AN-NAJAH NATIONAL UNIVERSITY

PALESTINE

Bazar.com Project Report

Lab 1&2

Distributed Operating Systems

By

Hiba Al Kurd,

Manar Sholi

Under the supervision of Dr. Samer Arandi

Submitted on 17/7/2021

Introduction

A Multi-tier Online Book Store implemented using micro services for the frontend server and backend that consists of two servers(catalog and order servers) in addition to a client console application that communicates with the front-end server.

The store microservices are built with Restful architectural style, using Flask framework, Python, and nginx web server. The application is run on virtual machines with each microservice running on a different machine and it is run in Docker using nginx as web server and uwsgi for deployment with each server running in a different container.

The store has 4 books with ids, titles, prices, and quantities, in this project we will do some operations on books through the client application or Postman.

Architecture

The application employs a two-layer web design separated into front-end and back-end and uses microservices at each tier , one at the front-end and two micro services at the backend for orders and catalog.

The Front-end server receives the requests then sends a corresponding request to the back-end servers, the back-end servers process the request and respond to the front-end server, each micro service handles a specific part of the application.

Catalog Server

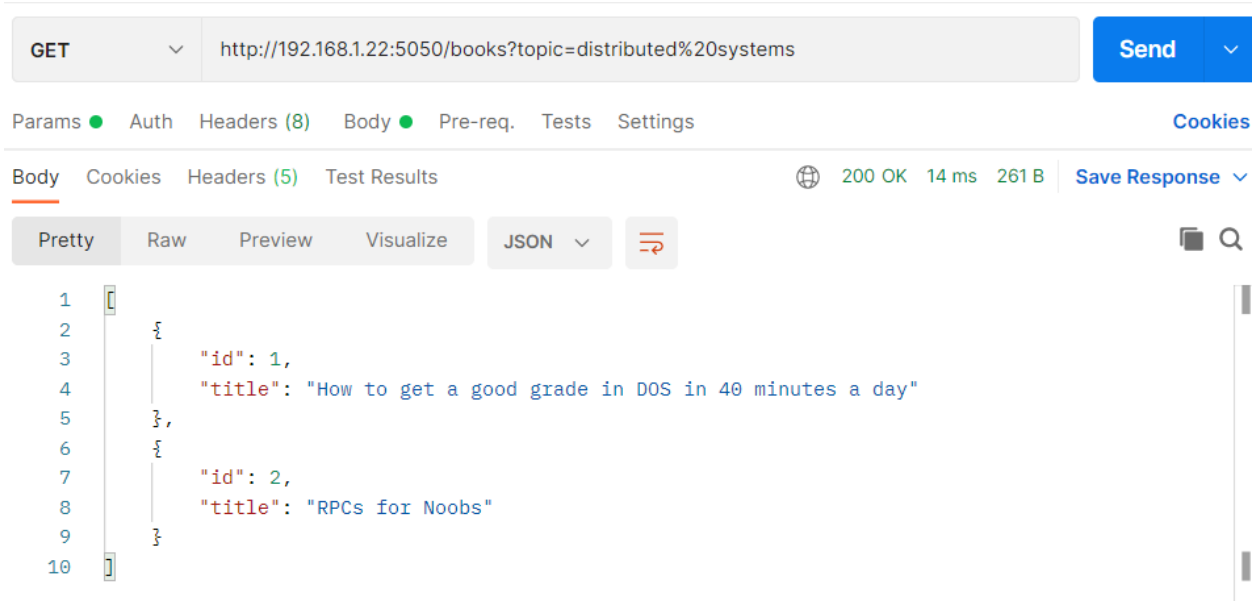
The catalog server keeps the books log using a JSON file which consists of four book entries each containing id, topic, title, quantity, price .


```
{
    "id": 1,
    "price": 100,
    "quantity": 197,
    "title": "How to get a good grade in DOS in 40 minutes a day",
    "topic": "distributed systems"
}
```

The catalog server supports three requests as follows:

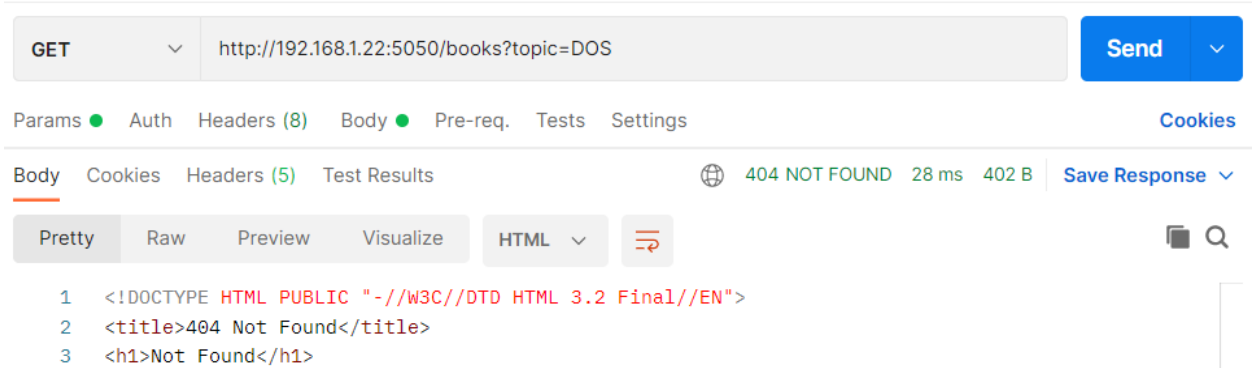
Method	Url	Request Body	Response Status Code	Response Body On success
GET	/books?topic={topic}	X	200 ok , 404 Not found	List of all books titles and ids with the given topic

Eg. Successful Request



Eg. Failed Request

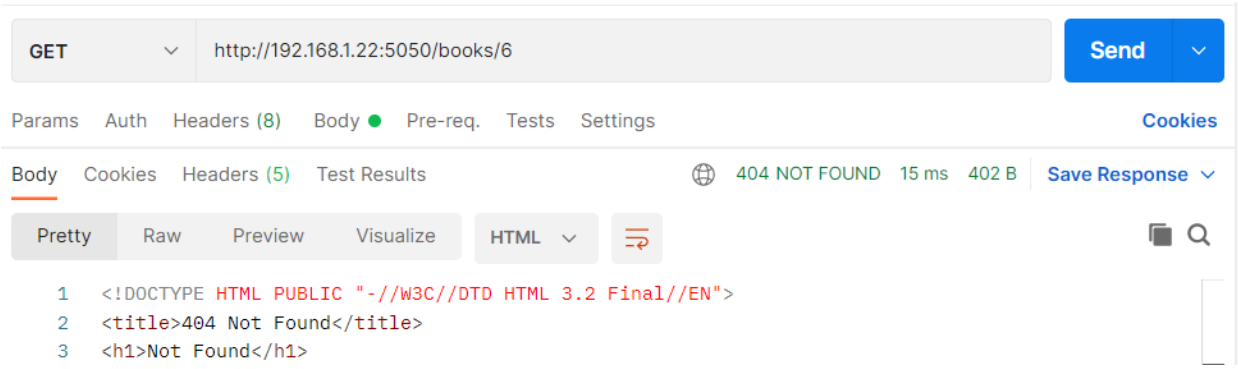
Caused by providing a topic that's not found in the database



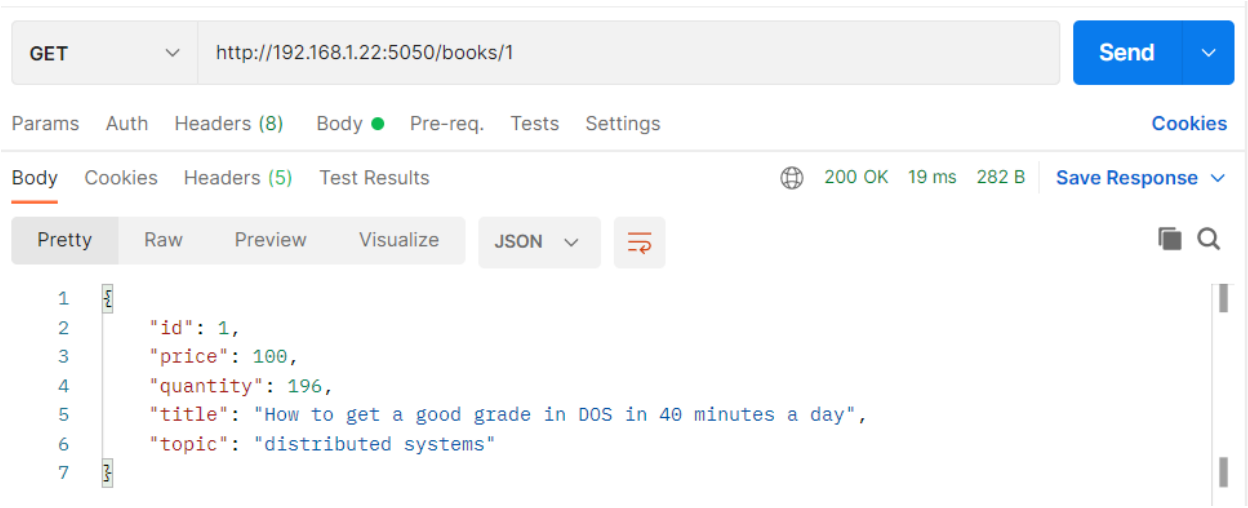
GET	/books/{id}	X	200 ok, on success 404 Not found , invalid id	Book block with all its information

Eg. Failed Request

Caused by providing invalid book number

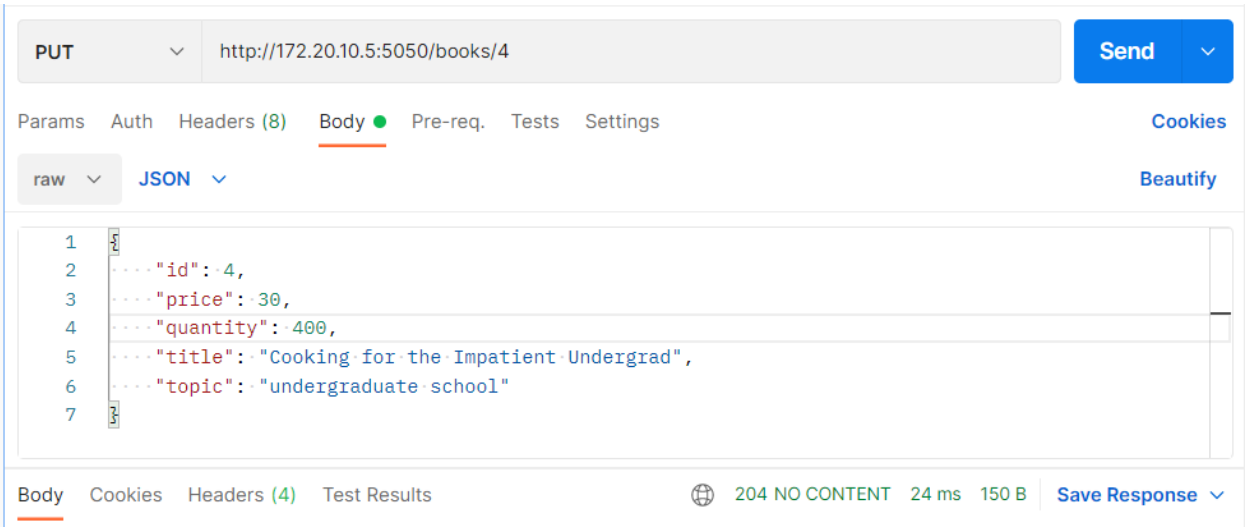


Eg. Successful Request



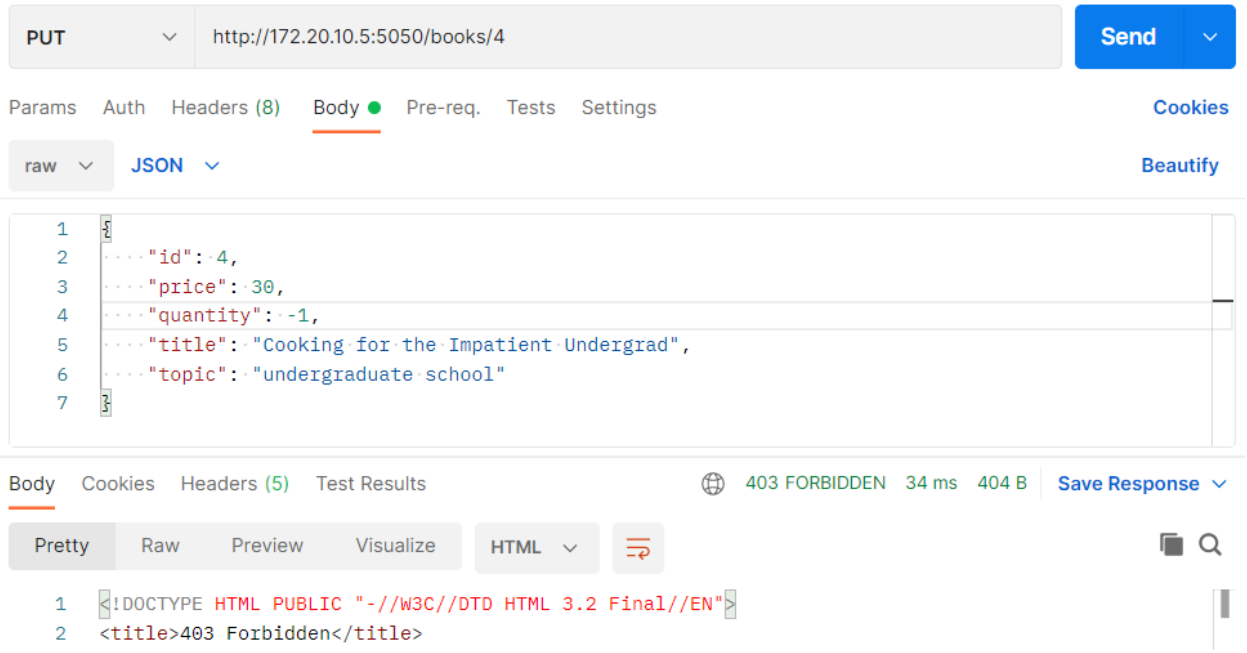
PUT	/books/{id}	The new book entity	204 updated successfully 403 Forbidden	X

Eg. Successful Request



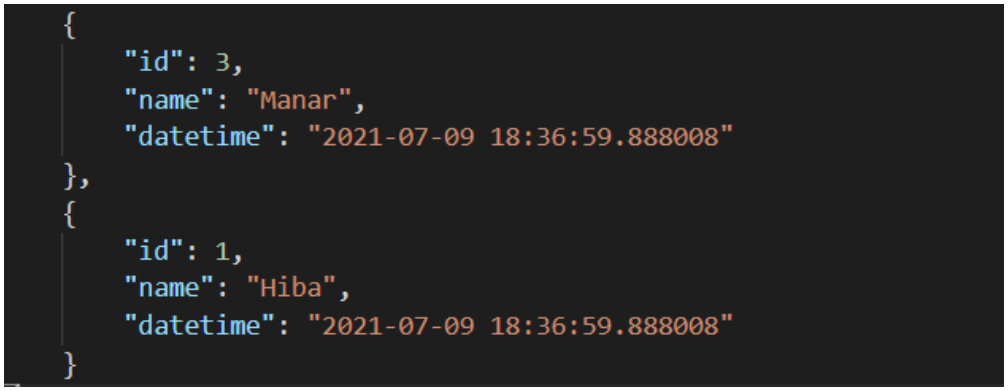
Eg. Failed Request

Caused by invalid quantity value or two clients accessing same resource



Orders Server

The Order server keeps track of the books that are being bought in a JSON file. The order server receives POST request from the front-end server to create a new order and update the book quantity in the catalog server.

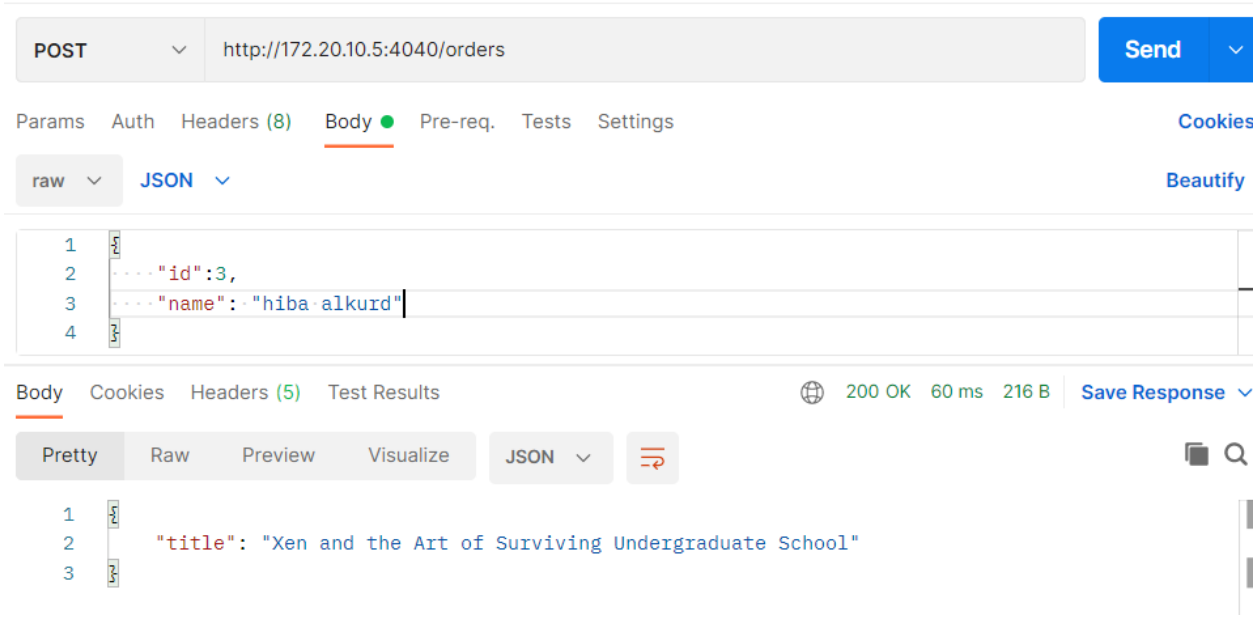


The order server supports one request as follows:

Method	Url	Request Body	Response Status Code	Response Body On success
POST	/orders	Requested book id client name	200 ok 404 Not found 403 Forbidden	Bought book with its title

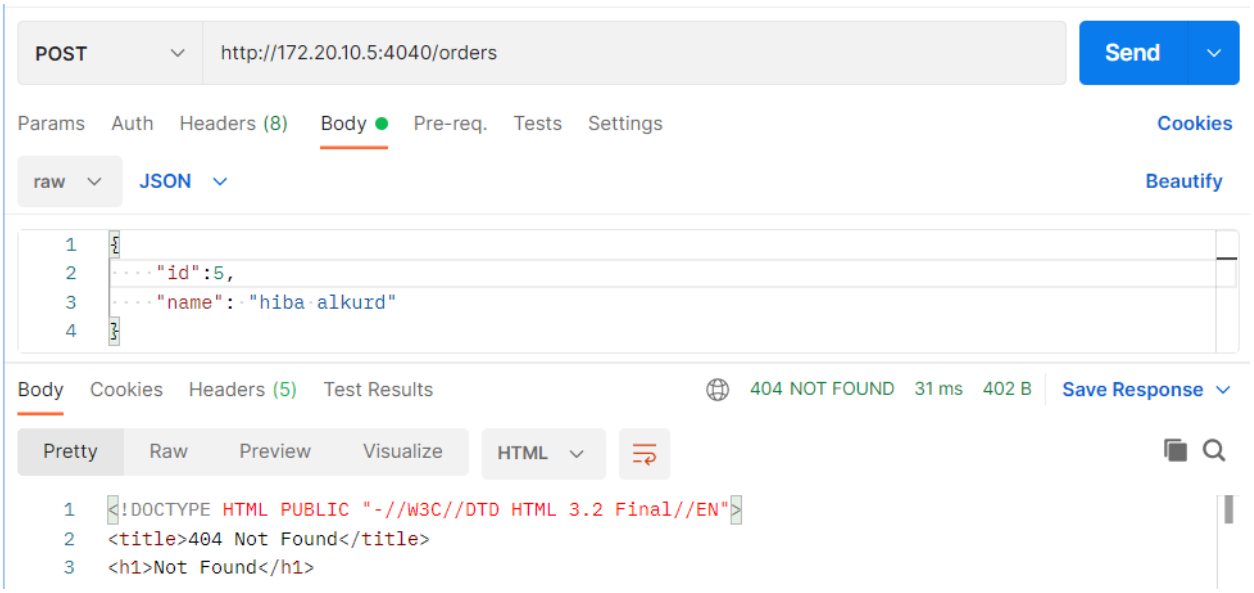
When received the order server sends a GET request to the catalog server to check the quantity and the book id after that it sends a PUT request to update the book quantity in the catalog server then it creates a new order and adds it to the orders JSON file.

Eg. Successful Request



Eg. Failed Request

Caused by invalid id



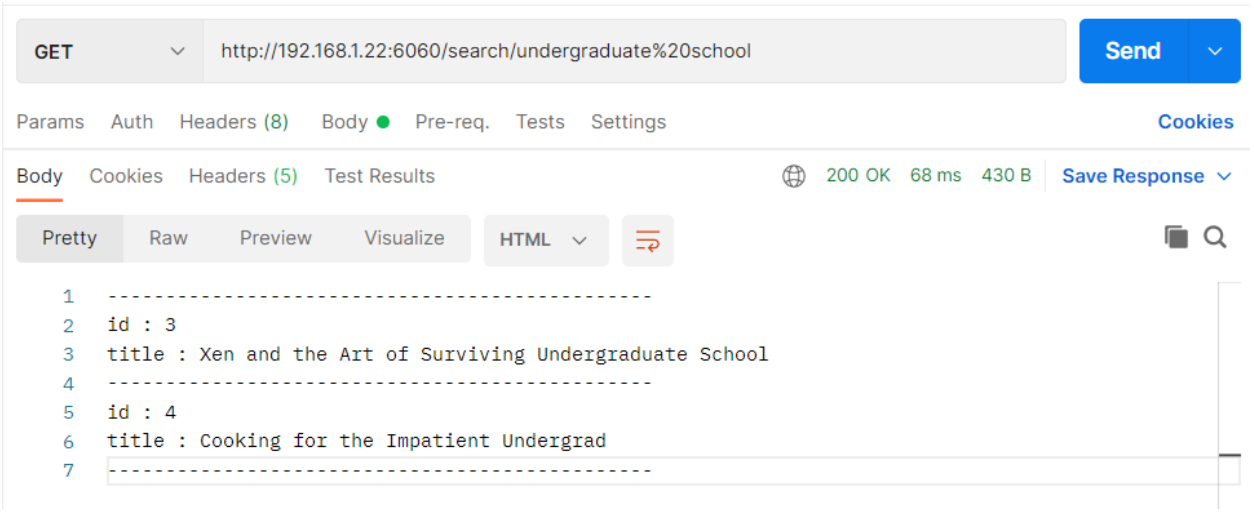
Front-end Server

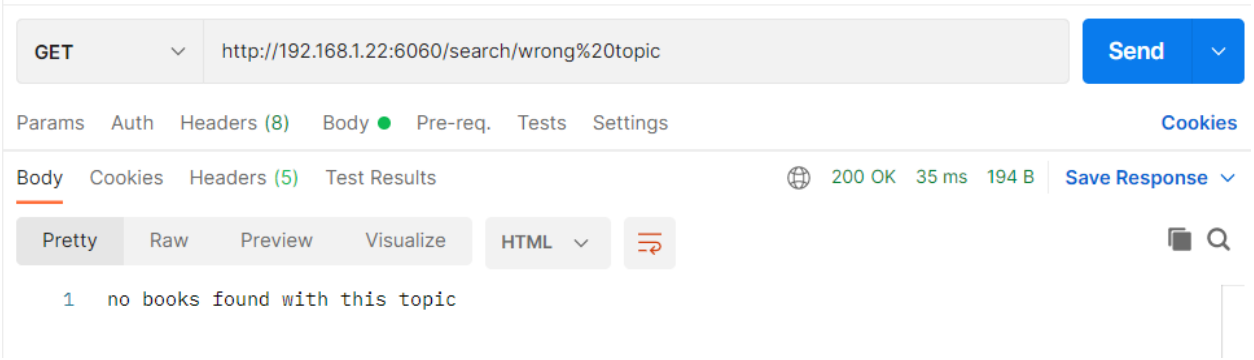
The front-end server handles the client requests then returns the response with a friendly format instead of the received JSON response

The Front-end server supports three requests as follows:

Method	Url	Request Body	Response Status Code	Response Body On success
GET	/search/{topic}	X	200 ok	List of all books titles and ids of this topic

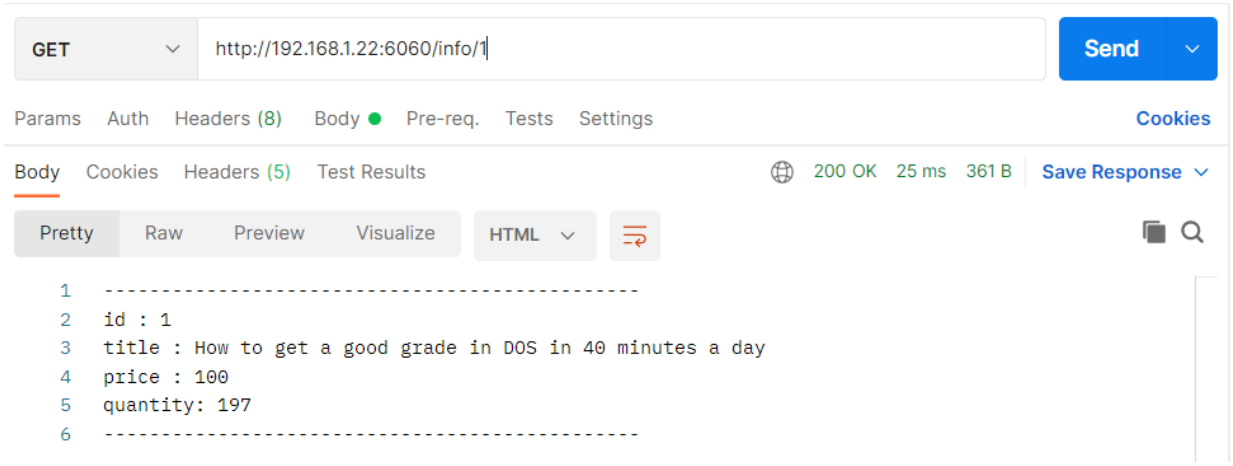
Eg.

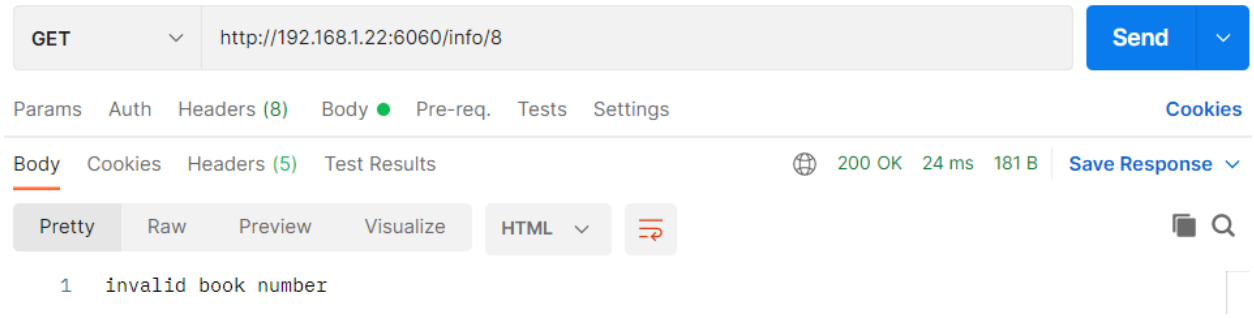




GET	/info/{id}	X	200 ok	The book's information

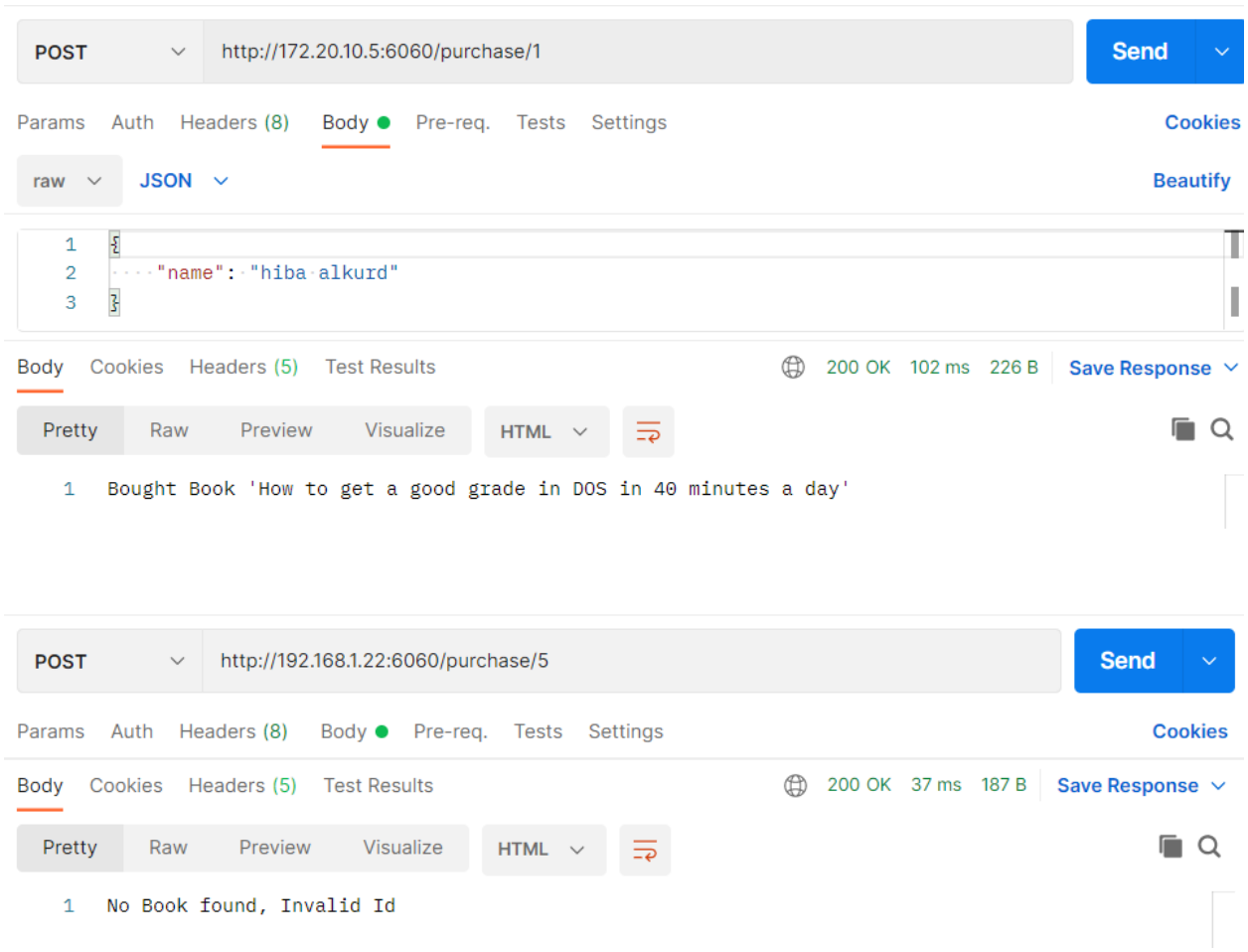
Eg.





POST	/purchase/{id}	client name	200 OK

Eg:



LAB 2

Cache

We added cache support at the frontend server, so any ‘get’ request received is cached inside a json file cache.json.
Initially the cache is empty, and every new request is brought from the catalog server and cached. Any subsequent cached requests are fulfilled at the frontend server.

An additional route was added, ‘/invalidate’ to keep cache consistent with the database, so whenever a resource is updated or added, the catalog server sends an invalidate request to remove that resource from the cache.

POST	/invalidate	Book object	200 ok

Replication

To improve reliability, fault-tolerance, and accessibility, we added two replicas of the catalog server with its database and two replicas of the order server.
To achieve consistency, we used a sequential model with Replicated-write Protocols.

Order Server Replication

When a POST order request is received at one of the replicas, after processing the request the replica sends a POST order to database request to all other replicas to keep the orders database consistent between replicas.

We added new routes to achieve immediate database update.

POST	/db/orders	Order object	200 ok

Catalog Server Replication

When a POST book or PUT book request is received at one of the replicas, after processing the request the replica sends the update/new request to all other replicas to keep the catalog database consistent between replicas.

We added new routes to achieve immediate database update.

POST	/db/books	Book object	200 ok

PUT	/db/books/id	Book object	200 ok

Load Balancing

To deal with replication, we implemented Round Robin algorithm that takes each incoming request and sends it to one of the replicas, the load balancing is implemented at the frontend server, we created a order-servers pool and a catalog-servers pool, for every request a server is picked from the servers pool following round robin algorithm that distributes client requests equally between servers

Performance

Response time (avg) when using docker

	/info/1	/search/{topic}	/purchase/1
cache	**33 ms**	**29 ms**	**120 ms**
no cache	**348 ms**	**350 ms**	**100 ms**

Response time and Latency (avg) when using VM.

	/info/1	**/search/{topic}**	**/purchase/1**
cache	6.3214 ms	4.971 ms	44.5421 ms
no cache	9.9344 ms	14.6706 ms	45.3156 ms

We notice that requests are satisfied at the frontend server cache. They are faster than requests from the catalog server with no cache since the data will be used more than once, and the information is stored in it to access them faster, and saving time, and this brings more users, because the users will not wait a lot of time to get/post data.

Update database latency and overhead

	Total time	Transfer time
with replication	635 ms	120 ms
without replication	595 ms	71 ms

We notice when running the api without replication is faster than with replication, because when replication exists, we will have an overhead to maintain the consistency(send requests to the replicas to update their data) and this takes more time.

Docker

We used docker to run the servers, for each micro service there's two folders one that contains the Flask application and one for the nginx server each folder has a dockerfile to build the containers.

Folder Tree

/server
    /app
        /main.py
        /wsgi.init
        /requirements.txt
        /Dockerfile
    /nginx
        /nginx.conf
        /Dockerfile
    docker-compose.yml

A docker compose file is used to build the containers and connect them together to run the Flask application using nginx server

How to run:

build the container using

$ docker compose-up --build

to run all services on the same device create a network in docker and connect the containers using:

$ docker network create -d bridge shared-net

$ docker network connect shared-net [container id]

Virtual machines

We used 3 virtual machines, each machine for one of the servers that we mentioned previously, and used the ip addresses in the GET, POST, PUT URLs.