An-Najah National University

Faculty of Engineering

Computer Engineering Department

Distributed Operation Systems

Lab 1: Bazar.com: A Multi-tier Online Book Store

Roa Abo Libdeh, 12028262

Noor Sayeh, 12027678

Introduction

The store will apply a microservices approach at each tier of its two-tier web architecture, comprising a front-end and a backend where the front-end tier provides initial processing on the user requests. The back-end consists of two tiers: Catalog server and Order Server.

Materials

To complete this lab, we used the following:

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

Architecture

1- Frontend

The front-end is what users see and use. It handles what the user clicks or types. And coordinates the communication of the user interface with the Catalog service or Order service according to the users' requests. So it acts as an entry point for users' requests.

2- Catalog-server

This part of the system stores all the book details such as title, cost, stock, and topic. It is a microservice responsible for maintaining and providing information on the available books. It implements a RESTful API that allows for operations such as updating book info or searching for books by topic or ID. and it's implemented as an independent process.

3- Order-server

It's responsible for managing customer orders. The Order-server, being a microservice, talks to the front-end and catalog server in order to ease the purchasing process. This microservice provides a single operation endpoint via RESTful API and is designed to handle the purchase process efficiently.

Implementation

The project was built using **Node.js** and **Express.js**, which are great for creating lightweight microservices. **Express.js** is a flexible, minimalist web application framework for **Node.js**, providing a powerful set of features for web and mobile apps. For testing and interacting with the **REST endpoints**, we used **Postman**, a tool that allows making and testing HTTP requests between servers. For the database, we used a **DB.js** file, which holds a JavaScript object representing a small collection of books. Each book entry includes details like an **ID**, **title**, **stock**, **cost**, and **topic**.

Testing the APIs using API test tool

We tested APIs using Postman, here are the endpoint available:

Frontend APIs:

URL	METHOD	DESCRIPTION
/Bazarcom/info/:id	GET	Get information about a book by its <ID>
/Bazarcom/search/:topic	GET	Search for books by their <topic>, return all matching books IDs and Titles
/Bazarcom/ purchase/:id	POST	Buy a book by its <ID>

Catalog APIs:

URL	METHOD	DESCRIPTION
/catalogServer/query	GET	Allows users to search a book based on either topic or id
/CatalogServer/updateStock/:itemNumber	POST	Update the stock and cost of a specific book based on its (ID) .

Order APIs:

URL	METHOD	DESCRIPTION
/OrderServer/purchase/:itemNumber	POST	Handles the purchase of a specific book by its (ID)

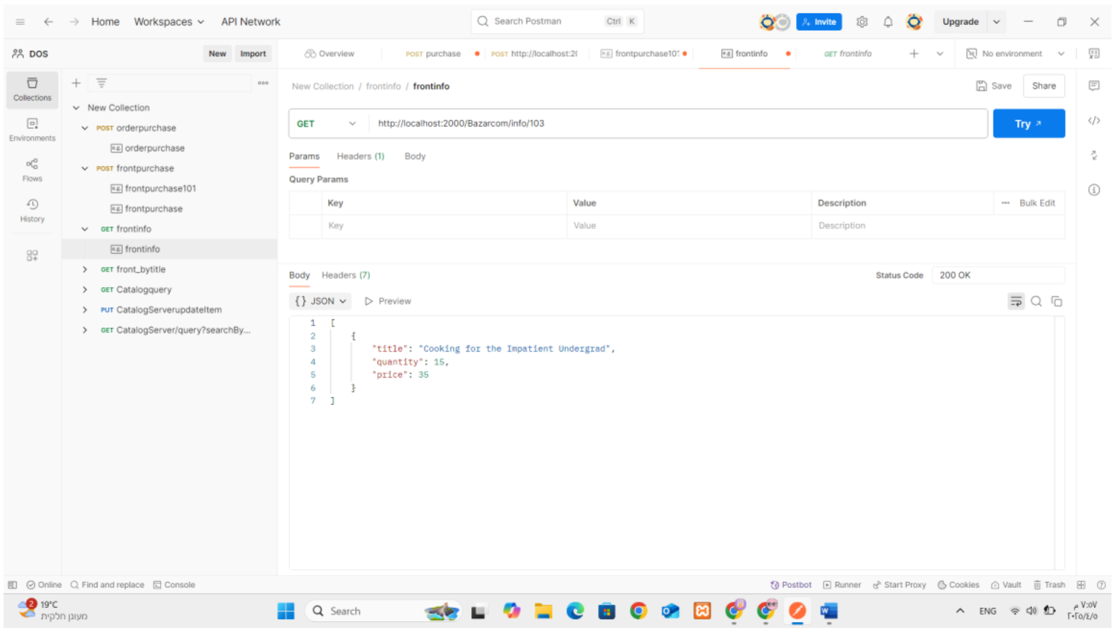
GitHub link: https://github.com/DOS

Examples:

From frontend:

I. Get information about a book by its <ID>

Endpoint: http://localhost:2000/Bazarcom/info/:id



New Collection / frontinfo / **frontinfo**

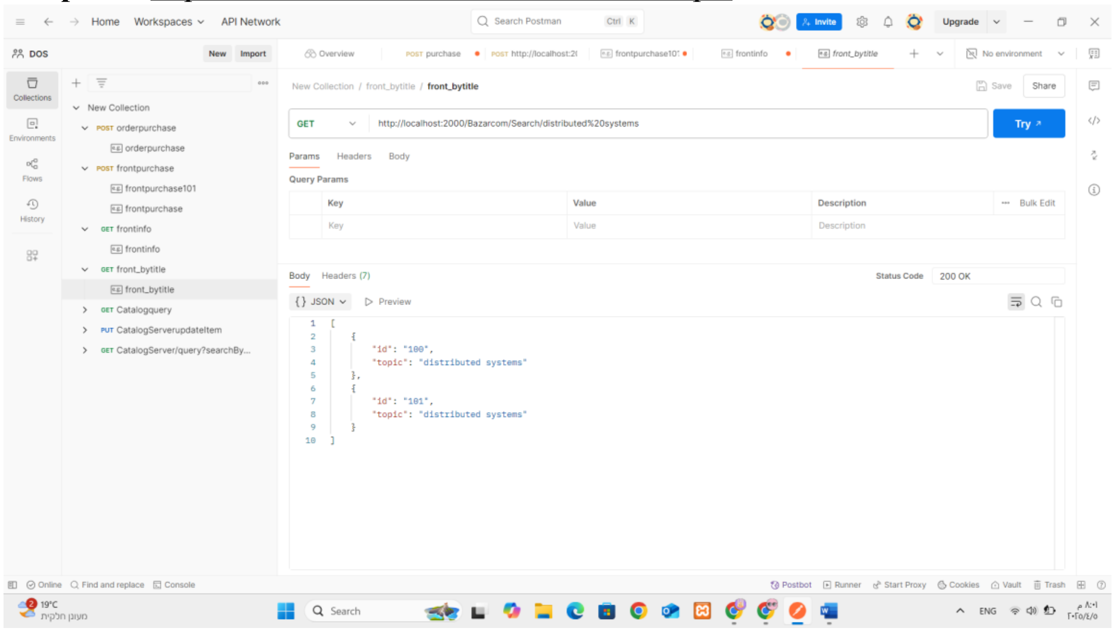
GET http://localhost:2000/Bazarcom/info/103

Body Headers (7) Status Code 200 OK

```
[
    {
        "title": "Cooking for the Impatient Undergrad",
        "quantity": 15,
        "price": 35
    }
]
```

II. Search for books by their <title>, return all matching books IDs and Titles

Endpoint: http://localhost:2000/Bazarcom/Search/:topic



New Collection / front_bytitle / **front_bytitle**

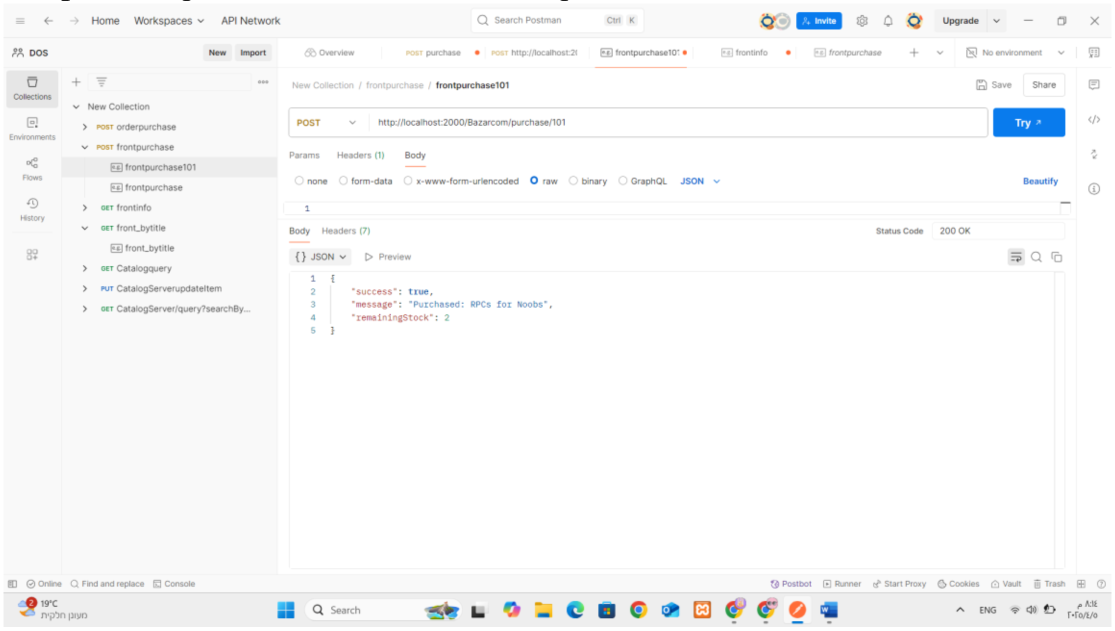
GET http://localhost:2000/Bazarcom/Search/distributed%20systems

Body Headers (7) Status Code 200 OK

```
[
    {
        "id": "100",
        "topic": "distributed systems"
    },
    {
        "id": "101",
        "topic": "distributed systems"
    }
]
```

III. Buy a book by its <ID>

Endpoint: http://localhost:2000/Bazarcom/purchase/:id



New Collection / frontpurchase / frontpurchase101

POST http://localhost:2000/Bazarcom/purchase/101

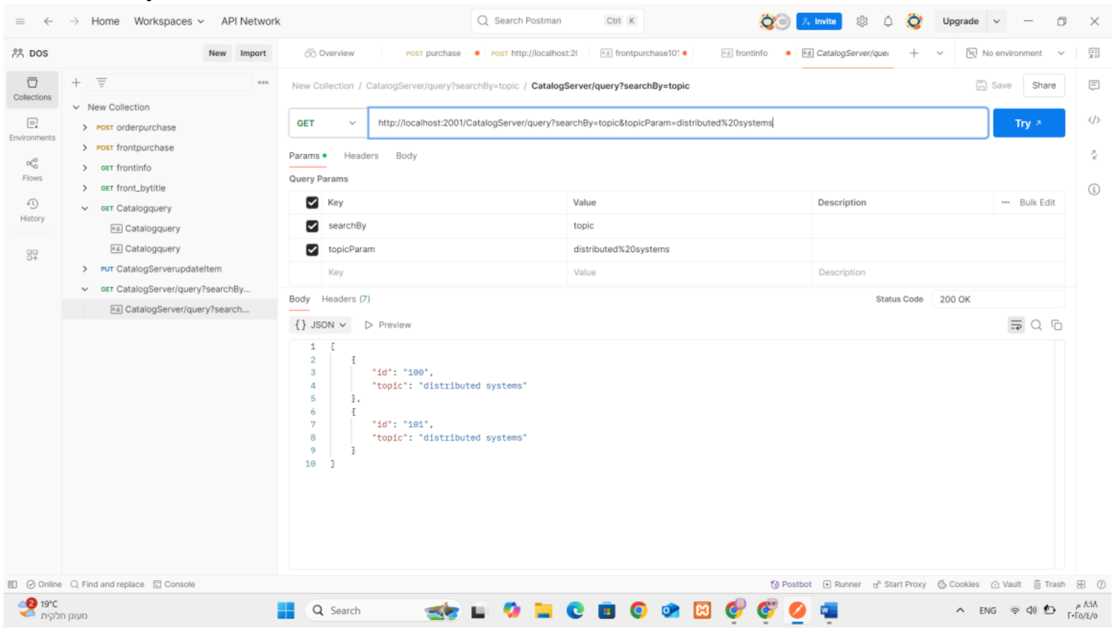
Status Code 200 OK

```
{
    "success": true,
    "message": "Purchased: RPCs for Noobs",
    "remainingStock": 2
}
```

From Catalog-server:

I. Allows users to search a book based on either topic or id

Endpoint with Example:
http://localhost:2001/CatalogServer/query?searchBy=topic&topicParam=distributed%20systems



New Collection / CatalogServer/query?searchBy=topic / CatalogServer/query?searchBy=topic

GET http://localhost:2001/CatalogServer/query?searchBy=topic&topicParam=distributed%20systems

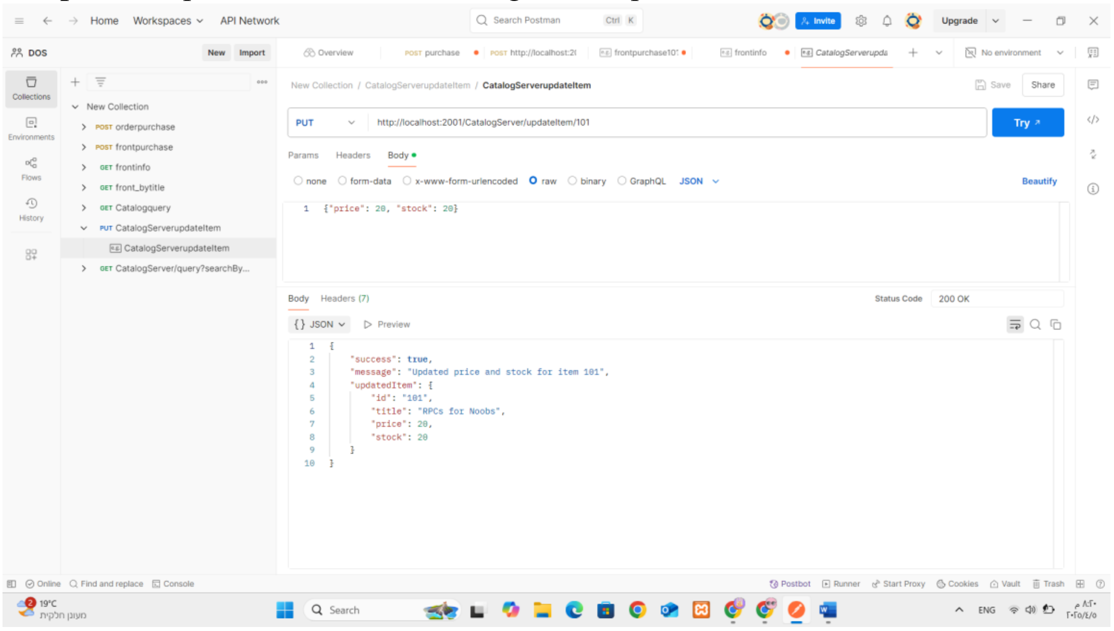
Query Params

Key	Value	Description
searchBy	topic	
topicParam	distributed%20systems	

Status Code 200 OK

```
[
    {
        "id": "100",
        "topic": "distributed systems"
    },
    {
        "id": "101",
        "topic": "distributed systems"
    }
]
```

II. Update the stock of a specific book based on its (ID) when it has bought.
Endpoint: http://localhost:2001/CatalogServer/updateItem/:id



PUT http://localhost:2001/CatalogServer/updateItem/101

Body: raw, JSON

```
{"price": 20, "stock": 20}
```

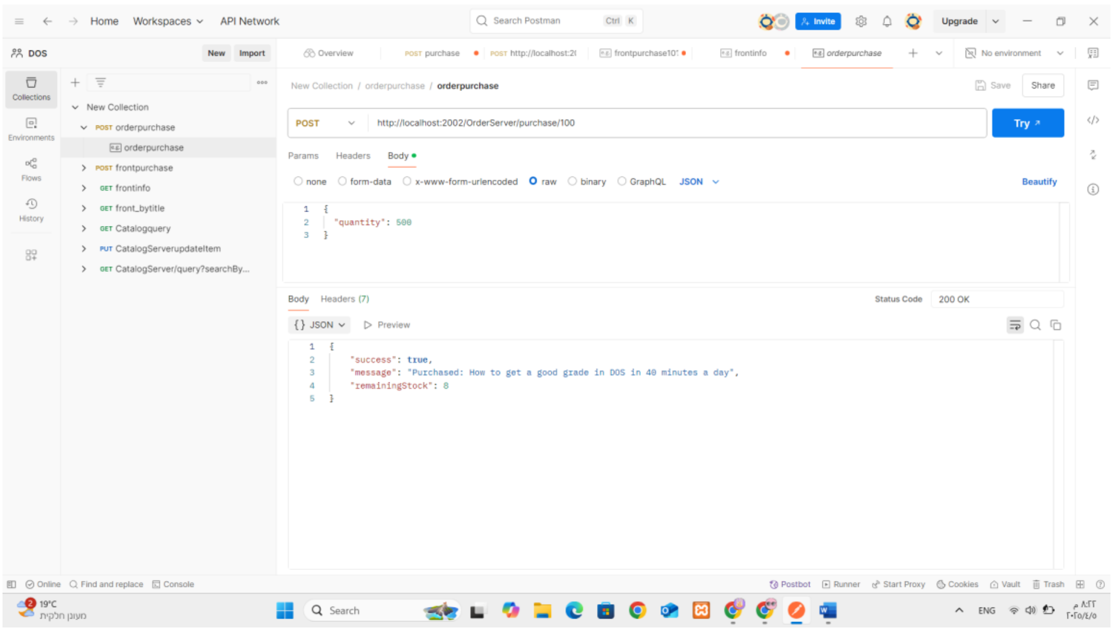
Status Code: 200 OK

```
{
    "success": true,
    "message": "Updated price and stock for item 101",
    "updatedItem": {
        "id": "101",
        "title": "RPCs for Noobs",
        "price": 20,
        "stock": 20
    }
}
```

From Order-server:

Handles the purchase of a specific book by its (ID).

Endpoint: http://localhost:2002/OrderServer/purchase/:id



POST http://localhost:2002/OrderServer/purchase/100

Body: raw, JSON

```
{
  "quantity": 500
}
```

Status Code: 200 OK

```
{
    "success": true,
    "message": "Purchased: How to get a good grade in DOS in 40 minutes a day",
    "remainingStock": 8
}
```

Note: The rest of outputs are existed on our **GitHub** repository (existing Docker):

GitHub-results

Conclusion

In this lab, we developed a multitier online bookstore system using microservices architecture. We implemented a multitier online bookstore system by realizing the catalog and order servers independently as microservices, mediating all user interactions via a frontend. This made the entire platform scalable and efficient. The use of Docker containers developed an easy deployment process whereby each service can be deployed and managed independently. Additionally, testing via Postman allowed us to verify that all APIs went well and provided the necessary endpoints for book search, stock management, or order processing. We learned from this lab quite a great deal of how to develop distributed systems using state-of-the-art web technologies and witnessed the flexibility that microservices-based architecture allows for in handling complex processes such as online purchases.