An-Najah National University

Department of Computer Engineering

Distributed Operation Systems

Project

Part #1

Submitted to:

Dr. Samer Arande

Created By:

Wasan Awwad (12042150)

Areen Halabi (11923535)

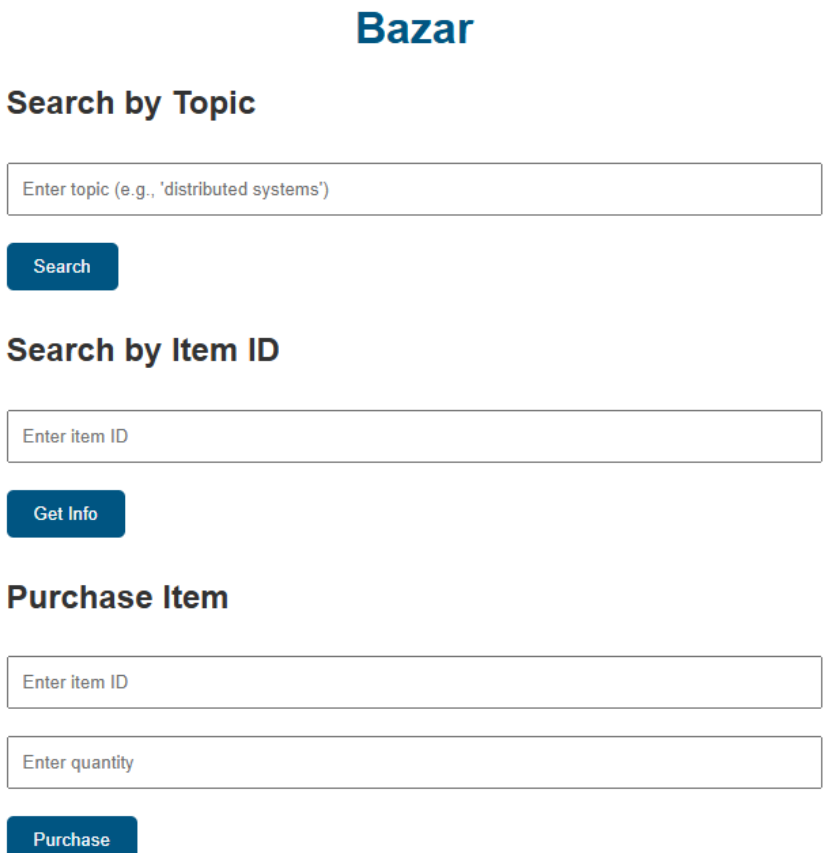
https://github.com/AreenHalabe/Book_Store.git

Introduction

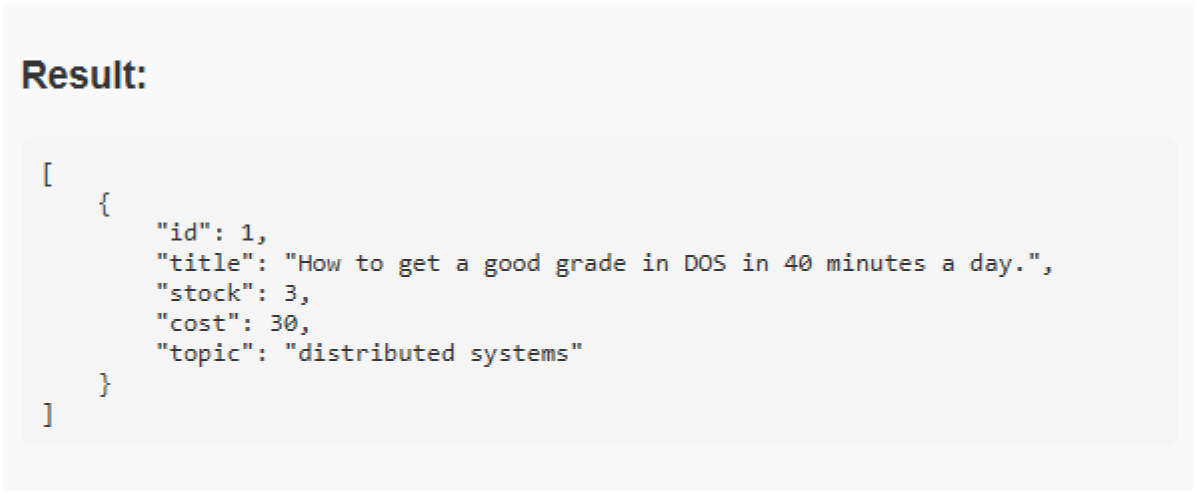
This project is to create and design Bazar.com, the world's smallest bookstore, which lists only four books. The system is built with two-tier web architecture and has microservices-based design in order to achieve a modular and scalable system. With the front-end and back-end services isolated, the design achieves better performance, flexibility, and maintainability. The project focuses on giving an uninterrupted user experience and benefiting from efficient and well-organized design concepts.

Project Components

- **Front-End:**



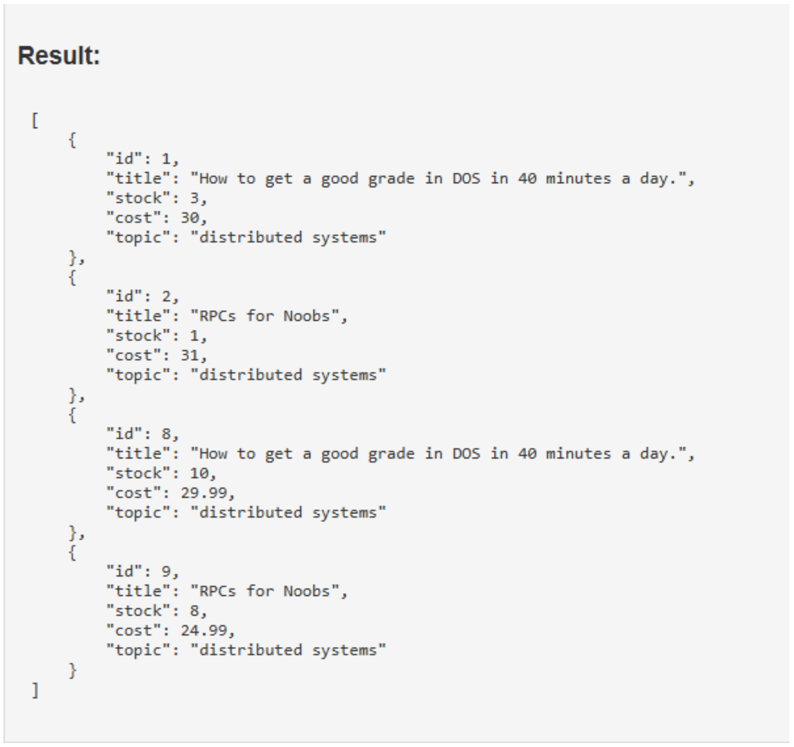
search with id = 1



Result:

```
[
    {
        "id": 1,
        "title": "How to get a good grade in DOS in 40 minutes a day.",
        "stock": 3,
        "cost": 30,
        "topic": "distributed systems"
    }
]
```

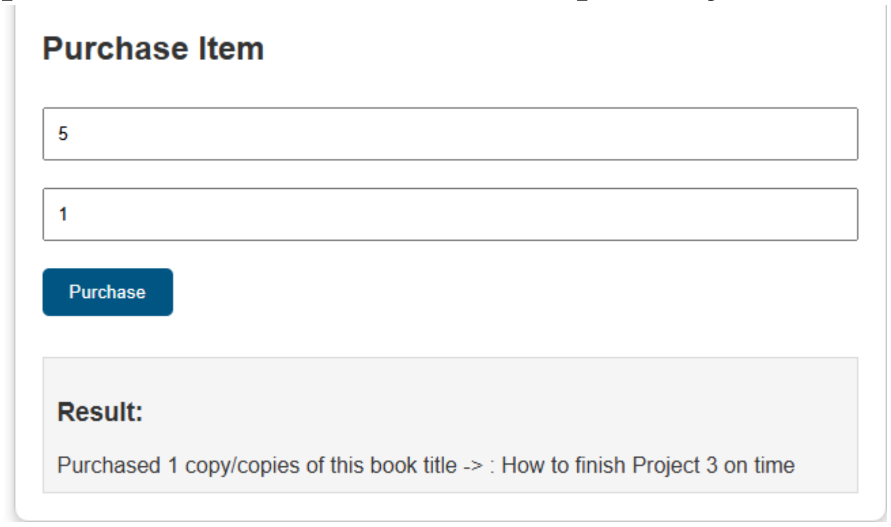
search with topic distributed systems



Result:

```
[
    {
        "id": 1,
        "title": "How to get a good grade in DOS in 40 minutes a day.",
        "stock": 3,
        "cost": 30,
        "topic": "distributed systems"
    },
    {
        "id": 2,
        "title": "RPCs for Noobs",
        "stock": 1,
        "cost": 31,
        "topic": "distributed systems"
    },
    {
        "id": 8,
        "title": "How to get a good grade in DOS in 40 minutes a day.",
        "stock": 10,
        "cost": 29.99,
        "topic": "distributed systems"
    },
    {
        "id": 9,
        "title": "RPCs for Noobs",
        "stock": 8,
        "cost": 24.99,
        "topic": "distributed systems"
    }
]
```

purchase book with id 5 and quantity 1



Result:

Purchased 1 copy/copies of this book title -> : How to finish Project 3 on time

- **Back-End:**

  Comprises two components:

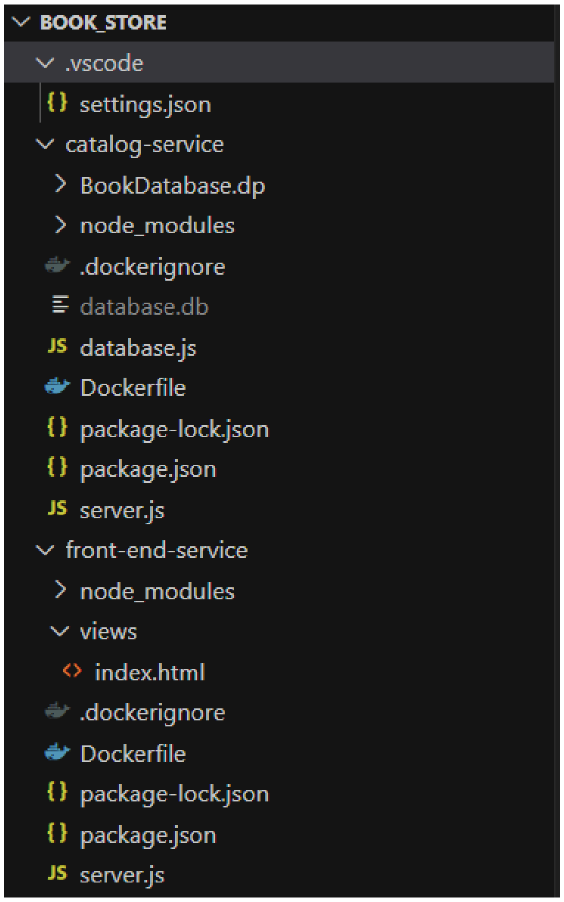
  1. Catalog Server: Manages the book catalog, with information on stock number, price, and subject for each book.

  2. Order Server: Keeps track of all incoming orders, verifying stock availability with the Catalog Server before processing sales

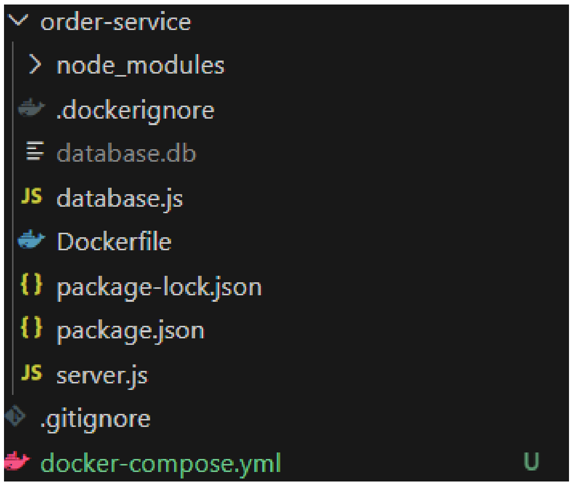
RESTful Interface

• The system must include HTTP REST interfaces to all operations of the components (search, info, purchase).
• REST endpoints must be written in a consumable and easy-to-integrate manner, exchanging data in the form of JSON.
• The system must process multiple requests simultaneously effectively, utilizing the inherent concurrency capabilities of the web framework being used.
The application deployable across multiple machines in a distributed manner, with Docker containers for separation and ease of deployment.

Project Hierarchy



```
BOOK_STORE
  .vscode
    settings.json
  catalog-service
    BookDatabase.dp
    node_modules
    .dockerignore
    database.db
    database.js
    Dockerfile
    package-lock.json
    package.json
    server.js
  front-end-service
    node_modules
    views
      index.html
    .dockerignore
    Dockerfile
    package-lock.json
    package.json
    server.js
```



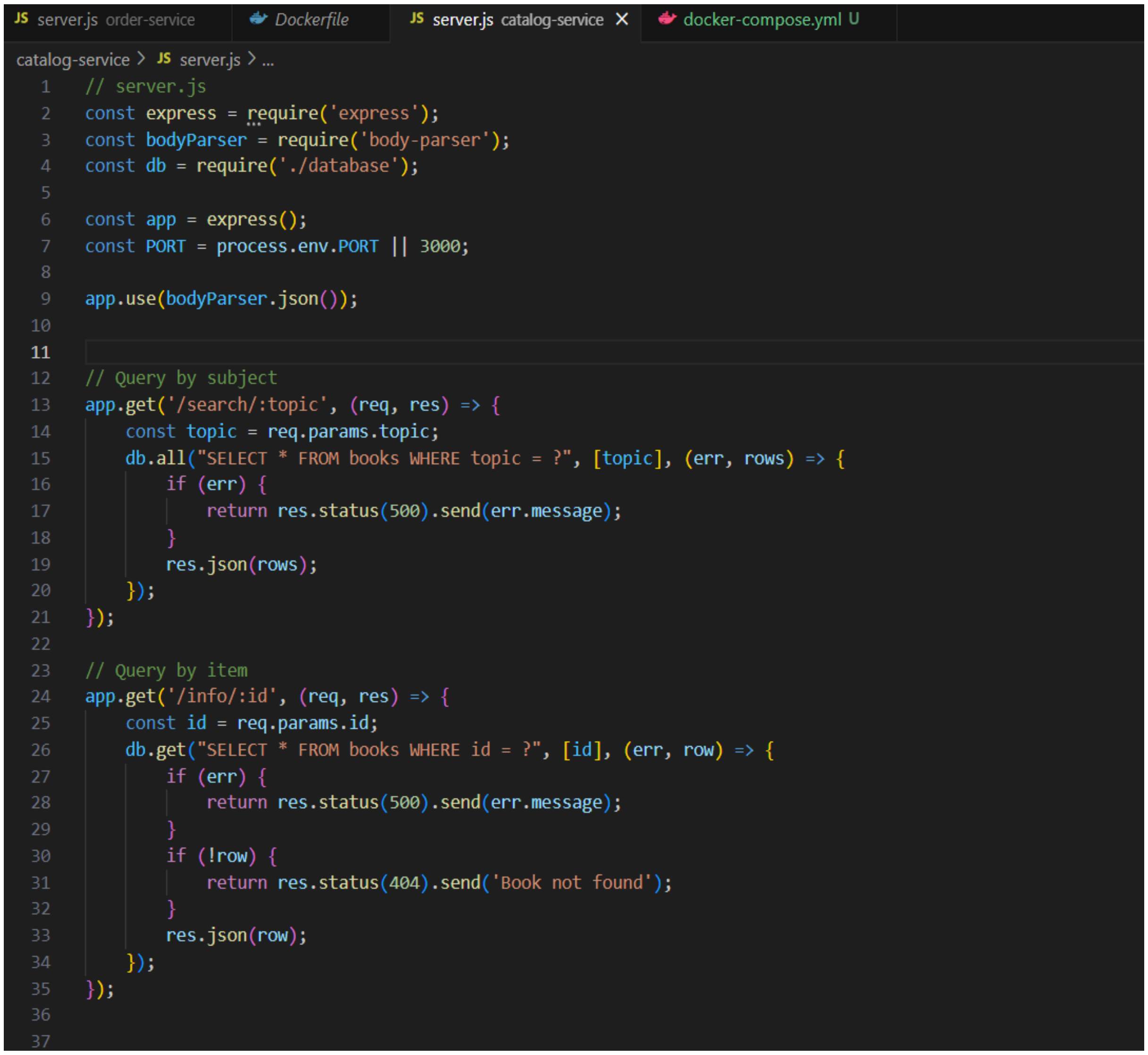
```
order-service
  node_modules
  .dockerignore
  database.db
  database.js
  Dockerfile
  package-lock.json
  package.json
  server.js
.gitignore
docker-compose.yml          U
```

For each service, we designed a particular file with the service, all the supplier files, and the Dockerfile. This enables us to run everything smoothly with the assistance of Docker Containers.
We utilize the docker-compose file to enable the running of all the Docker files concurrently. It helps us determine the relationship between the services as well as assign the ports upon which they are to be run, facilitating effective and simultaneous deployment of our applications.

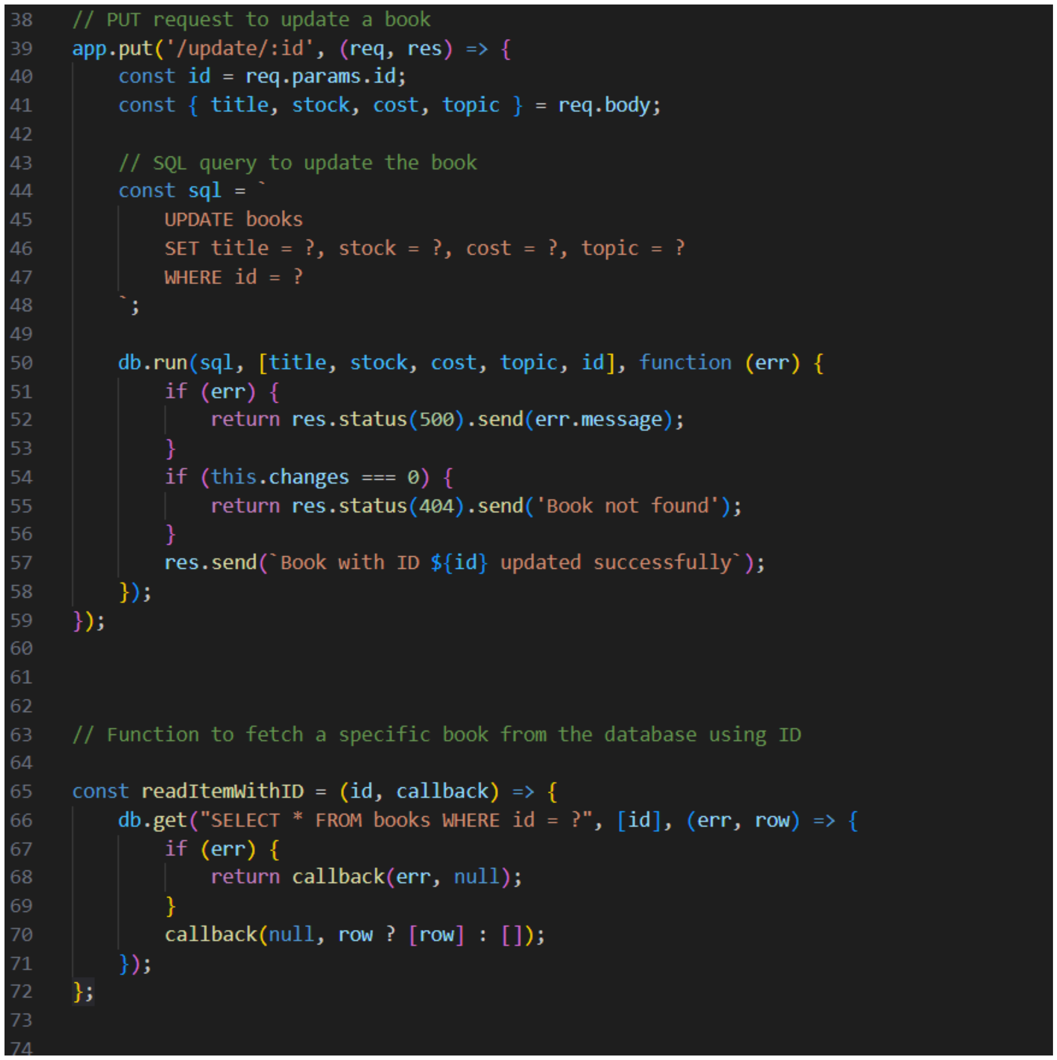
Catalog Service

We utilize NodeJS to carry out the services, utilizing the express library to deal with various request methods such as POST and GET. We also utilize axios for request transfer between services. Our services are on port number 3000, which we selected based on availability and system compatibility.
The initial search service in our setup is a bridge that successfully directs incoming requests to our catalog service on the Docker network. By utilizing the container name instead of localhost, we facilitate seamless communication between services. If the catalog service successfully processes the request with no issues, a response with a status code of 200 is returned with the correct data. In case of any errors during processing, our system responds instantly with a status code of 500, which signifies a problem needing attention. This vigorous method guarantees smooth functionality and trustworthy data transmission in our service architecture.

The code that implements it is below:



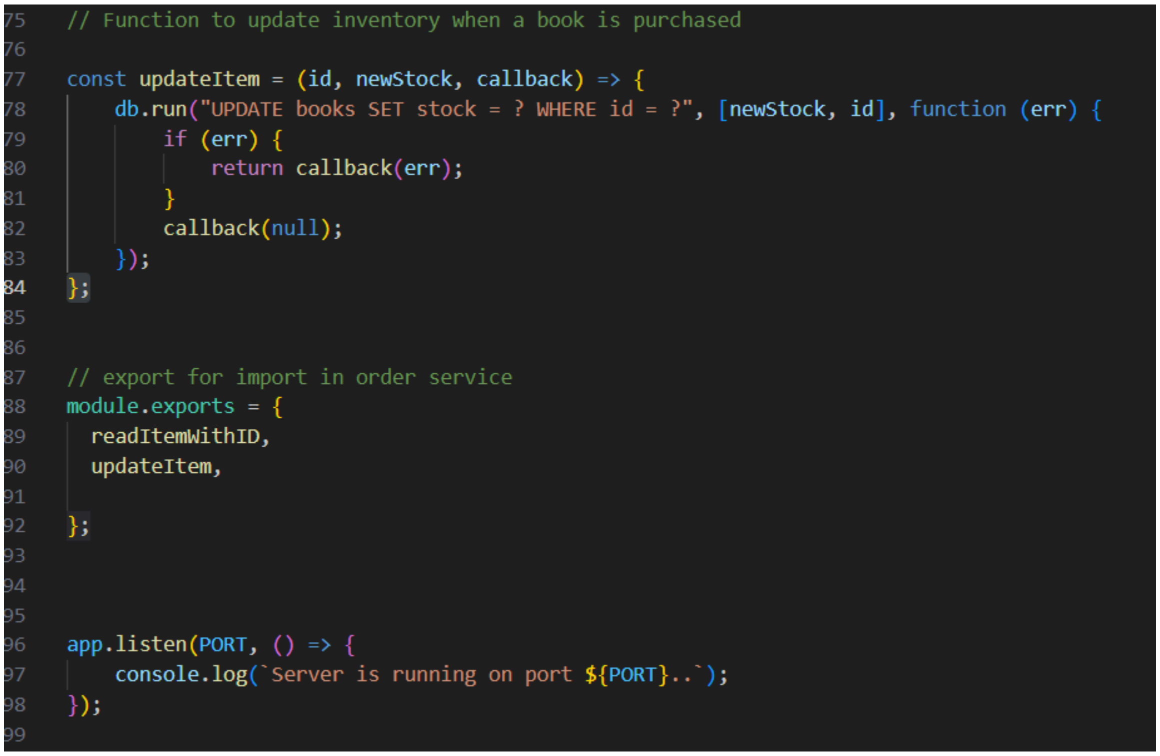
```javascript
// server.js
const express = require('express');
const bodyParser = require('body-parser');
const db = require('./database');

const app = express();
const PORT = process.env.PORT || 3000;

app.use(bodyParser.json());


// Query by subject
app.get('/search/:topic', (req, res) => {
    const topic = req.params.topic;
    db.all("SELECT * FROM books WHERE topic = ?", [topic], (err, rows) => {
        if (err) {
            return res.status(500).send(err.message);
        }
        res.json(rows);
    });
});

// Query by item
app.get('/info/:id', (req, res) => {
    const id = req.params.id;
    db.get("SELECT * FROM books WHERE id = ?", [id], (err, row) => {
        if (err) {
            return res.status(500).send(err.message);
        }
        if (!row) {
            return res.status(404).send('Book not found');
        }
        res.json(row);
    });
});
```



```
// PUT request to update a book
app.put('/update/:id', (req, res) => {
    const id = req.params.id;
    const { title, stock, cost, topic } = req.body;

    // SQL query to update the book
    const sql = `
        UPDATE books
        SET title = ?, stock = ?, cost = ?, topic = ?
        WHERE id = ?
    `;

    db.run(sql, [title, stock, cost, topic, id], function (err) {
        if (err) {
            return res.status(500).send(err.message);
        }
        if (this.changes === 0) {
            return res.status(404).send('Book not found');
        }
        res.send(`Book with ID ${id} updated successfully`);
    });
});



// Function to fetch a specific book from the database using ID

const readItemWithID = (id, callback) => {
    db.get("SELECT * FROM books WHERE id = ?", [id], (err, row) => {
        if (err) {
            return callback(err, null);
        }
        callback(null, row ? [row] : []);
    });
};

```

The above code is used in a Catalog file in the context of a microservices architecture, in the Catalog Service of an online bookstore. Catalog Service is responsible for managing details of the books present in the store, for instance, names, authors, prices, stock, and themes.

Simplicity and Performance: Express is a light-weight framework that makes it easy to develop web servers in Node.js, It's used here to handle HTTP requests efficiently

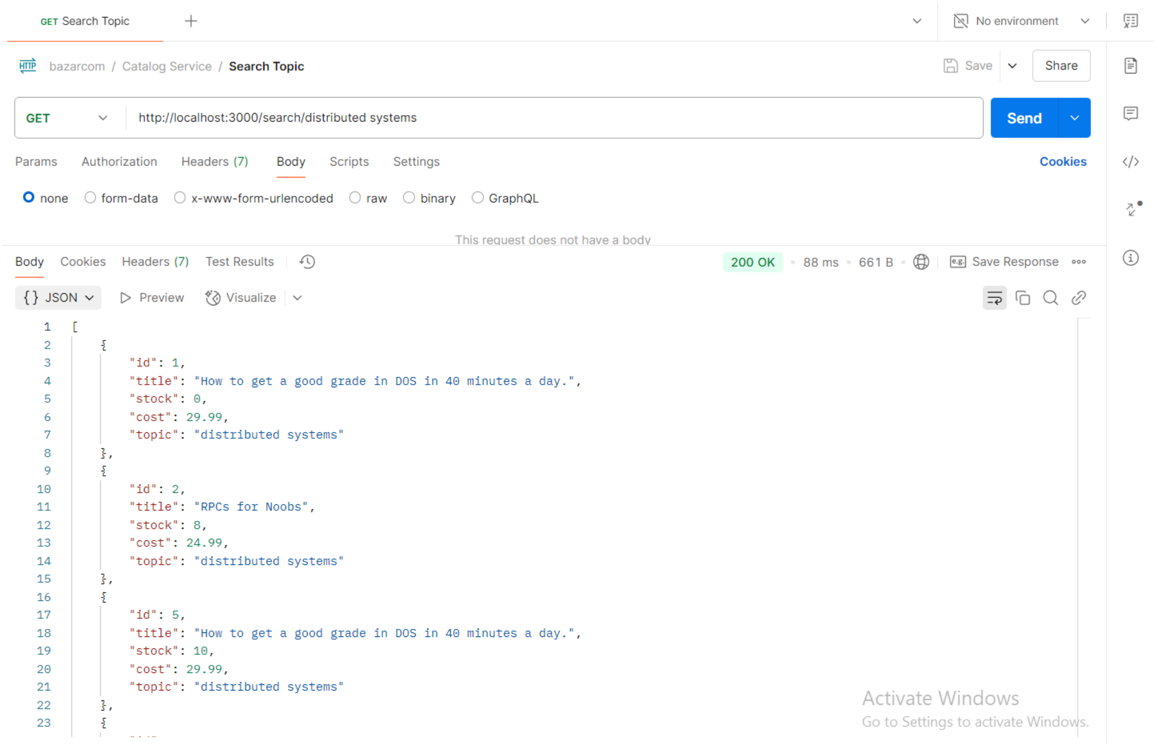


```
// Function to update inventory when a book is purchased

const updateItem = (id, newStock, callback) => {
    db.run("UPDATE books SET stock = ? WHERE id = ?", [newStock, id], function (err) {
        if (err) {
            return callback(err);
        }
        callback(null);
    });
};


// export for import in order service
module.exports = {
  readItemWithID,
  updateItem,

};



app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}..`);
});

```

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 3000, 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.

The Output



GET Search Topic

bazarcom / Catalog Service / Search Topic

GET http://localhost:3000/search/distributed systems

Params Authorization Headers (7) Body Scripts Settings Cookies

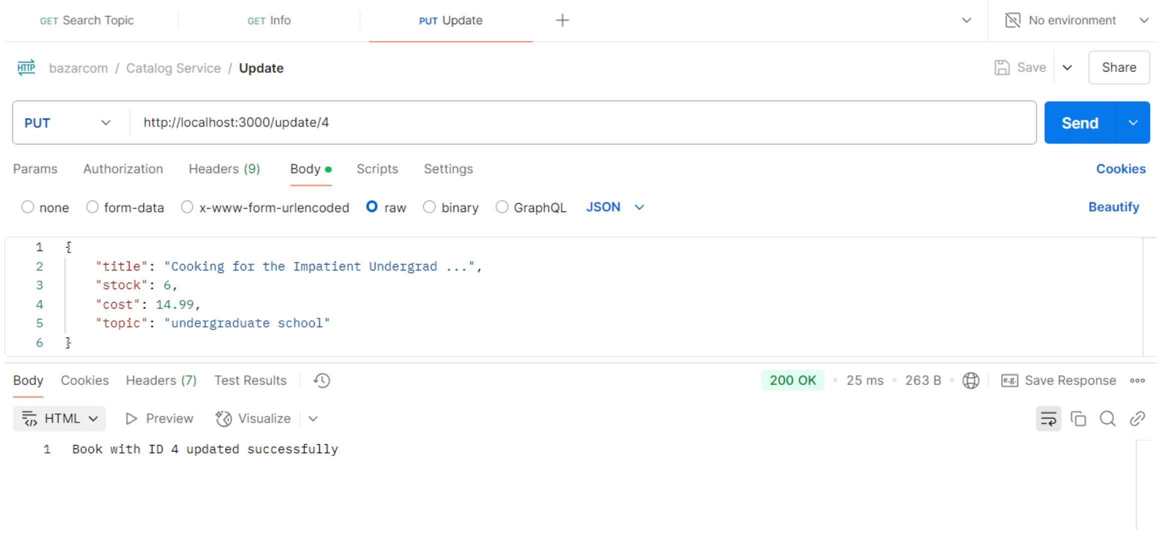
[✓ none] form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

Body Cookies Headers (7) Test Results 200 OK 88 ms 661 B Save Response

```
[
    {
        "id": 1,
        "title": "How to get a good grade in DOS in 40 minutes a day.",
        "stock": 0,
        "cost": 29.99,
        "topic": "distributed systems"
    },
    {
        "id": 2,
        "title": "RPCs for Noobs",
        "stock": 8,
        "cost": 24.99,
        "topic": "distributed systems"
    },
    {
        "id": 5,
        "title": "How to get a good grade in DOS in 40 minutes a day.",
        "stock": 10,
        "cost": 29.99,
        "topic": "distributed systems"
    },
    {
```

And for the second request:



GET Search Topic GET Info PUT Update

bazarcom / Catalog Service / Update

PUT http://localhost:3000/update/4

Params Authorization Headers (9) Body Scripts Settings Cookies

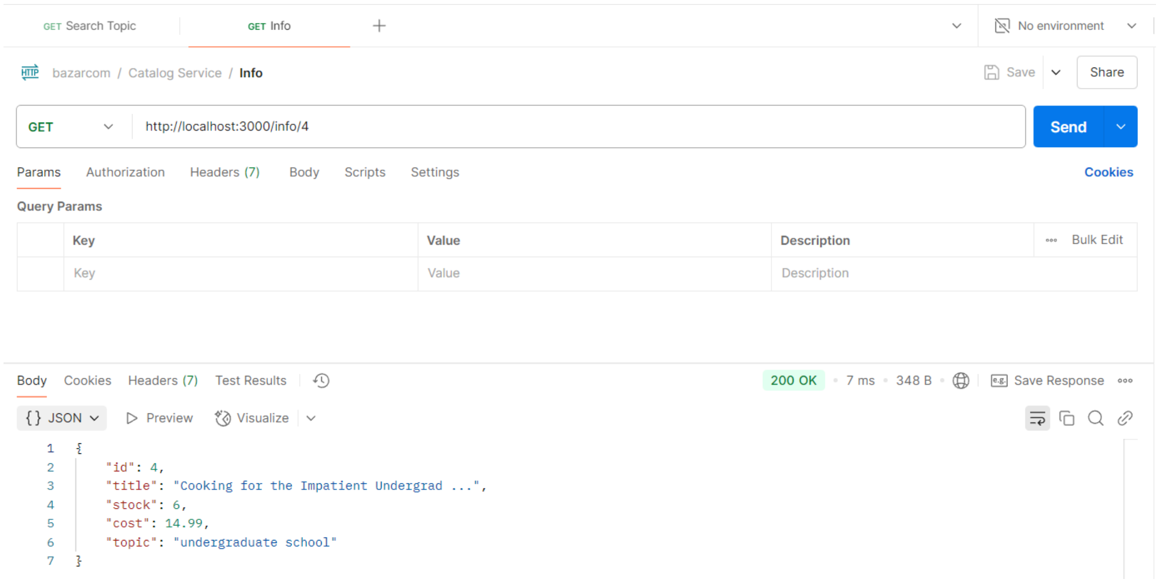
none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON Beautify

```
{
    "title": "Cooking for the Impatient Undergrad ...",
    "stock": 6,
    "cost": 14.99,
    "topic": "undergraduate school"
}
```

Body Cookies Headers (7) Test Results 200 OK 25 ms 263 B Save Response

```
Book with ID 4 updated successfully
```

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

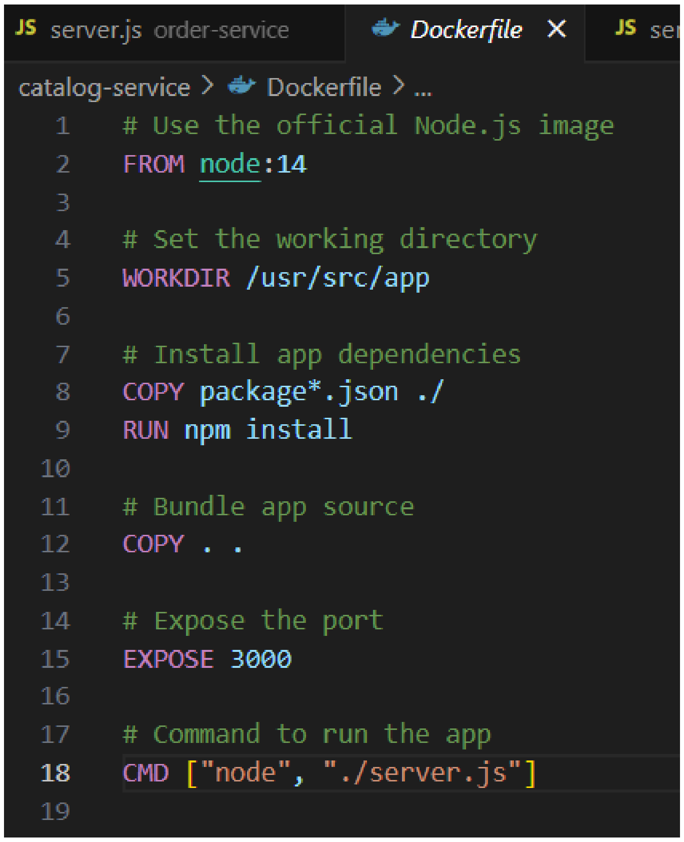


GET http://localhost:3000/info/4

Key	Value	Description
Key	Value	Description

```
{
    "id": 4,
    "title": "Cooking for the Impatient Undergrad ...",
    "stock": 6,
    "cost": 14.99,
    "topic": "undergraduate school"
}
```

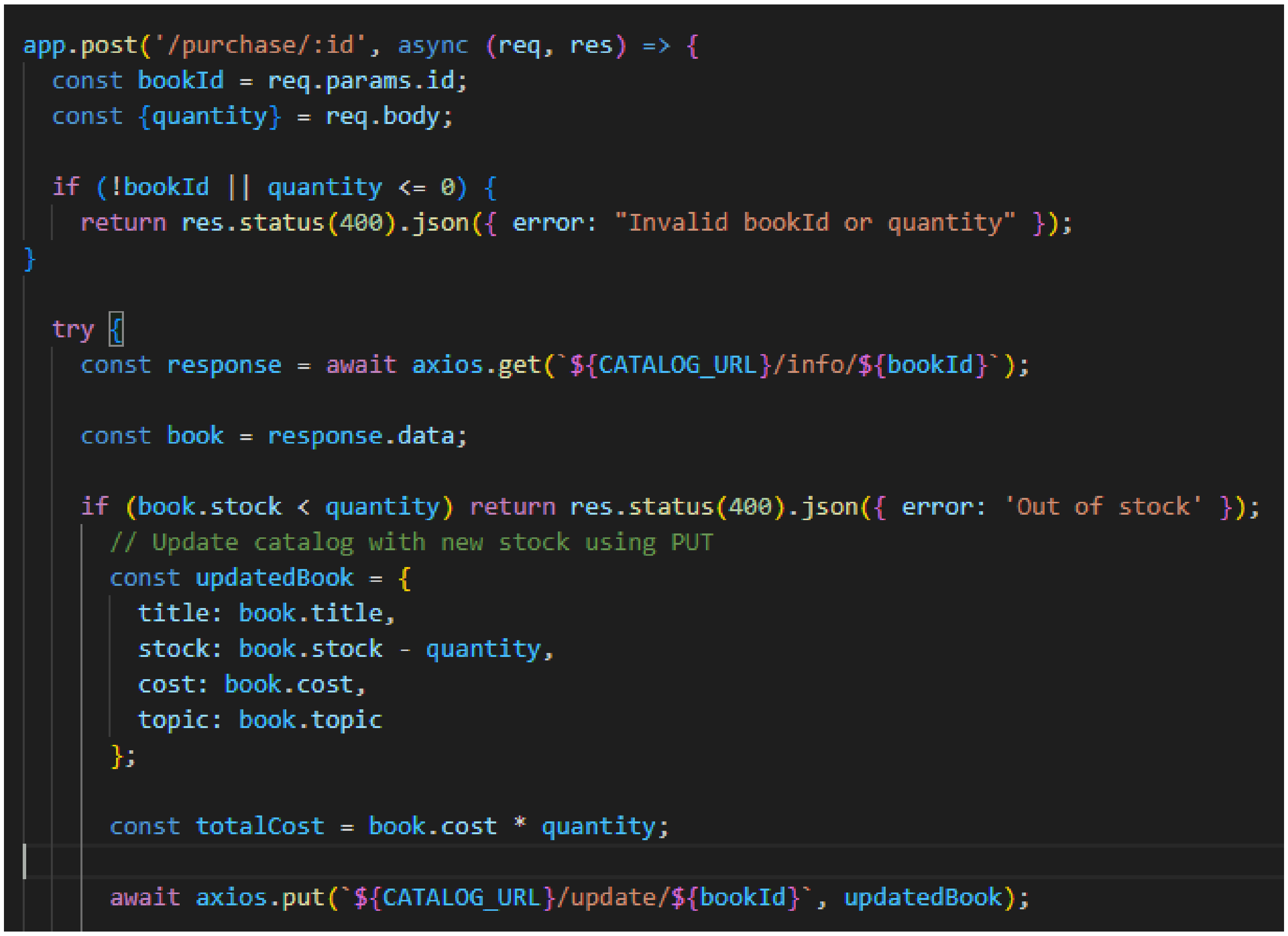
The Dockerfile for the Catalog service:



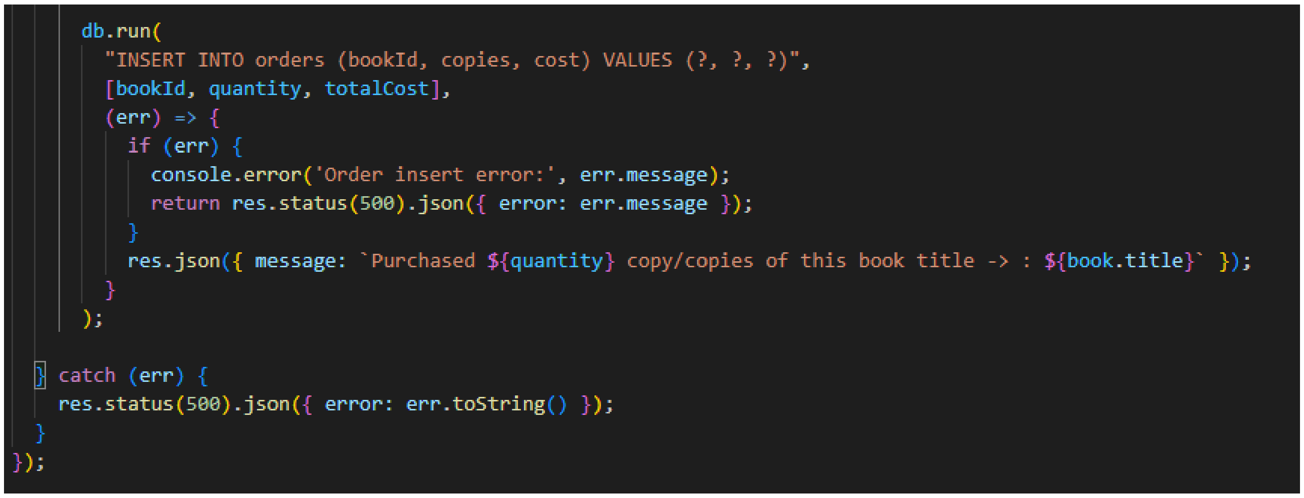
```
# Use the official Node.js image
FROM node:14

# Set the working directory
WORKDIR /usr/src/app

# Install app dependencies
COPY package*.json ./
RUN npm install

# Bundle app source
COPY . .

# Expose the port
EXPOSE 3000

# Command to run the app
CMD ["node", "./server.js"]
```

We set up a scenario which is using Nodejs as environment where to run the code,Upon entering the given 'app' directory with all the files needed, we begin by copying the package.json files used to download the dependencies. We also download the SQLite database needed to hold our data. Then, we switch the port configuration to use port 3000, the same port used in our application. Finally, upon image execution, we begin the application with all pieces set to have it work appropriately.

Order Service

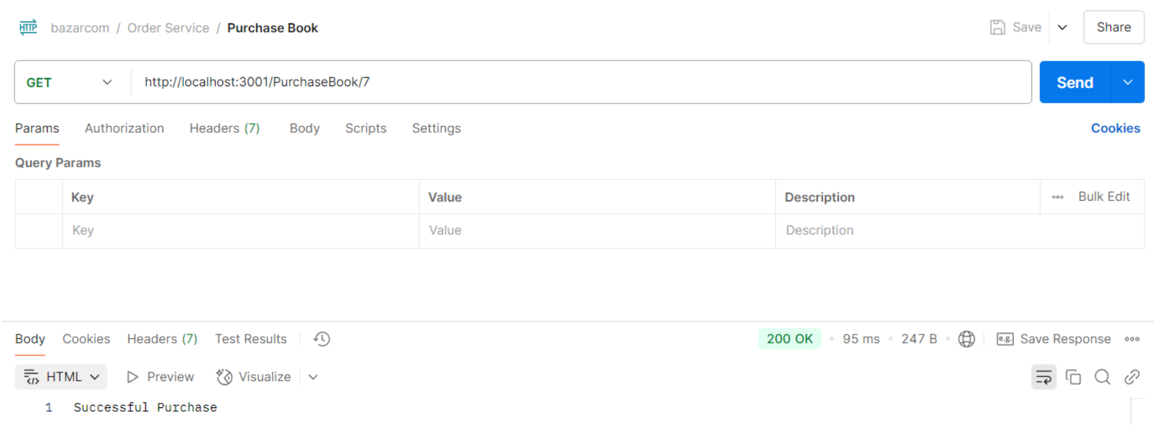


```
app.post('/purchase/:id', async (req, res) => {
  const bookId = req.params.id;
  const {quantity} = req.body;

  if (!bookId || quantity <= 0) {
    return res.status(400).json({ error: "Invalid bookId or quantity" });
}

  try {
    const response = await axios.get(`${CATALOG_URL}/info/${bookId}`);

    const book = response.data;

    if (book.stock < quantity) return res.status(400).json({ error: 'Out of stock' });
      // Update catalog with new stock using PUT
      const updatedBook = {
        title: book.title,
        stock: book.stock - quantity,
        cost: book.cost,
        topic: book.topic
      };

      const totalCost = book.cost * quantity;

      await axios.put(`${CATALOG_URL}/update/${bookId}`, updatedBook);
```



```
      db.run(
        "INSERT INTO orders (bookId, copies, cost) VALUES (?, ?, ?)",
        [bookId, quantity, totalCost],
        (err) => {
          if (err) {
            console.error('Order insert error:', err.message);
            return res.status(500).json({ error: err.message });
          }
          res.json({ message: `Purchased ${quantity} copy/copies of this book title -> : ${book.title}` });
        }
      );

  } catch (err) {
    res.status(500).json({ error: err.toString() });
  }
});
```

The route /PurchaseBook/:id is defined to handle GET requests to purchase a book by ID. This design assumption is that purchasing a book is an inconsequential action, perhaps directly addressable from a browser or a minimal HTTP client, and does not need a body in the request.
Communication with Catalog Service: Before processing the buy, the service uses Axios to issue a GET request to the Catalog Service (http://dosproject-catalog-service-1:8058/SpecificBookWithID/${id}). The request is sent to verify whether the book exists and whether it is available in stock.
If the book exists and stock is greater than 0, the service returns back to the client "Successful Purchase" and decrements the stock by 1 through updateItem. If the book is not in stock, it returns to the client appropriately



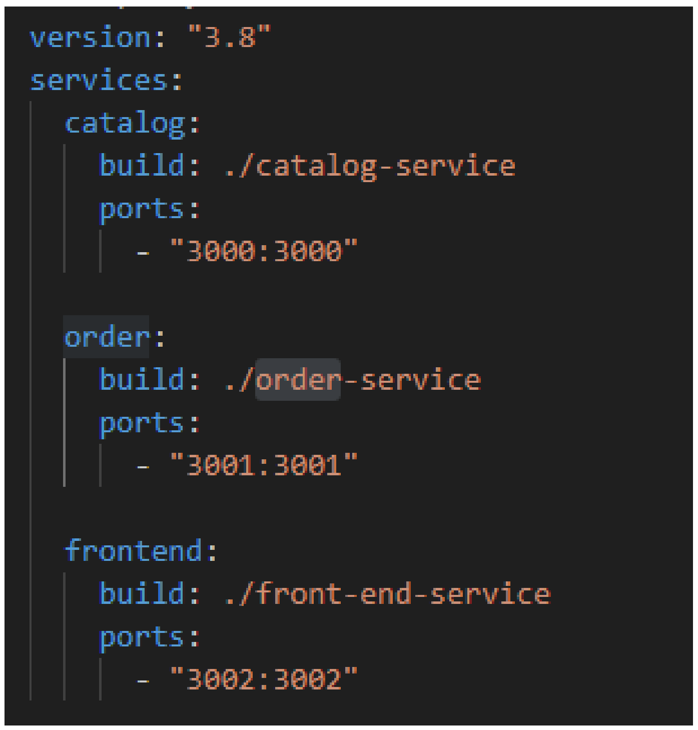
```
GET http://localhost:3001/PurchaseBook/7
```

Key	Value	Description
Key	Value	Description

200 OK • 95 ms • 247 B

```
Successful Purchase
```

docker-compose.yml File



```yaml
version: "3.8"
services:
  catalog:
    build: ./catalog-service
    ports:
      - "3000:3000"

  order:
    build: ./order-service
    ports:
      - "3001:3001"

  frontend:
    build: ./front-end-service
    ports:
      - "3002:3002"
```

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

Conclusion

Our project demonstrates a thorough and detailed process of building and deploying a microservices architecture in Docker containers. We have leveraged technologies like NodeJS, Express, Axios, and SQLite to build a scalable and efficient system that could handle different types of requests and data transfers between services. Through the use of Docker and docker-compose, we have been able to orchestrate and deploy our services seamlessly and ensure consistency and reliability across environments. Our focus on error handling, portability, and efficient use of resources clearly depicts our promise of delivering strong and robust solutions. This project, therefore, is not only addressing the problems associated with distributed systems but also has laid down a good foundation for future extensions and scalability.