DOS PROJECT PART 1
BAZAR.COM

Students:

Sarah Hinno 11926569

Marah Direeni 11923808

- Objectives:

- Learn more about Microservice systems.
- Build system based on Microservices using 3 Microservices.
- Dealing with Docker and getting familiar with it.

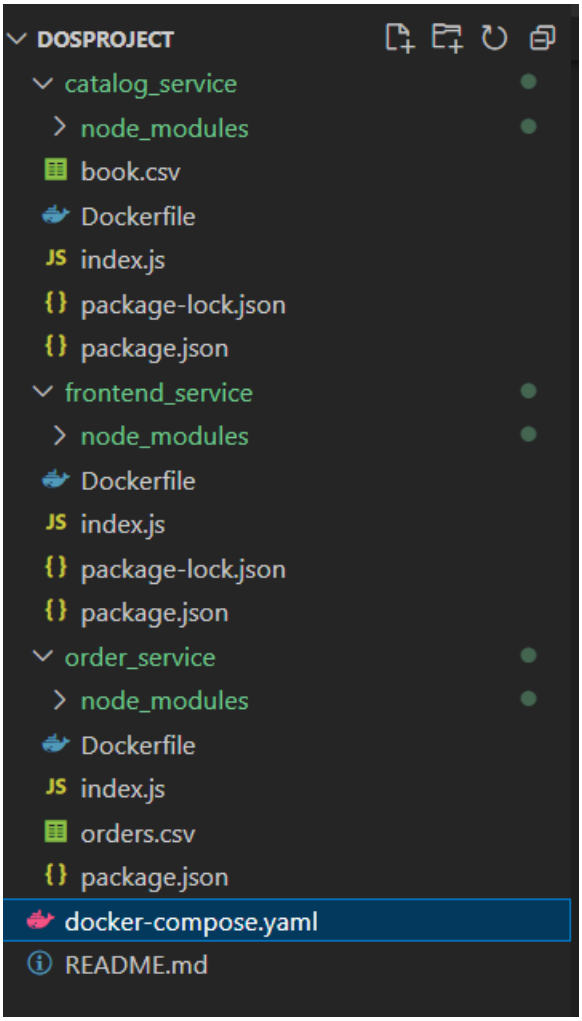
- Services:

- Frontend service.
- Catalog service.
- Order service.

- System APIs:

- http://localhost:3001/bazar/search/topic
- http://localhost:3001/bazar/info/ID
- http://localhost:3001/bazar/purchase/ID

- Project structure:



○ Front-end Microservice:

This service is responsible of receiving the requests from user and forward the requests to appropriate server. Additionally, it receives the response from these servers and forward it back to the user.

Takes requests from users for information, search, and purchasing procedures. interacts with the catalog server to perform informational and search functions. exchanges information with the order server in order to make purchases.

In our project, we utilized Nginx as a high-performance web server and reverse proxy. Nginx played a crucial role in handling incoming client requests and efficiently distributing them to the appropriate microservices within our architecture. As a reverse proxy, Nginx served as the entry point for external

requests, allowing us to route traffic to the respective backend services based on predefined rules and configurations.

○ Catalog Microservice:

This service is responsible of search and info requests:

- Search request: user can search any book by topic and this service will respond with a list off all books (title and id of the book) belongs to the specified type in the request.

  http://localhost:3001/bazar/search/topic ➔ topic represent the topic that user is searching for and it sent within URI.

- Info request: user can send request with id of book and this service will respond with all information about this book (id, title, topic, quantity, price).

  http://localhost:3001/bazar/info/ID ➔ ID represent the ID of the book that user is informing about and it sent within URI.

We used csv file to store information's about all books in the system (book.csv) and this file is located on catalog service so can get data from the file when processing the requests.

We need to focus that search and info requests are forwarded to catalog service from the frontend service and responses are sent back to frontend service as well.

Catalog service also handle requests from order service. When new order requests are send the order service send a request to get the quantity of the book ordered. Also after completing the order successfully it sends another request to catalog service to decrease the quantity of the book has been ordered. Since the catalog service has book.csv file it is more logically to make responsible of reducing quantity and getting it.

○ Order Microservice:

This service is responsible of handling order request that sent by the URI:
http://localhost:3001/bazar/purchase/ID ➔ where ID represent the id of the book that user want to order.

Order (purchase) request is forwarded from frontend service and the response of it is sent back to frontend service.

When order service is processing the order operation first need to make sure that the ordered book is in stock so it sent a request to catalog service to get the quantity based on the ID of the book.

```
app.post('/order/purchase/:id', (req, res) => {
  const id = req.params.id;
  console.log("from purchase in order with id : "+id);

  axios.get(`http://catalog:3002/catalog/item/${id}`)
    .then(response => {
```

```
app.get('/catalog/item/:id',async (req, res) => {
  const id =req.params.id;
  const book=await searchBooksbyID(id);

  res.json(book);

});
```

After getting the response from catalog service and making sure that the book is in stock order operation will success and a new record is stored in orders,csv file. This file is located on order server to store every purchase operation. It stores the ID and title of the book and stores the timestamp of the purchase operation when is done.

```
axios.get(`http://catalog:3002/catalog/reduce/${id}`)
  .then(reduceResponse => {
    res.send(`Purchase successful for item number ${id}`);
  })
```

```
app.get('/catalog/reduce/:id',async (req, res) => {
  const id =req.params.id;
  const book=await searchBooksbyID(id);
  reduceQuantity(id);
});
```

Finally, it will send request to catalog service again to reduce the quantity of the purchased book.

Note: all APIs in the system are rest APIs.

○ Dockerfile:

Dockerfiles played a crucial role in containerizing our microservices architecture. Each microservice, including the catalog, order, and frontend

services, had its own Dockerfile tailored to its specific requirements. These Dockerfiles began with base images containing the necessary runtime environments, such as Node.js for our JavaScript-based services. They then proceeded to install dependencies, copy application code into the container, and expose ports for communication. Additionally, we utilized Docker Compose to orchestrate the deployment of multiple containers, defining the relationships and dependencies between services. By using Dockerfiles, we achieved consistency in our development and production environments, ensuring that each service ran in a controlled and isolated environment with its dependencies encapsulated.

Our three services containers are based on the Ubuntu image, which serves as the foundation for setting up their respective environments. During the Docker build process, commands are executed within each Dockerfile to install the necessary packages and dependencies using package managers like apt-get (for Ubuntu) or npm (for Node.js packages).

catalog_service > Dockerfile > FROM

```
FROM ubuntu:latest

WORKDIR /home

RUN apt-get update && \
    apt-get install -y nodejs npm

RUN mkdir -p /home/microservices/catalog_service

WORKDIR /home/microservices/catalog_service

COPY package*.json ./

RUN npm install express axios fs csv-parser nodemon

RUN npm install

COPY . .

EXPOSE 3002

CMD npx nodemon index.js
```

○ Docker-compose file:

Docker Compose is a tool that simplifies the management of multi-container Docker applications. It allows you to define and run multi-container Docker applications using a YAML file called docker-compose.yml. This file specifies

the services, networks, and volumes required for our application, as well as their configurations and dependencies.

In our project, Docker Compose is used to define and orchestrate the interaction between the frontend, catalog, and order services. Each service is defined within the docker-compose.yml file, specifying its build context, dependencies, ports, and other configurations. Docker Compose ensures that these services can communicate with each other seamlessly by creating a shared network for communication between containers.
Using docker-compose file we can run and build the 3 containers by single command: docker-compose up –build

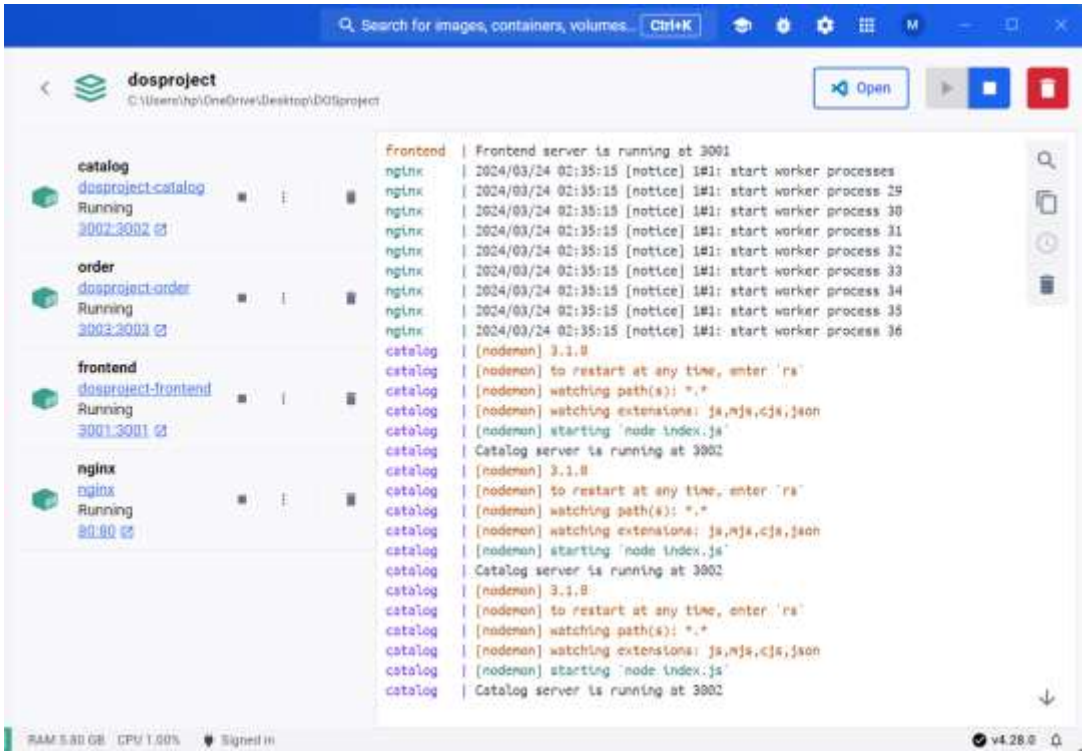
Our docker-compose.yml file:

docker-compose.yaml

```yaml
version: "3"
services:
  nginx:
    image: nginx
    container_name: nginx
    ports:
      - "80:80"
    depends_on:
      - frontend

  catalog:
    build: ./catalog_service
    container_name: catalog
    ports:
      - "3002:3002"
    volumes:
      - ./catalog_service:/home/microservices/catalog_service

  order:
    build: ./order_service
    container_name: order
    ports:
      - "3003:3003"
    volumes:
      - ./order_service:/home/microservices/order_service

  frontend:
    build: ./frontend_service
    container_name: frontend
    ports:
      - "3001:3001"
    volumes:
      - ./frontend_service:/home/microservices/frontEnd_service
    depends_on:
      - catalog
      - order
```

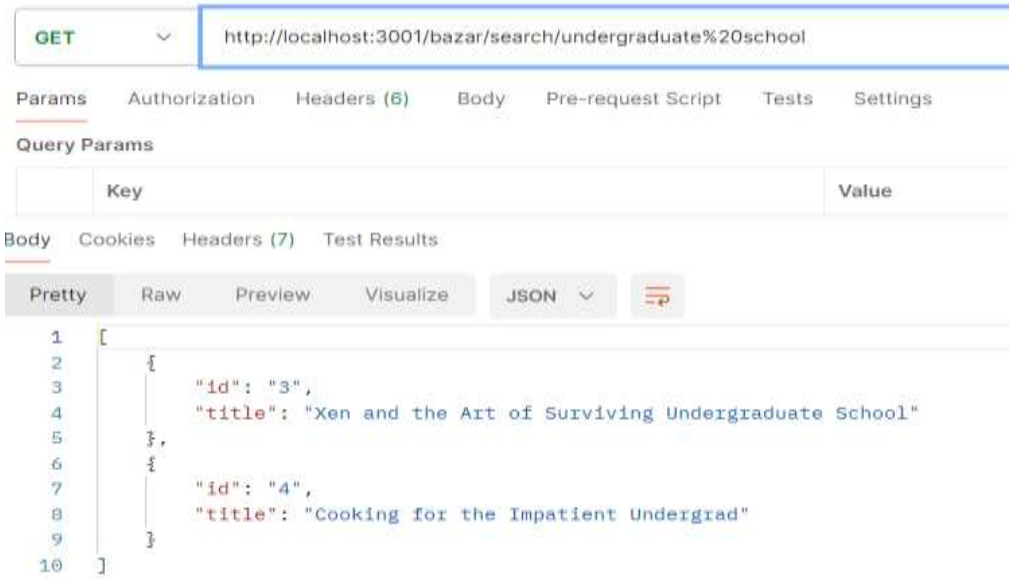
○ Test and Results:

To test our system and the APIs we used postman and terminal to test. First step from the terminal run this command docker-compose up –build to build the services and run the containers.



Then from postman:

- Test search using this URI : http://localhost:3001/bazar/search/topic



GET http://localhost:3001/bazar/search/cooking

Params Authorization Headers (6) Body Pre-request Script Tests

Query Params

	Key	V

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2     "message": "No books found for the specified topic"
3 }
```

- Testing info using URI: http://localhost:3001/bazar/info/ID

GET http://localhost:3001/bazar/info/1

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

	Key	Value

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

```
1 [
2     {
3         "Id": "1",
4         "Title": "How to get a good grade in DOS in 40 minutes a day",
5         "Quantity": "9",
6         "Price": "20.0",
7         "Topic": "distributed systems"
8     }
9 ]
```

GET http://localhost:3001/bazar/info/12

Params Authorization Headers (6) Body Pre-request Script Tests

Query Params

Key	Va

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize HTML

```
1  No book for this ID 12
```

- Testing purchase using URI: http://localhost:3001/bazar/purchase

book.csv file before purchasing book with id =2:

catalog_service > book.csv

```
1  ID,Title,Quantity,Price,Topic
2  1,How to get a good grade in DOS in 40 minutes a day,9,20.0,distributed systems
3  2,RPCs for Noobs,4,25.0,distributed systems
4  3,Xen and the Art of Surviving Undergraduate School,8,25.0,undergraduate school
5  4,Cooking for the Impatient Undergrad,9,12.0,undergraduate school
```

GET http://localhost:3001/bazar/purchase/2

Params Authorization Headers (6) Body Pre-request Script Test

Query Params

Key

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize HTML

```
1  Purchase successful for item number 2
```

book.csv after purchasing successfully:

catalog_service > book.csv

```csv
ID,Title,Quantity,Price,Topic
1,How to get a good grade in DOS in 40 minutes a day,9,20.0,distributed systems
2,RPCs for Noobs,3,25.0,distributed systems
3,Xen and the Art of Surviving Undergraduate School,8,25.0,undergraduate school
4,Cooking for the Impatient Undergrad,9,12.0,undergraduate school
```

GET http://localhost:3001/bazar/purchase/22

Params Authorization Headers (6) Body Pre-request Script Tests S

Query Params

Key	Val

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize HTML

```
No books for this ID 22
```

GET http://localhost:3001/bazar/purchase/2

Params Authorization Headers (6) Body Pre-request Script T

Query Params

Key

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize HTML

```
Purchase failed: Item is out of stock
```