Computer Engineering Department

Course Name: Distributed Operating System

PROJECT: Bazar.com: A Multi-tier Online Book Store

Dr.Samer Arandi

Student Name	Student Id
Shahed Jawabreh	12028521
Aya Walid Awwad	12043062

Bazar.com exemplifies the benefits of using **Docker** to create a scalable and efficient online bookstore. With its **Catalog Service** facilitating book searches and information retrieval, the **Order Service** ensuring secure and seamless purchases, and the **Front Service** providing an engaging user interface, Baza.com delivers a comprehensive solution that meets the diverse needs of its customers.

Tools and library used in project :

- Docker
- Nodejs
- REST Api

Library

- Express
- csv-parser
- Body-parser
- axios
- Cors

We developed the **backend using Node.js** and Express.js, enabling us to make use of the endpoints for communication between the frontend and the microservices And use library to deal with csv file

First We Are Create 3 Images as shown in Figure:

The **fronted** used to write code in the Operating System Host and test it Using Postman and We make it appear as an Gui by used several Languages Html,Css,Javascript and the fronted put in port 80

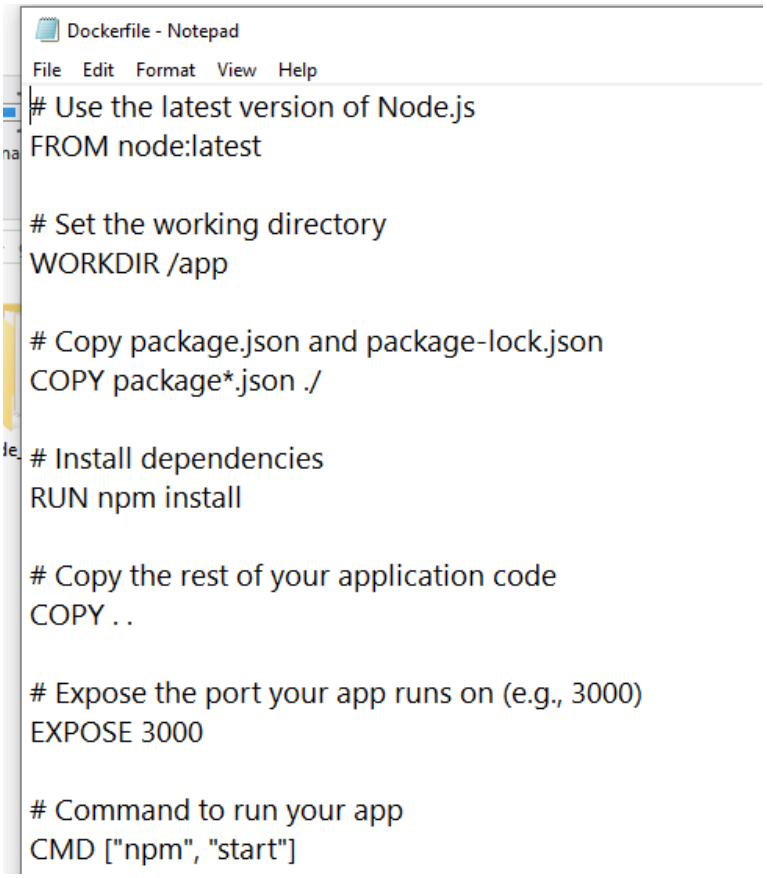
Catalog The service that used to receive request from front just request **Search & info** at port 3000

Order The service that used to receive request from front just request **purchase** at port 3001

Dockerized Service Interconnectivity

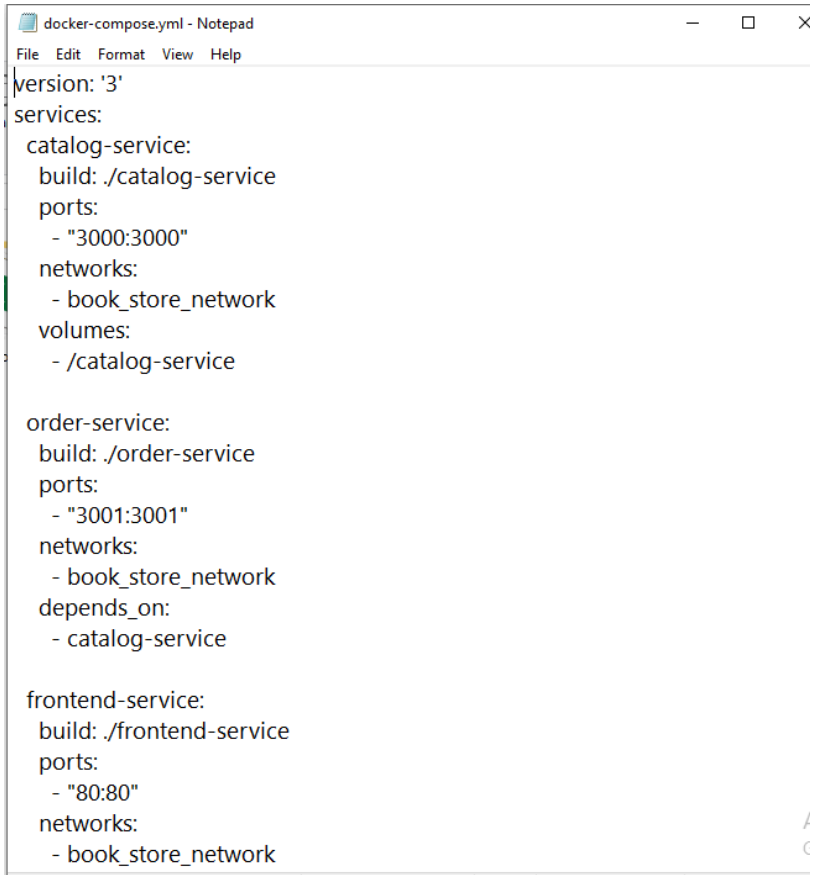
Each service is built and deployed as a Docker image, allowing Bazar.com to manage and connect these services seamlessly. By connecting the services through their dedicated ports, Docker allows for isolated environments that can still communicate efficiently, enabling Bazar.com to provide users with streamlined shopping experience.

This Dockerfile automates setting up a Node.js environment with all necessary dependencies, so the app can run consistently in any environment.



- **FROM node**
  This sets the base image to the latest version of Node.js, which includes Node and npm.
- **WORKDIR /app**
  Sets the working directory inside the container to /app. This is where your app's files will be stored and run.
- **COPY package*.json ./**
  Copies package.json and package-lock.json files to the /app directory in the container. These files contain your app's dependencies.
- **RUN npm install**
  Installs all the dependencies specified in package.json.
- **COPY . .**
  Copies the rest of your application's code from your local machine to the /app directory in the container.
- **EXPOSE 3000**
  Opens port 3000 on the container, which is where your app will be accessible. And when i create Dockerfile to order service , all command the same , but different in this command , it run on port 3001

- **CMD ["npm", "start"]**
  Sets the command to start your app. When the container runs, it executes npm start to launch your application.
  This is a docker-compose.yml file that defines and manages multiple services (containers) for a Node.js application with a **catalog service** and an **order service**.



- Catalog-service :
  build: ./catalog-service – This tells Docker to build the catalog-service container from a Dockerfile located in the catalog-service directory.
  ports: 3000:3000 – Exposes port 3000 on the host machine and maps it to port 3000 inside the container.

  networks: book_store_network – Connects the service to a network named book_store_network, allowing it to communicate with other services in the network.

- Order-service :

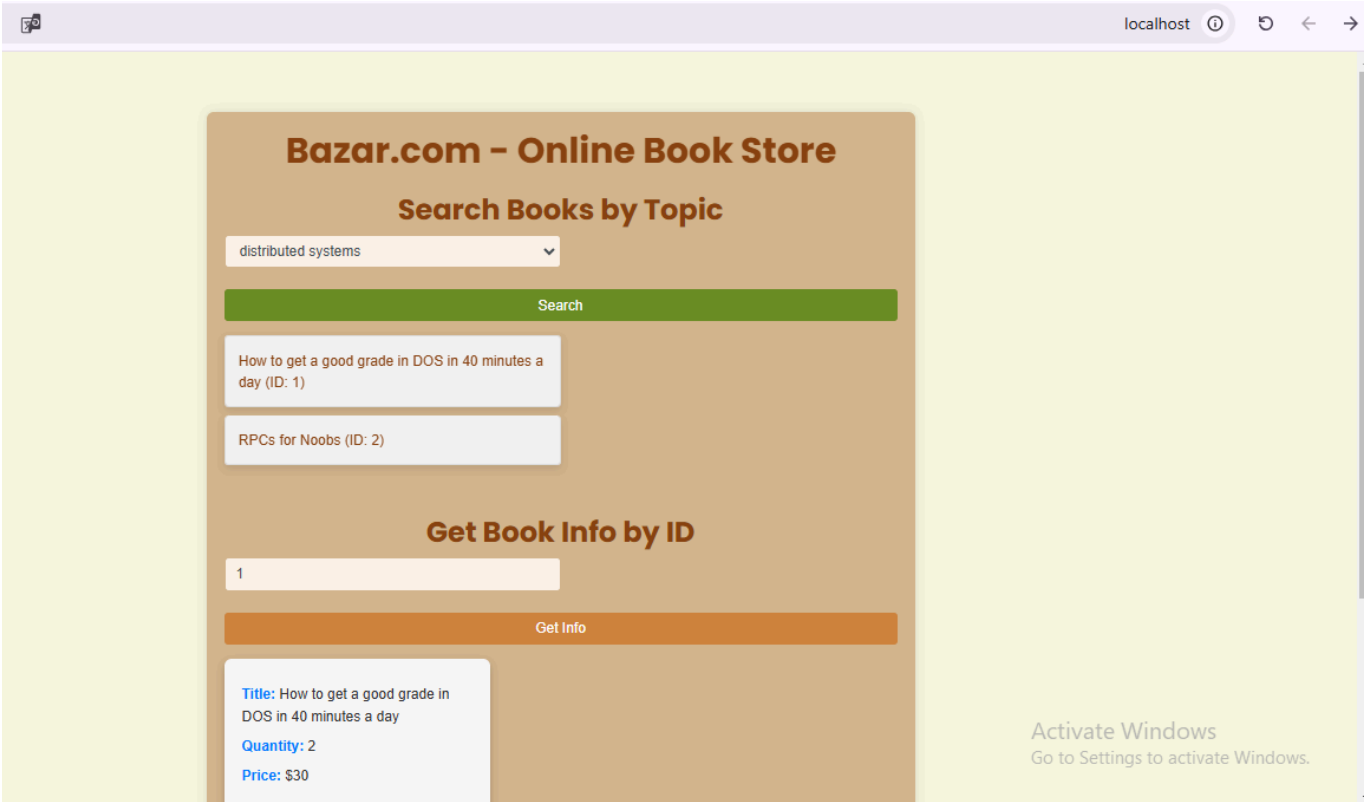
  depends_on: catalog-service – Ensures the catalog-service starts before the order-service, so dependencies are loaded in the correct order.

- Network :

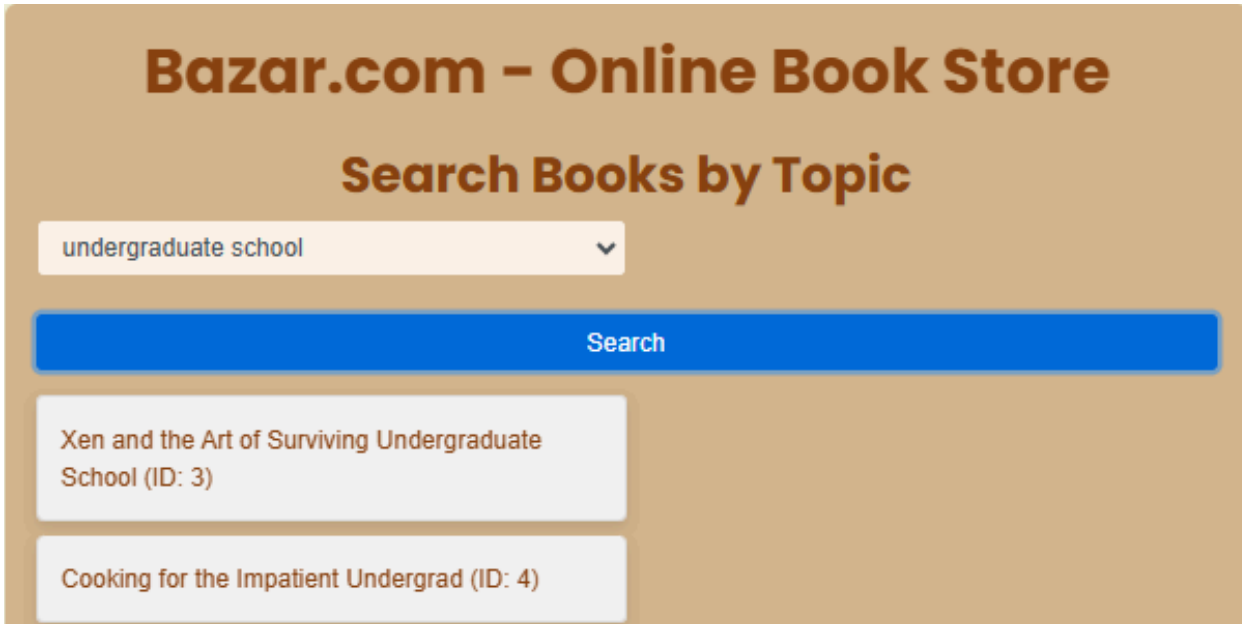
  book_store_network: driver: bridge – Creates an isolated bridge network, book_store_network, enabling services to communicate internally without exposing all ports externally.

When I run docker-compose up command, Docker Compose reads the docker-compose.yml file and accesses the associated Dockerfiles for each service specified in the configuration. It builds the necessary images based on these Dockerfiles, creates containers for each service, and sets up networks for seamless communication between them.

Final output :



Catalog-Service



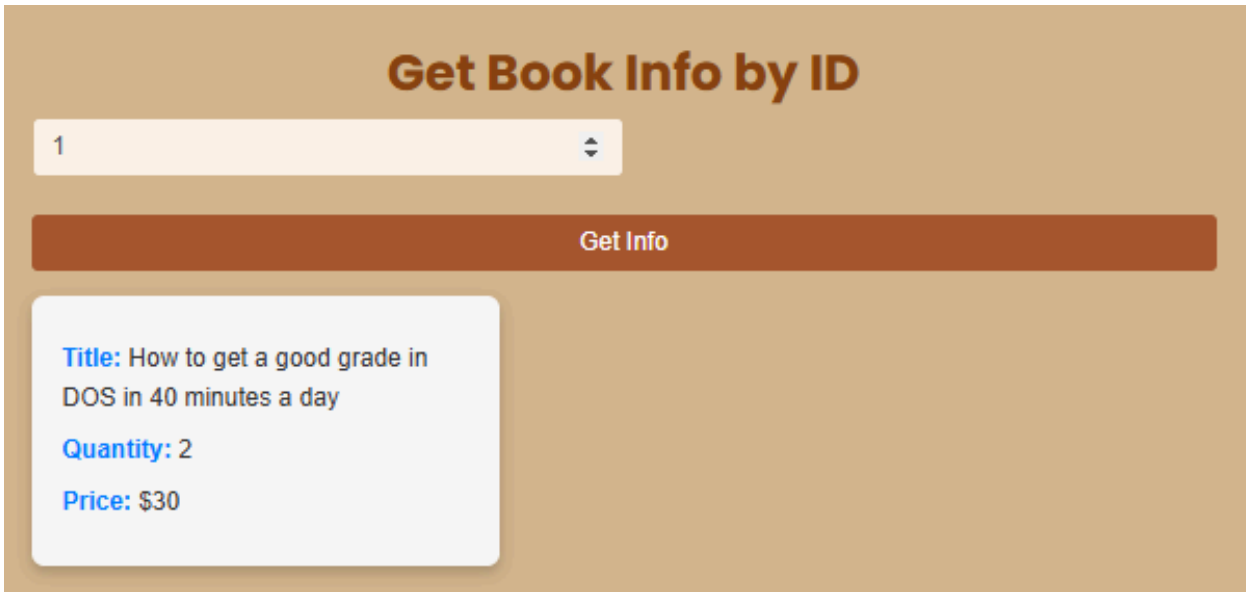

```
// Search for books by topic
function searchBooks() {
    const topic = document.getElementById('topic').value;
    if (!topic) {
        alert('Please select a topic.');
        return;
    }

    fetch(`${catalogServerUrl}/search/${encodeURIComponent(topic)}`)
        .then(response => response.json())
        .then(data => {
            const resultsList = document.getElementById('search-results');
            resultsList.innerHTML = '';
            data.forEach(book => {
                const listItem = document.createElement('li');
                listItem.className = 'list-group-item';
                listItem.textContent = `${book.Title} (ID: ${book.ID})`;
                resultsList.appendChild(listItem);
            });
        })
        .catch(error => console.error('Error fetching books:', error));
}
```

When i press to button (search) ,, searchBook() function is invoked ,This function sends a fetch request to the **search API** that make by catalog service and passing the selected topic as a parameter

Search api in catalog-service ,, read data from file and depend on topic retrieves the list of books that match the specified topic.

```
// Search by topic
app.get('/search/:topic', async (req, res) => {
    try {
        const catalog = await readCsv('catalogData.csv'); // Await the reading of the CSV
        const topic = req.params.topic.toLowerCase();
        const filteredResults = catalog.filter(book => book.Topic.toLowerCase() === topic);
        res.json(filteredResults);
    } catch (error) {
        res.status(500).send('Error reading catalog data');
    }
});
```



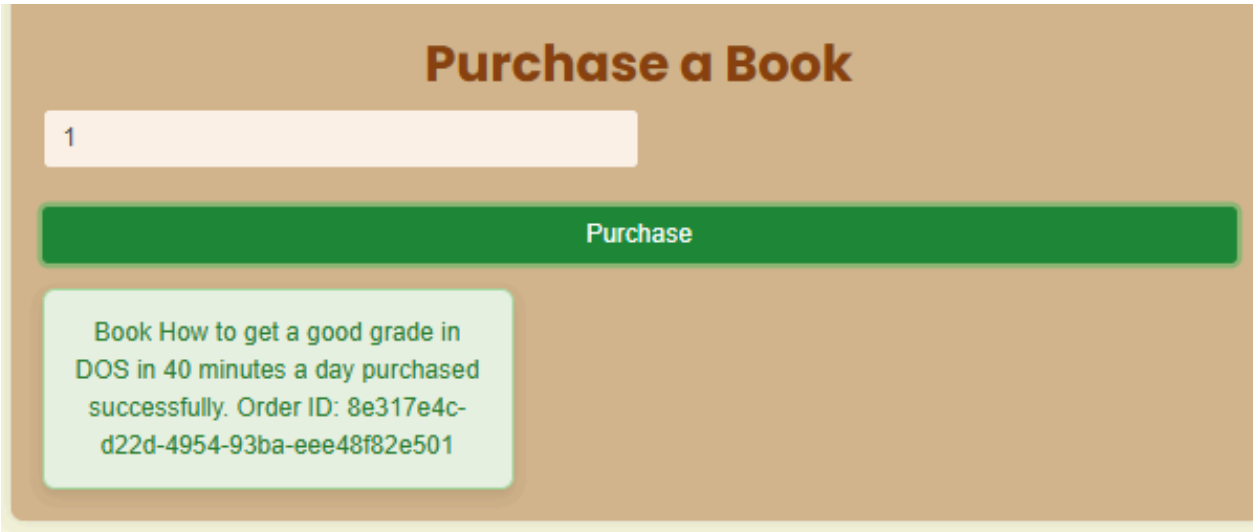
When I press the **Get Info** button in the frontend , the below function is invokedThis function sends a fetch request to the **Catalog API** provided by the **CatalogServer.**

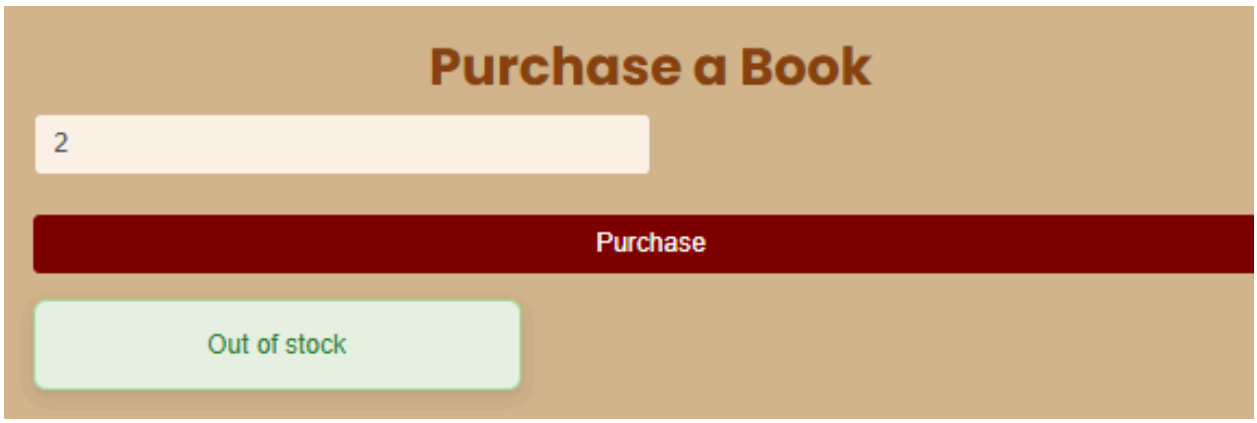
```
// Get book info by ID
function getBookInfo() {
    const bookId = document.getElementById('book-id').value;
    if (!bookId) {
        alert('Please enter a book ID.');
        return;
    }
    console.log(catalogServerUrl)
    fetch(`${catalogServerUrl}/info/${bookId}`)
    .then(response => response.json())
    .then(data => {
        console.log(data);
        const bookInfoDiv = document.getElementById('book-info');
        bookInfoDiv.innerHTML = `
            <p><span>Title:</span> ${data.Title}</p>
            <p><span>Quantity:</span> ${data.Stock}</p>
            <p><span>Price:</span> $${data.Price}</p>
        `;
    })
        .catch(error => console.error('Error fetching book info:', error));
}
```

This Function **getBookInfo** main Function to fetch book information based on user input -book Id, and make it **input Validation** it to ensure the user has entered a book id before this process

fetch() make an asynchronous Http get request to retrieve book data, after that make **Response**

To convert the server response into json format, Finally Update the Web Page with the fetched book info , &catch for any error that may occur during the fetching process.





When I press the purchase button in the frontend , the below function is invokedThis function sends a fetch request to the **Order API** provided by the **Order Service**, passing along the book's ID that the user wants to purchase.

```
function purchaseBook() {
    const bookId = document.getElementById('purchase-book-id').value;
    if (!bookId) {
        alert('Please enter a book ID.');
        return;
    }

    fetch(`${orderServerUrl}/purchase/${bookId}`, { method: 'POST' })
.then(response => response.text())
.then(result => {
    const purchaseResultDiv = document.getElementById('purchase-result');
    purchaseResultDiv.textContent = result;
})
.catch(error => {
    console.error('Error making purchase:', error);
    const purchaseResultDiv = document.getElementById('purchase-result');
    purchaseResultDiv.textContent = 'There was an error processing your purchase. Please try again.';
    purchaseResultDiv.style.color = '#d9534f';
    purchaseResultDiv.style.backgroundColor = '#f8d7da';
    purchaseResultDiv.style.border = '1px solid #f5c6cb';
});
```

This API endpoint allows users to purchase a book by its unique ID. When a user initiates a purchase, the endpoint **retrieves the book's details from the Catalog Service** using the provided ID. It then checks if the book is in stock by verifying that the stock count is greater than zero. If the book is available, a unique order ID is generated, and a request is made to the Catalog Service to update the stock, reducing it by one. If this update is successful, an order record is created with the new order ID and book ID. The API then sends a confirmation message back to the user, including the book title and order ID. If the book is out of stock, the user is notified,

```
// Endpoint to purchase a book
app.post('/purchase/:id', async (req, res) => {
    const id = parseInt(req.params.id);
    const books = await axios.get(`http://catalog-service:3000/info/${id}`);
    const book = books.data
    try {
        if (book && parseInt(book.Stock) > 0) { // Ensure to parse stock to an integer

            // Create a new order
            const newOrderId = uuidv4(); // Generate a unique order ID using uuid
            const response = await axios.post(`http://catalog-service:3000/stock/${id}`);
            if(response.status == 200)
           { await createOrder(newOrderId, id); // Create the order

            res.send(`Book ${book.Title} purchased successfully. Order ID: ${newOrderId}`);}
            else  res.send('Out of stock');
        } else {
            res.send('Out of stock');
        }
    } catch (error) {
        console.error('Error purchasing the book:', error);
        res.status(500).send('Internal Server Error');
    }
});
```

Conclusion:

We are designing a system scalable for managing the book store microservice **REST API** and **lightweight framework** as docker so we can achieve flexibility , modularity ,easy development and implementation.