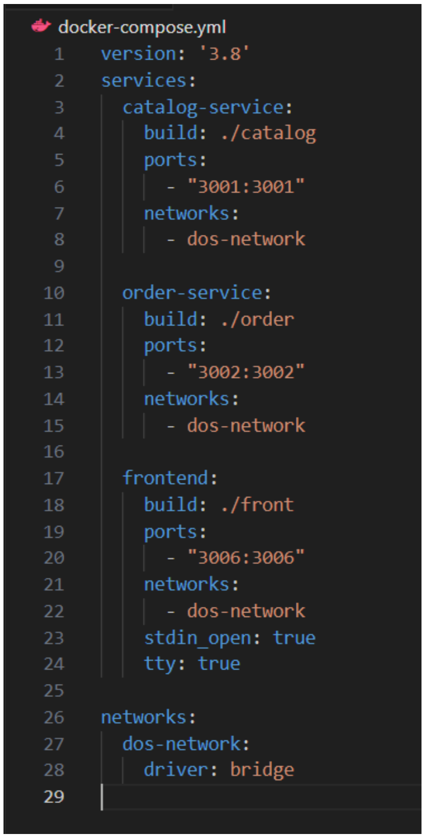
- **Order Server**: Processes purchase requests by verifying stock with the catalog server. If stock is available, it updates the inventory; otherwise, it notifies the user of unavailability.

3. RESTful API and Data Flow

A RESTful API connects each component. For instance, a ***GET*** request to server IP : http://localhost:3001/search/distributed%20systems

retrieves all books under Distributed Systems, formatted as JSON for easy client handling. Similarly, *POST* requests to the order server initiate purchases, ensuring asynchronous, stateless communication between services.

➔ Dockerization:



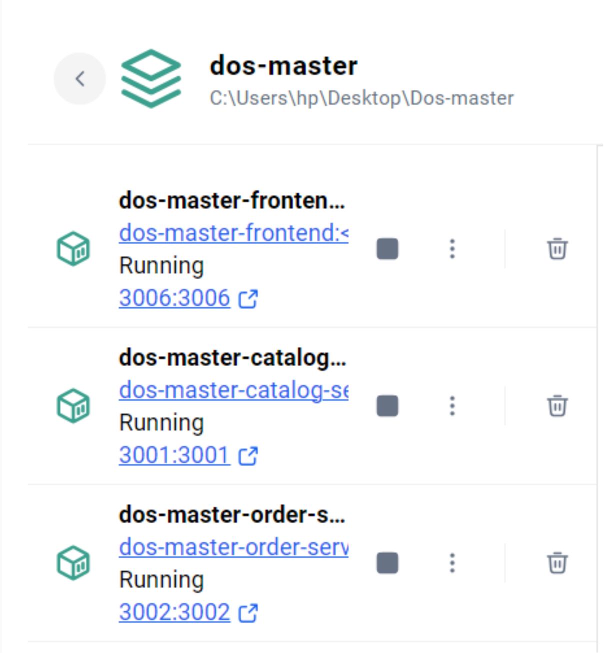
docker-compose.yml

```
version: '3.8'
services:
  catalog-service:
    build: ./catalog
    ports:
      - "3001:3001"
    networks:
      - dos-network

  order-service:
    build: ./order
    ports:
      - "3002:3002"
    networks:
      - dos-network

  frontend:
    build: ./front
    ports:
      - "3006:3006"
    networks:
      - dos-network
    stdin_open: true
    tty: true

networks:
  dos-network:
    driver: bridge

```

The **docker-compose.yml file** serves as the orchestrator for this multi-service application, unifying the individual microservices **catalog**, **order**, and **front** into a coordinated, easily manageable system.

Using Docker Compose, the file defines each service's unique build context and configuration, mapping external ports to internal container ports, ensuring that they're accessible while maintaining isolated environments for each service.

For instance, catalog and order services are assigned ports **3001** and **3002**, respectively, to facilitate internal API requests while keeping communication between services seamless and private within the custom network, dos-network. This network, defined as a bridge, acts as an internal highway, allowing services to interact securely without exposing unnecessary routes to the outside world.



dos-master
C:\Users\hp\Desktop\Dos-master

dos-master-fronten...
dos-master-frontend:<
Running
3006:3006

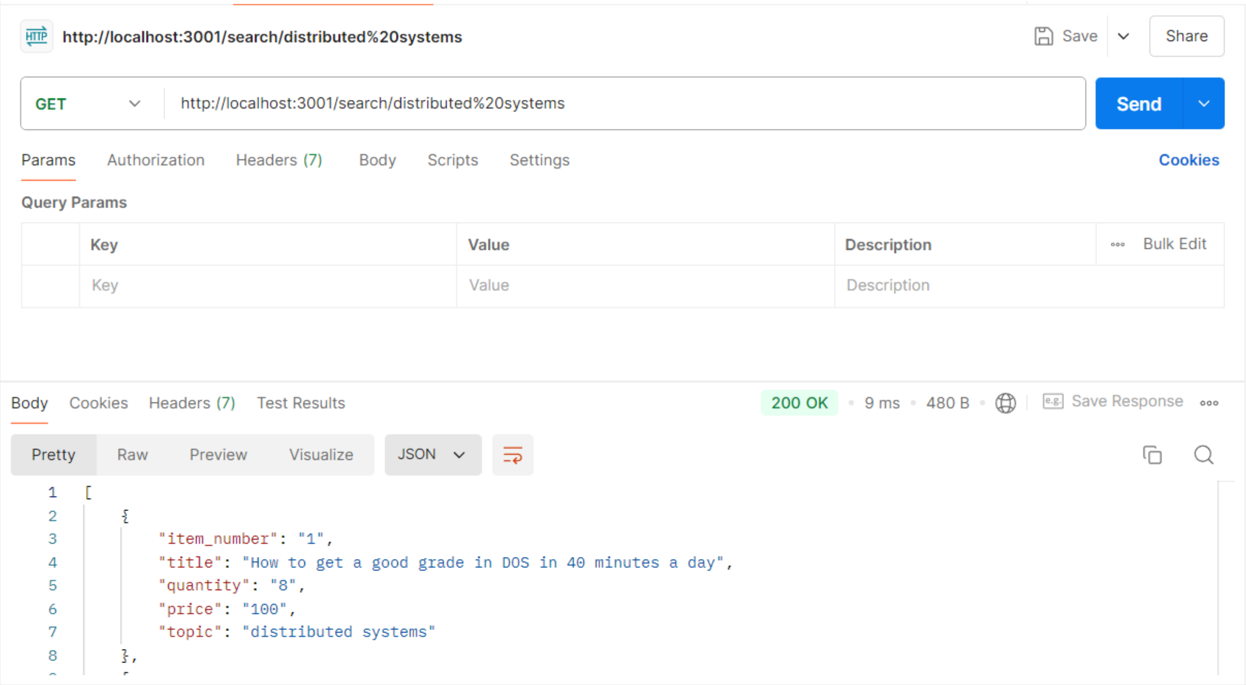
dos-master-catalog...
dos-master-catalog-se
Running
3001:3001

dos-master-order-s...
dos-master-order-serv
Running
3002:3002

The front service, which operates as a command-line interface for user interactions, has additional configurations with **stdin_open** and **tty** settings. These options keep the CLI open and interactive, making it ready for user commands upon startup. The structured and modular layout of this **docker-compose.yml** file not only streamlines service deployment but also simplifies scaling and debugging, encapsulating each service in an isolated, manageable container that works in harmony with others in the microservices ecosystem.

➔ Testing (using Postman)

1) Find by topic:



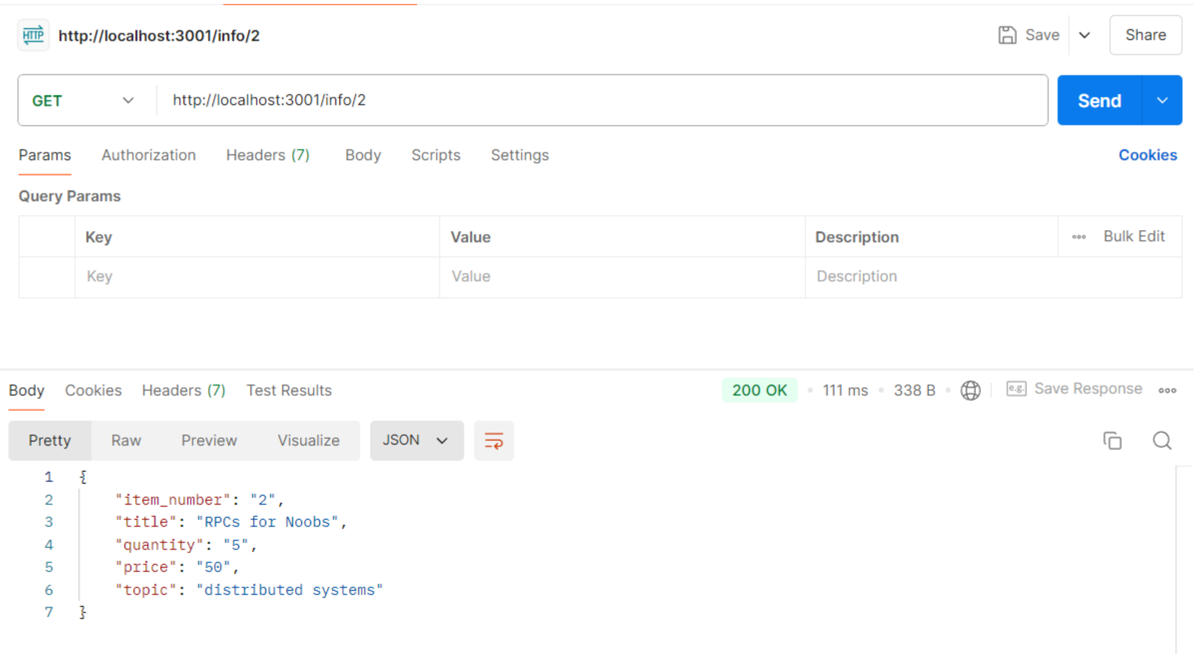
http://localhost:3001/search/distributed%20systems

GET http://localhost:3001/search/distributed%20systems

200 OK • 9 ms • 480 B

```
[
    {
        "item_number": "1",
        "title": "How to get a good grade in DOS in 40 minutes a day",
        "quantity": "8",
        "price": "100",
        "topic": "distributed systems"
    },
```

2) Find by item_number:



http://localhost:3001/info/2

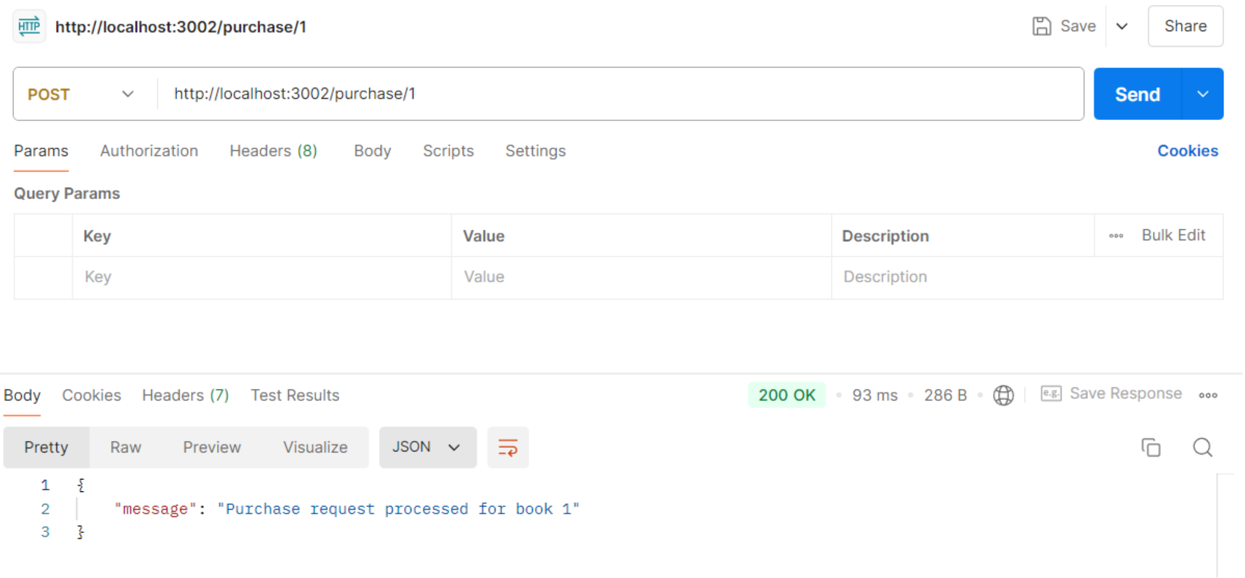
GET http://localhost:3001/info/2

200 OK • 111 ms • 338 B

```
{
    "item_number": "2",
    "title": "RPCs for Noobs",
    "quantity": "5",
    "price": "50",
    "topic": "distributed systems"
}
```

3) Purchase by item_number:

For this request, the order serverwas used. this server has only 1 request that takes the book ID as a parameter and sends an “info” query to the Catalog server, if the stock is not 0, then it sends a “updateStock” request that fulfills the purchasing process. While Gateway server was only responsible for forwarding the requests to the Catalog and Order servers and forwarding the responses to the client.



```
{
    "message": "Purchase request processed for book 1"
}
```