An-Najah National University

Department of Computer Engineering

Distributed Operation Systems

Project

Part #1

Submitted to:

Dr. Samer Arande

Done By:

Adam Yaesh (12028670)

Ayham Omar Al-Dwairy (12029113)

Introduction

Brief overview of the project objective: to design Bazar.com, described as the world's smallest book store with only four titles, using a two-tier web design and microservices architecture

This project involves designing and implementing an online bookstore named Bazar.com using a microservices architecture. The store is unique in its offering, selling only four books. The project is structured around a two-tier web design, employing microservices for both the front-end and back-end tiers. Below is a detailed breakdown of the project's components and objectives.

Project Components

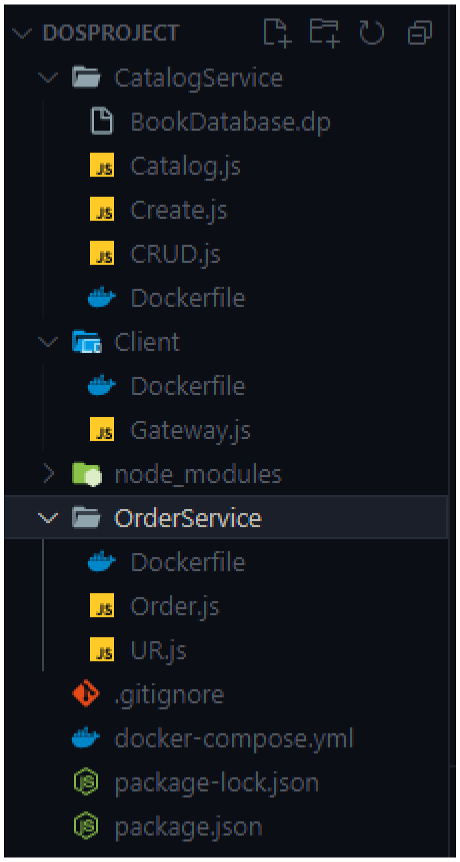
- **Front-End Tier:** Serves as the interface for user requests, handling initial processing. It supports three operations:
    1. search(topic): Allows users to search for books by topic.
    2. info(item_number): Provides details such as stock number and cost for a given item.
    3. purchase(item_number): Processes purchase requests for a specified item.
- **Back-End Tier:** Comprises two components:
    1. Catalog Server**:** Maintains the book catalog, tracking the stock number, cost, and topic of each book.
    2. Order Server**:** Keeps a record of all orders received, verifying stock availability with the Catalog Server before processing purchases

RESTful Interface

- The system must implement HTTP REST interfaces for each component's operations (search, info, purchase).
- REST endpoints should be structured in a way that supports easy interaction and integration, using JSON for data exchange.
- The system should handle concurrent requests efficiently, leveraging the built-in concurrency features of the chosen web framework.

The application deployable across different machines in a distributed manner, using Docker containers for isolation and ease of deployment.

Project Hierarchy



```
DOSPROJECT
  CatalogService
    BookDatabase.dp
    Catalog.js
    Create.js
    CRUD.js
    Dockerfile
  Client
    Dockerfile
    Gateway.js
  node_modules
  OrderService
    Dockerfile
    Order.js
    UR.js
  .gitignore
  docker-compose.yml
  package-lock.json
  package.json
```

For each service, we made a special file that includes the service itself, all the supplier files, and the Dockerfile. This helps us run everything smoothly using Docker Containers.

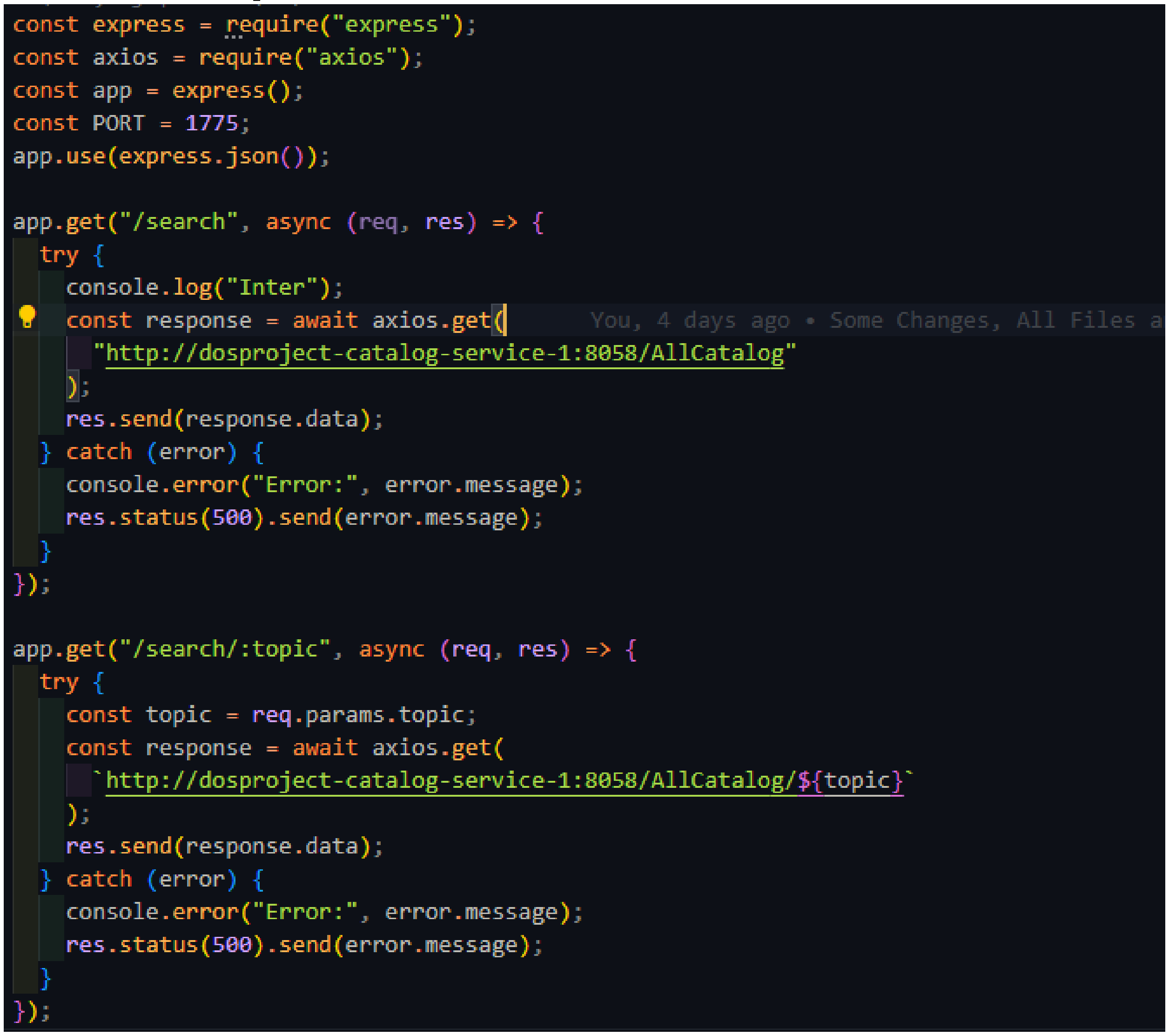
We employ the docker-compose file to coordinate the simultaneous execution of all Docker files. It helps us determine dependencies between services and specify the ports they operate on, ensuring efficient and synchronized deployment of our applications.

Frontend Service

We utilize NodeJS for service implementation, leveraging the express library to handle various request methods such as POST and GET. Additionally, we employ axios for facilitating request transfers between services. Our services operate on port number 1775, selected for its availability and compatibility with our system.

In our setup, the first search service acts as a bridge, efficiently routing incoming requests to the catalog service within our Docker network. By referencing the container name instead of localhost, we ensure seamless communication between services. Should the catalog service process the request without encountering any issues, a response with a status code of 200 is dispatched, containing the relevant data. However, if any errors arise during processing, our system promptly responds with a status code of 500, indicating an issue that requires attention. This robust approach ensures smooth operation and reliable data transmission within our service architecture.

The code that implements it is below:

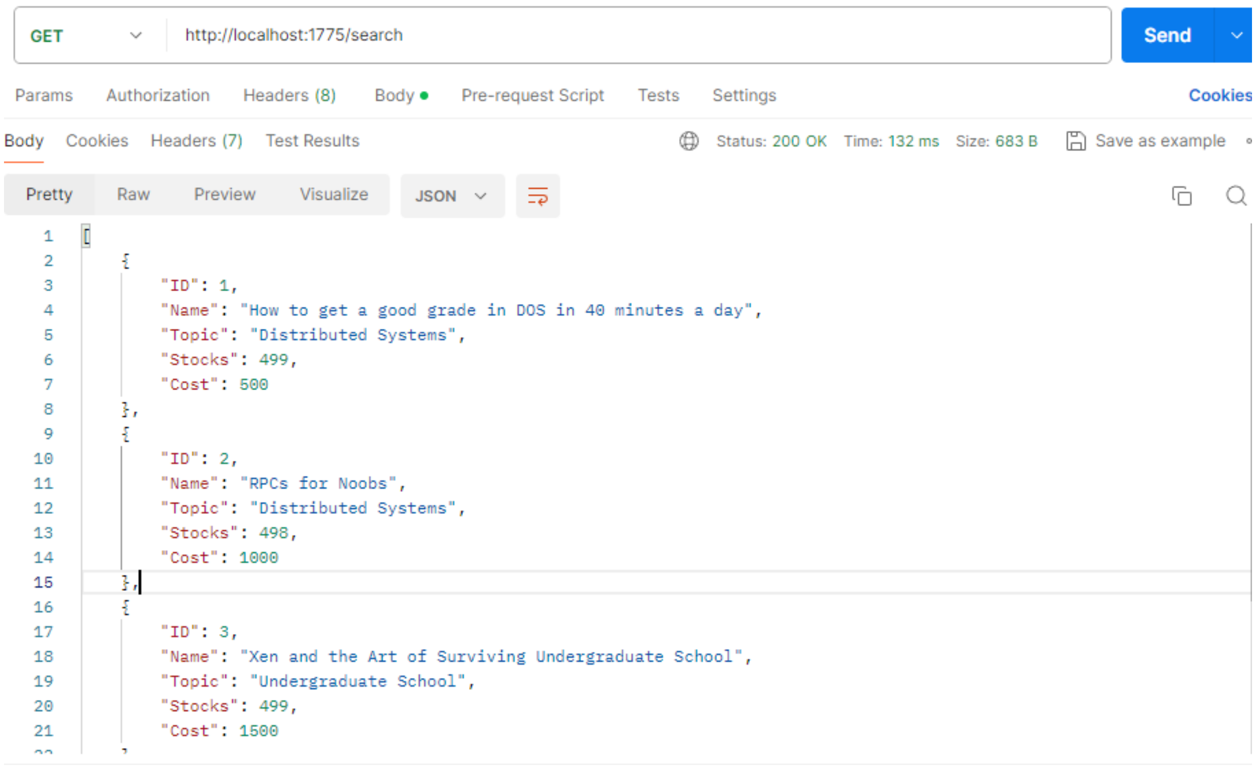


```
const express = require("express");
const axios = require("axios");
const app = express();
const PORT = 1775;
app.use(express.json());

app.get("/search", async (req, res) => {
  try {
    console.log("Inter");
    const response = await axios.get(       You, 4 days ago • Some Changes, All Files a
      "http://dosproject-catalog-service-1:8058/AllCatalog"
    );
    res.send(response.data);
  } catch (error) {
    console.error("Error:", error.message);
    res.status(500).send(error.message);
  }
});

app.get("/search/:topic", async (req, res) => {
  try {
    const topic = req.params.topic;
    const response = await axios.get(
      `http://dosproject-catalog-service-1:8058/AllCatalog/${topic}`
    );
    res.send(response.data);
  } catch (error) {
    console.error("Error:", error.message);
    res.status(500).send(error.message);
  }
});
```

Note that we use instead of localhost the name of the container as we mention earlier.

And the second service is to get the books with specific topic, and return a json representation response.

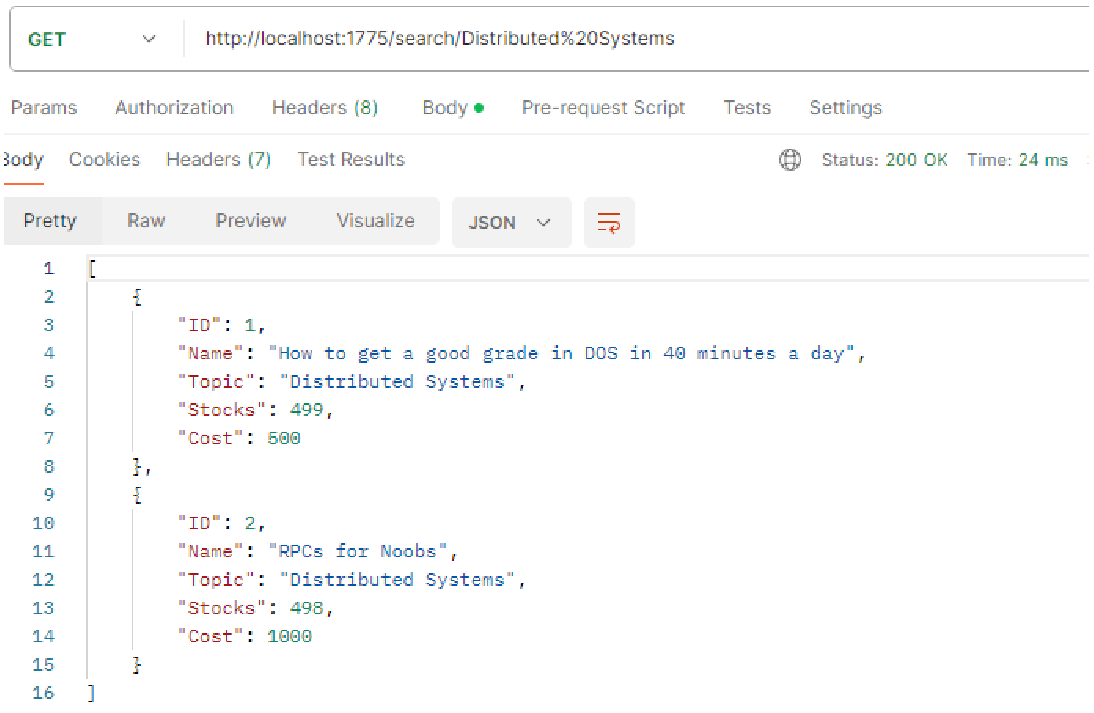
The output:


GET http://localhost:1775/search

Status: 200 OK Time: 132 ms Size: 683 B

```
[
    {
        "ID": 1,
        "Name": "How to get a good grade in DOS in 40 minutes a day",
        "Topic": "Distributed Systems",
        "Stocks": 499,
        "Cost": 500
    },
    {
        "ID": 2,
        "Name": "RPCs for Noobs",
        "Topic": "Distributed Systems",
        "Stocks": 498,
        "Cost": 1000
    },
    {
        "ID": 3,
        "Name": "Xen and the Art of Surviving Undergraduate School",
        "Topic": "Undergraduate School",
        "Stocks": 499,
        "Cost": 1500
```

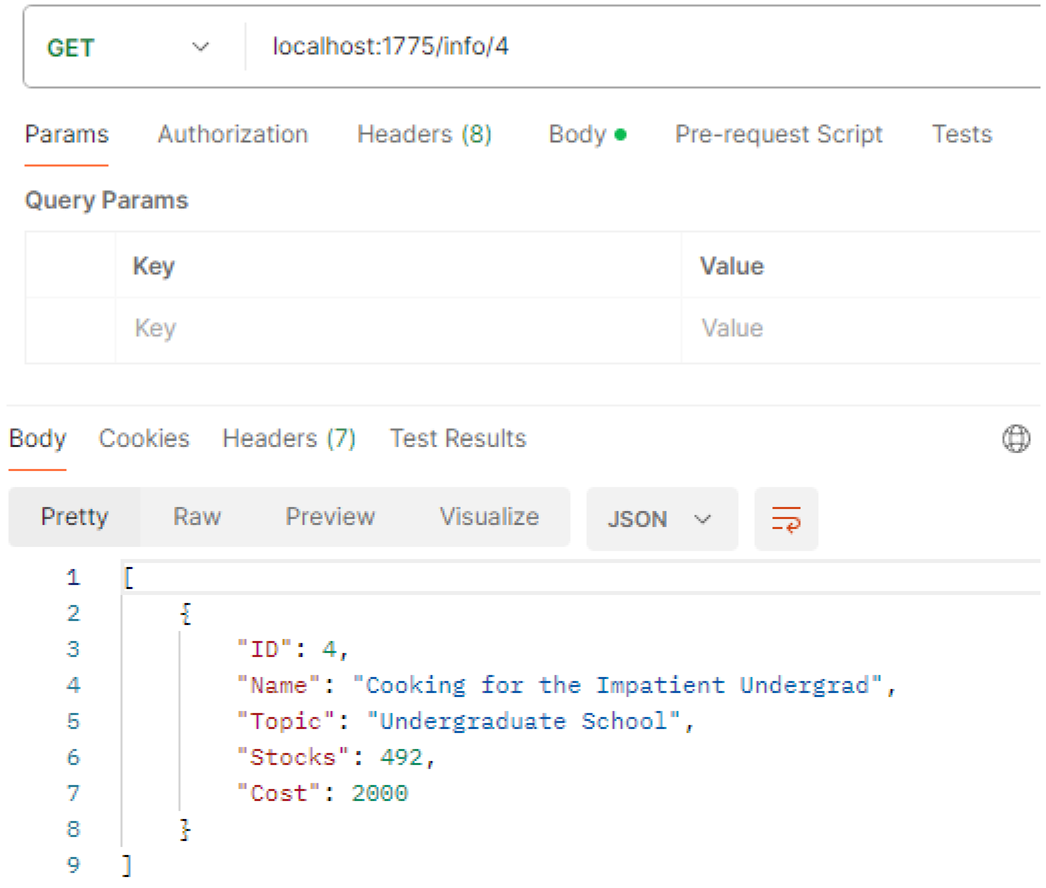
And for the second request:


GET http://localhost:1775/search/Distributed%20Systems

Status: 200 OK Time: 24 ms

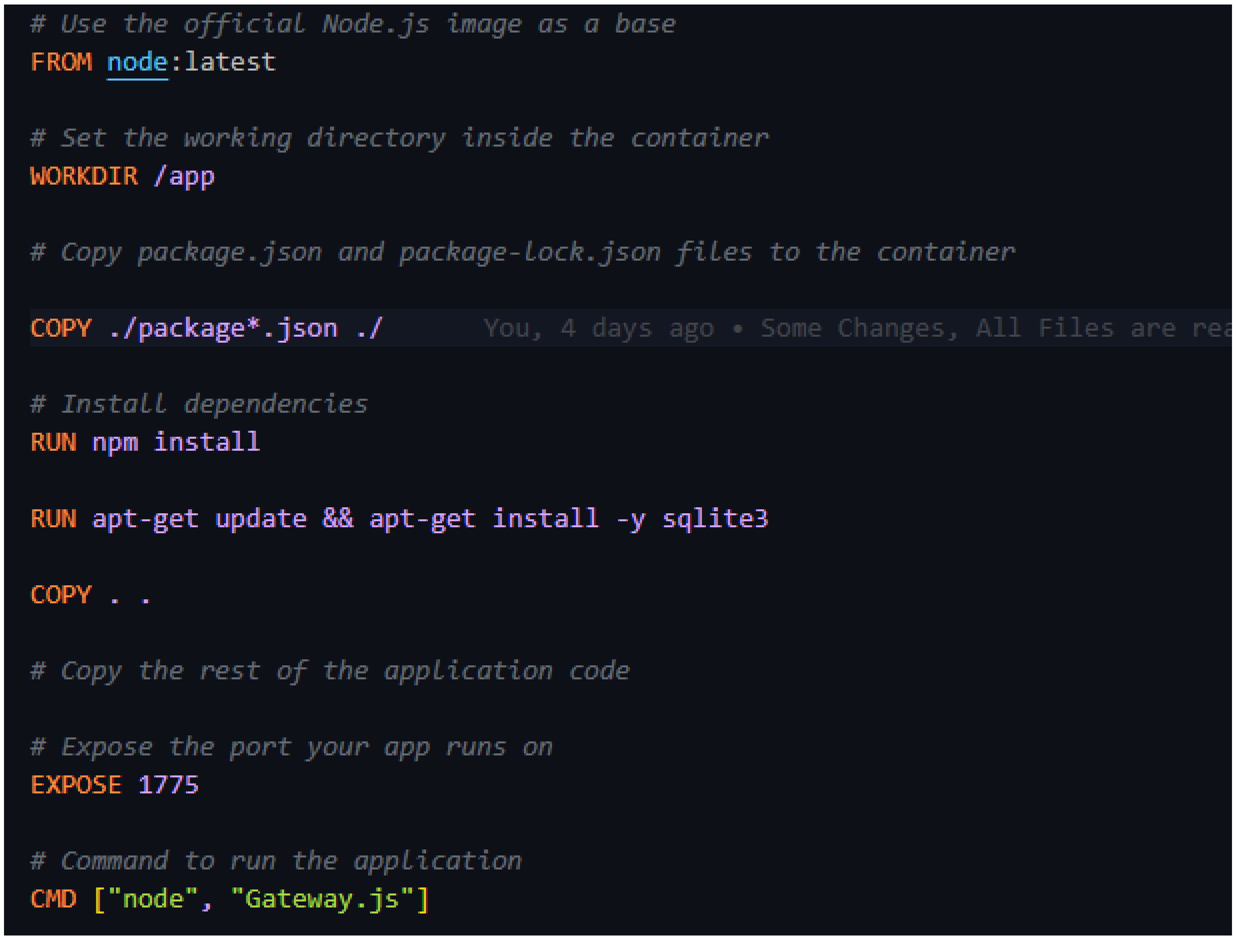
```
[
    {
        "ID": 1,
        "Name": "How to get a good grade in DOS in 40 minutes a day",
        "Topic": "Distributed Systems",
        "Stocks": 499,
        "Cost": 500
    },
    {
        "ID": 2,
        "Name": "RPCs for Noobs",
        "Topic": "Distributed Systems",
        "Stocks": 498,
        "Cost": 1000
    }
]
```

This request we use it to get a specific information about specific book using the id of the book:



```
[
    {
        "ID": 4,
        "Name": "Cooking for the Impatient Undergrad",
        "Topic": "Undergraduate School",
        "Stocks": 492,
        "Cost": 2000
    }
]
```

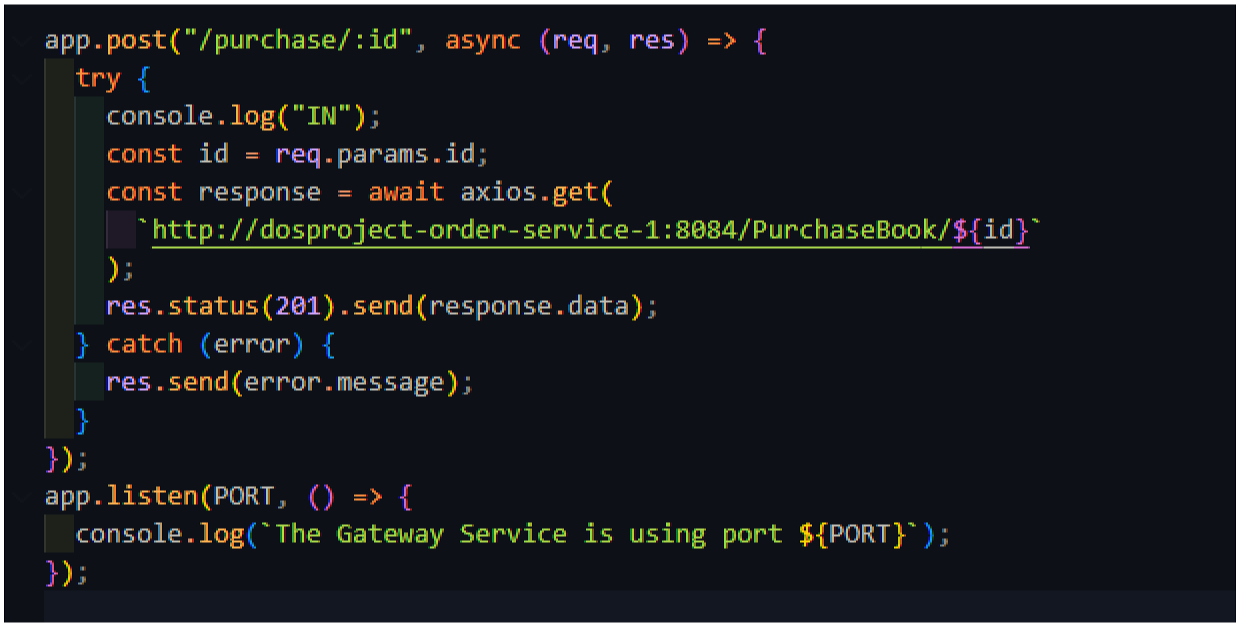
The Dockerfile for the Gateway service:



```
# Use the official Node.js image as a base
FROM node:latest

# Set the working directory inside the container
WORKDIR /app

# Copy package.json and package-lock.json files to the container

COPY ./package*.json ./      You, 4 days ago • Some Changes, All Files are rea

# Install dependencies
RUN npm install

RUN apt-get update && apt-get install -y sqlite3

COPY . .

# Copy the rest of the application code

# Expose the port your app runs on
EXPOSE 1775

# Command to run the application
CMD ["node", "Gateway.js"]
```

We create an image which is using Nodejs as environment to run the code,

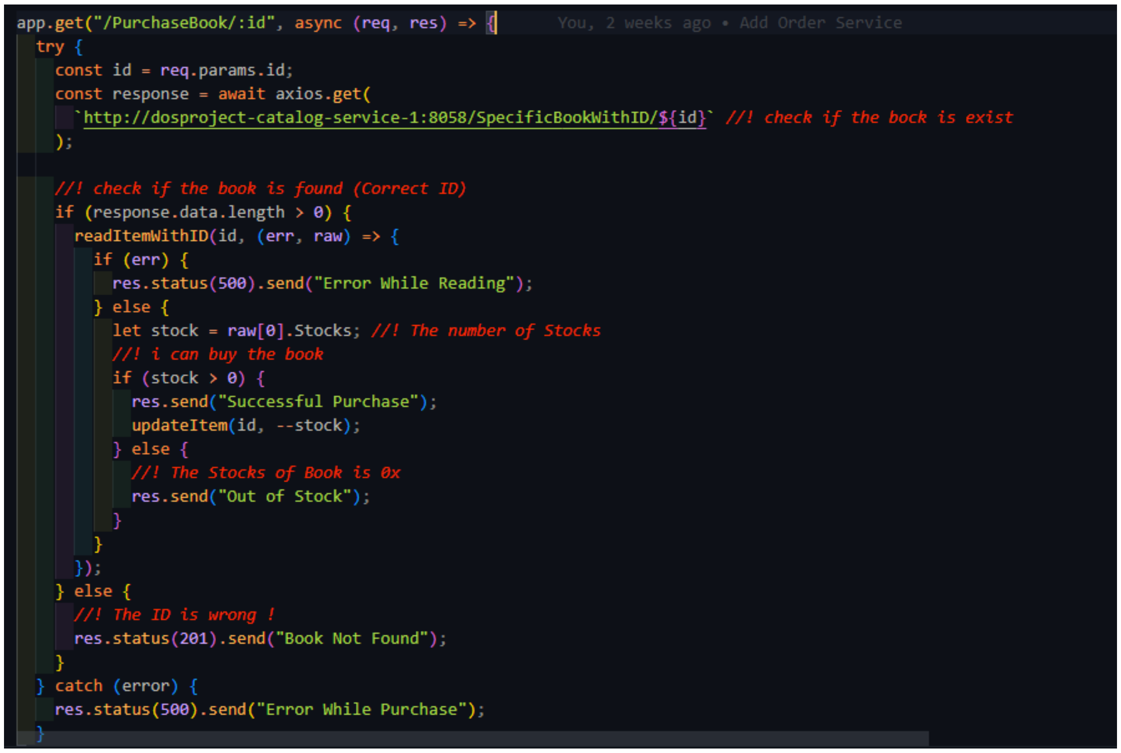
Upon navigating to the designated 'app' directory housing all requisite files, we initiate the process by copying over the package.json files essential for downloading dependencies. Additionally, we fetch the SQLite database necessary for storing our data. Subsequently, we modify the port configuration to utilize port 1775, aligning with the port utilized within our application. Finally, upon image execution, we initiate the application, ensuring all components are in place for seamless operation.



```javascript
app.post("/purchase/:id", async (req, res) => {
  try {
    console.log("IN");
    const id = req.params.id;
    const response = await axios.get(
      `http://dosproject-order-service-1:8084/PurchaseBook/${id}`
    );
    res.status(201).send(response.data);
  } catch (error) {
    res.send(error.message);
  }
});
app.listen(PORT, () => {
  console.log(`The Gateway Service is using port ${PORT}`);
});
```

This is the purchase request, that we handle it in order service, we will explain it in next section.

Order Service



```javascript
app.get("/PurchaseBook/:id", async (req, res) => {      You, 2 weeks ago • Add Order Service
  try {
    const id = req.params.id;
    const response = await axios.get(
      `http://dosproject-catalog-service-1:8058/SpecificBookWithID/${id}` //! check if the bock is exist
    );

    //! check if the book is found (Correct ID)
    if (response.data.length > 0) {
      readItemWithID(id, (err, raw) => {
        if (err) {
          res.status(500).send("Error While Reading");
        } else {
          let stock = raw[0].Stocks; //! The number of Stocks
          //! i can buy the book
          if (stock > 0) {
            res.send("Successful Purchase");
            updateItem(id, --stock);
          } else {
            //! The Stocks of Book is 0x
            res.send("Out of Stock");
          }
        }
      });
    } else {
      //! The ID is wrong !
      res.status(201).send("Book Not Found");
    }
  } catch (error) {
    res.status(500).send("Error While Purchase");
  }
```

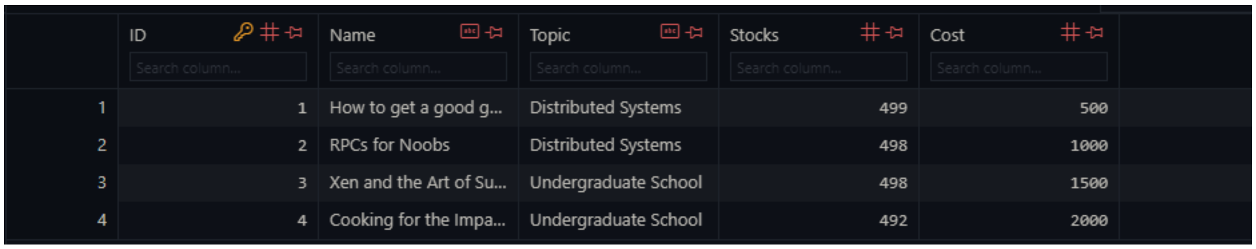
The route /PurchaseBook/:id is defined to handle GET requests aimed at purchasing a book by its ID. This design choice implies that purchasing a book is a simple operation, potentially accessible directly from a browser or a simple HTTP client, without the need for a body in the request.

Communication with Catalog Service: Before processing the purchase, the service uses Axios to send a GET request to the Catalog Service (http://dosproject-catalog-service-1:8058/SpecificBookWithID/${id}). This request is to verify the existence of the book and to check its stock availability.

If the book exists and has a stock greater than 0, the service responds to the client with "Successful Purchase" and decreases the stock by 1 using updateItem. If the book is out of stock, it informs the client accordingly

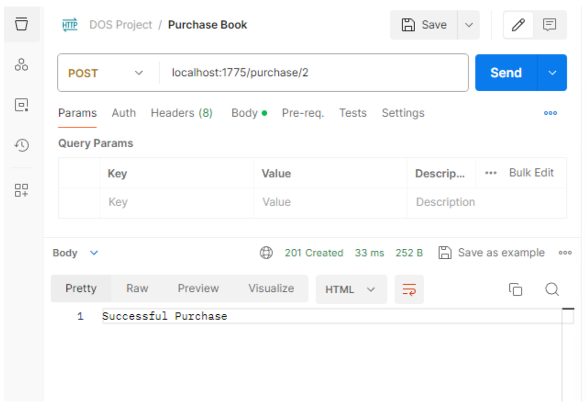
For example, note that the stocks of book number 2 is 498, after purchase request, will be 497:

Before:



	ID	Name	Topic	Stocks	Cost
1	1	How to get a good g...	Distributed Systems	499	500
2	2	RPCs for Noobs	Distributed Systems	498	1000
3	3	Xen and the Art of Su...	Undergraduate School	498	1500
4	4	Cooking for the Impa...	Undergraduate School	492	2000

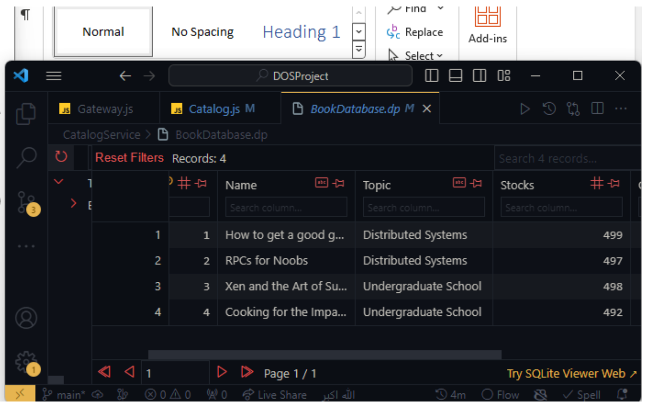
After:



POST localhost:1775/purchase/2

Body 201 Created 33 ms 252 B

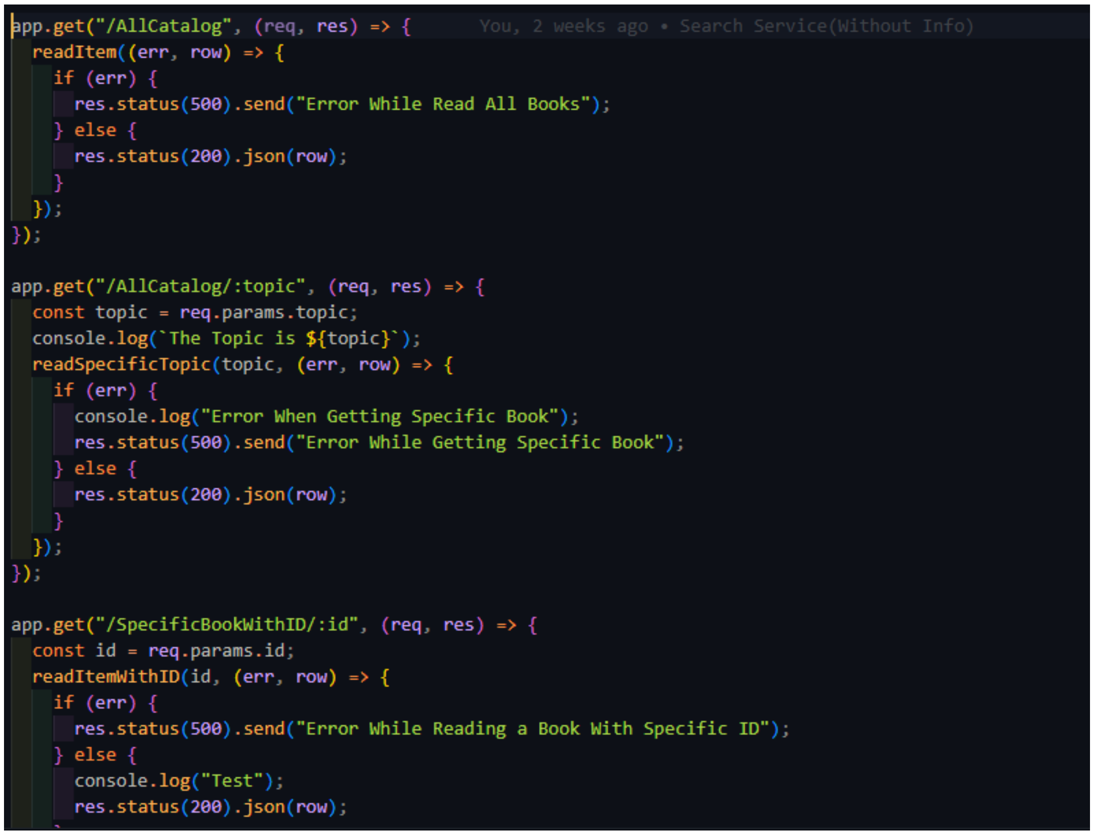
```
Successful Purchase
```



Records: 4

		Name	Topic	Stocks
1	1	How to get a good g...	Distributed Systems	499
2	2	RPCs for Noobs	Distributed Systems	497
3	3	Xen and the Art of Su...	Undergraduate School	498
4	4	Cooking for the Impa...	Undergraduate School	492

Catalog Service



```
app.get("/AllCatalog", (req, res) => {        You, 2 weeks ago • Search Service(Without Info)
  readItem((err, row) => {
    if (err) {
      res.status(500).send("Error While Read All Books");
    } else {
      res.status(200).json(row);
    }
  });
});

app.get("/AllCatalog/:topic", (req, res) => {
  const topic = req.params.topic;
  console.log(`The Topic is ${topic}`);
  readSpecificTopic(topic, (err, row) => {
    if (err) {
      console.log("Error When Getting Specific Book");
      res.status(500).send("Error While Getting Specific Book");
    } else {
      res.status(200).json(row);
    }
  });
});

app.get("/SpecificBookWithID/:id", (req, res) => {
  const id = req.params.id;
  readItemWithID(id, (err, row) => {
    if (err) {
      res.status(500).send("Error While Reading a Book With Specific ID");
    } else {
      console.log("Test");
      res.status(200).json(row);
```

This code is used in a Catalog file as part of a microservices architecture, specifically within the Catalog Service of an online bookstore. The Catalog Service is responsible for managing the information about the books available in the store, such as titles, authors, prices, stock levels, and topics.

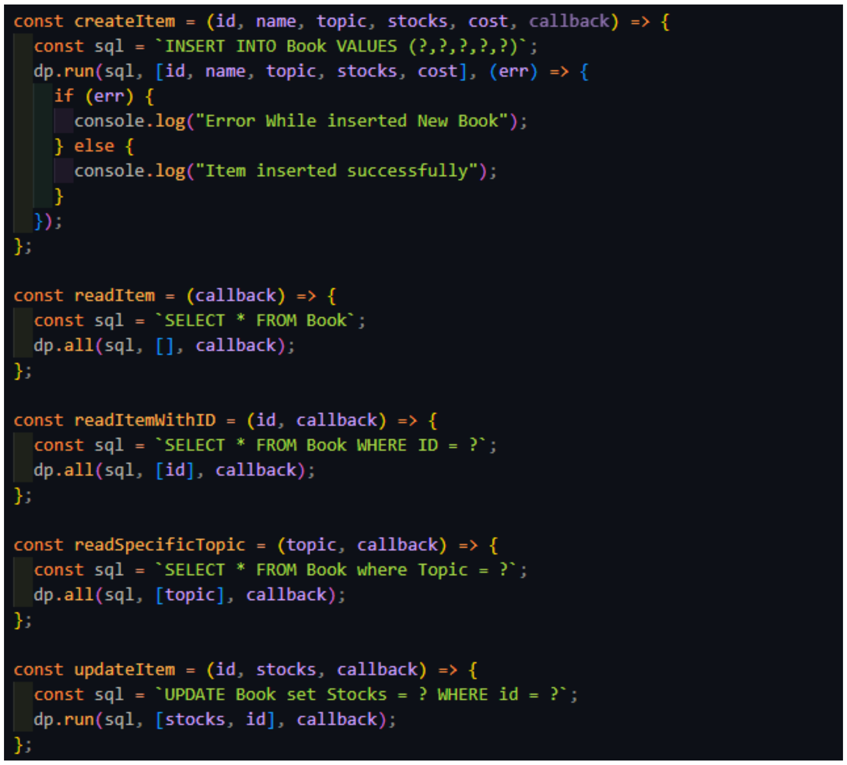
Simplicity and Performance: Express is a lightweight framework that simplifies web server creation in Node.js, It's used here to handle HTTP requests efficiently.

The code makes use of readItem, readSpecificTopic, and readItemWithID functions imported from a CRUD module.

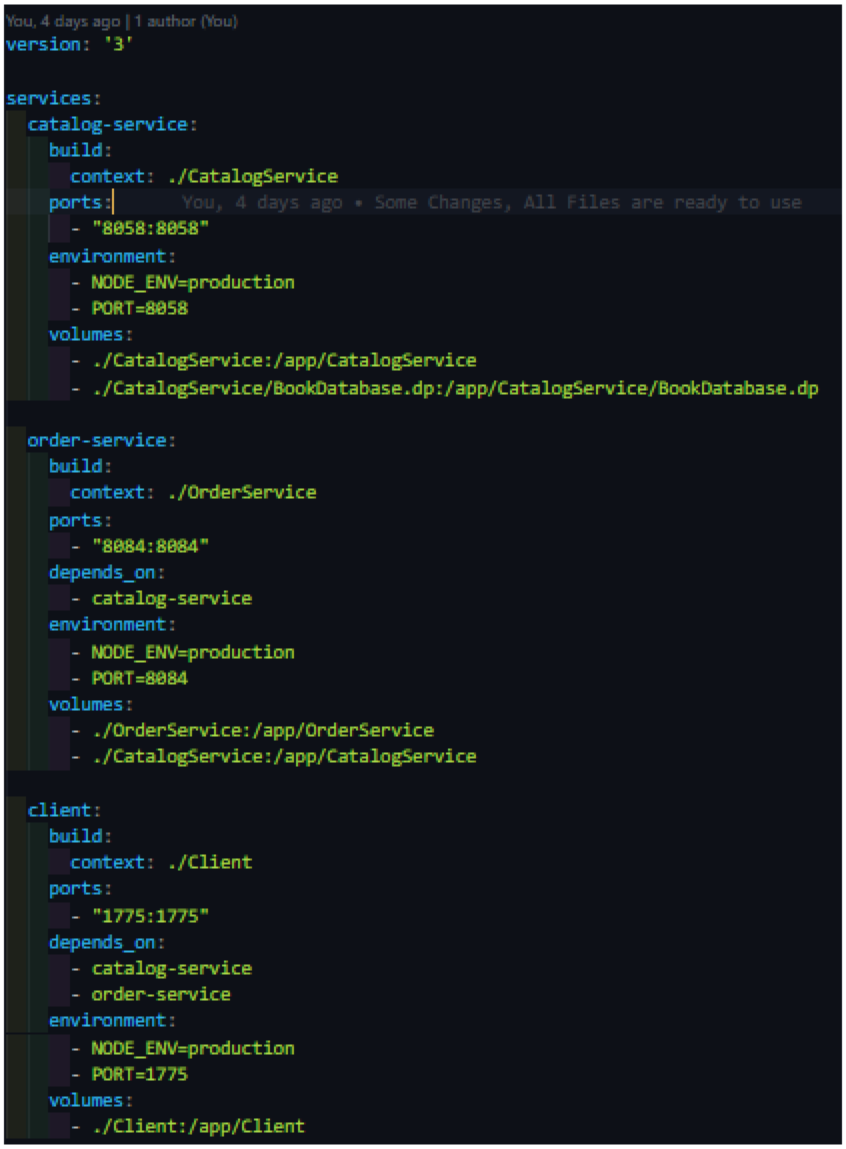
HTTP Status Codes: The service uses appropriate HTTP status codes (200 for success, 500 for server errors) to communicate the outcome of requests to clients, adhering to RESTful API best practices.

Port Listening: The service listens on port 8058, which is specified for the CatalogService. This port configuration allows other services within the ecosystem (like the Gateway Service) to communicate with the Catalog Service.

This is the CRUD.js file:



```
const createItem = (id, name, topic, stocks, cost, callback) => {
  const sql = `INSERT INTO Book VALUES (?,?,?,?,?)`;
  dp.run(sql, [id, name, topic, stocks, cost], (err) => {
    if (err) {
      console.log("Error While inserted New Book");
    } else {
      console.log("Item inserted successfully");
    }
  });
};

const readItem = (callback) => {
  const sql = `SELECT * FROM Book`;
  dp.all(sql, [], callback);
};

const readItemWithID = (id, callback) => {
  const sql = `SELECT * FROM Book WHERE ID = ?`;
  dp.all(sql, [id], callback);
};

const readSpecificTopic = (topic, callback) => {
  const sql = `SELECT * FROM Book where Topic = ?`;
  dp.all(sql, [topic], callback);
};

const updateItem = (id, stocks, callback) => {
  const sql = `UPDATE Book set Stocks = ? WHERE id = ?`;
  dp.run(sql, [stocks, id], callback);
};
```



```
You, 4 days ago | 1 author (You)
version: '3'

services:
  catalog-service:
    build:
      context: ./CatalogService
    ports:        You, 4 days ago • Some Changes, All Files are ready to use
      - "8058:8058"
    environment:
      - NODE_ENV=production
      - PORT=8058
    volumes:
      - ./CatalogService:/app/CatalogService
      - ./CatalogService/BookDatabase.dp:/app/CatalogService/BookDatabase.dp

  order-service:
    build:
      context: ./OrderService
    ports:
      - "8084:8084"
    depends_on:
      - catalog-service
    environment:
      - NODE_ENV=production
      - PORT=8084
    volumes:
      - ./OrderService:/app/OrderService
      - ./CatalogService:/app/CatalogService

  client:
    build:
      context: ./Client
    ports:
      - "1775:1775"
    depends_on:
      - catalog-service
      - order-service
    environment:
      - NODE_ENV=production
      - PORT=1775
    volumes:
      - ./Client:/app/Client
```

The provided docker-compose.yml file is used to define and run multi-container Docker applications. With Docker Compose, use a yml file to configure your application's services, networks, and volumes. Then, with a single command, can create and start all the services specified in your configuration.

Conclusion

In conclusion, our project demonstrates a meticulous and comprehensive approach to building and deploying a microservices architecture using Docker containers. Leveraging technologies such as NodeJS, Express, Axios, and SQLite, we've constructed a scalable and efficient system capable of handling various requests and data transfers between services. By adopting Docker and docker-compose, we've achieved seamless orchestration and deployment of our services, ensuring consistency and reliability across different environments. Our emphasis on error handling, portability, and efficient resource utilization underscores our commitment to delivering robust and resilient solutions. Through this project, we've not only addressed the challenges of distributed systems but also laid a solid foundation for future enhancements and scalability.