An-Najah National University

Computer Engineering Department

Distributed Systems - DOS

Project – Part1 Report

Shams Abd Al-Aziz

Yara Daraghmeh

Dr. Samer Arandi

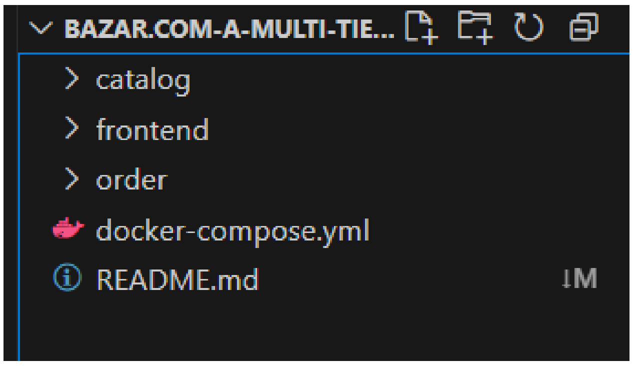
Introduction

Bazar.com is a tiny online bookstore built with microservices. It has three parts: a front-end, a catalog server, and an order server. Each service runs in its own Docker container and communicates via REST APIs. We used Flask (Python) and CSV files for storing book and order data. The system lets users search, view, and buy books.

Discussion:

at the first, this is our project hierarchy, since we have 3 services, we made a 3 folder, each folder contain one service.

Also, we put the docker-compose file in the root directory, this file run all services (catalog, order, frontend) together easily, simplifies setup, starts services in the right order, and run all services in one time, not manually. Also, we ass readme file for the GitHub repository.

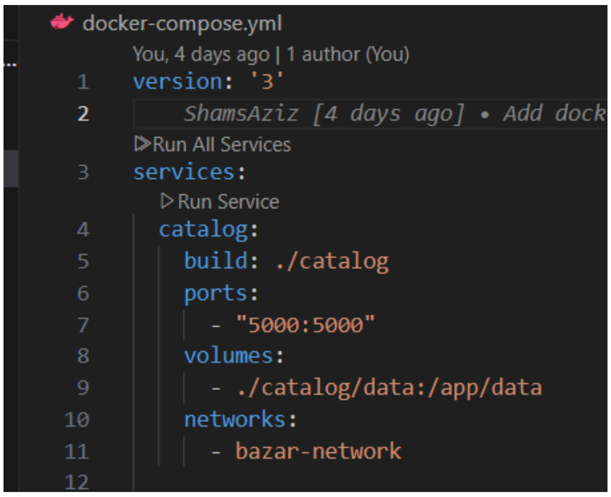


```
BAZAR.COM-A-MULTI-TIE...
  > catalog
  > frontend
  > order
  docker-compose.yml
  README.md                 M
```

let's start with the compose file:

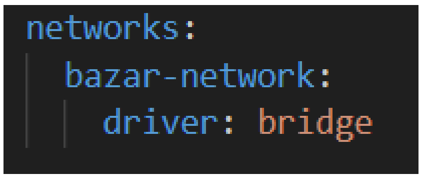
at the first, we define the services, and put the 3 services (catalog, frontend, order) in correct location in yml file under services.

For each service, like catalog, we define the image that the container will be instanced from it, we use build to build the container from the image that defined in docker file in catalog folder, and we use port to set the port that the service and container will work on.



```
docker-compose.yml
You, 4 days ago | 1 author (You)
version: '3'
  ShamsAziz [4 days ago] • Add dock
services:
  catalog:
    build: ./catalog
    ports:
      - "5000:5000"
    volumes:
      - ./catalog/data:/app/data
    networks:
      - bazar-network
```

Then we map the file on host (./order/data) to the container (/app/data) so data is saved even if the container stops, by put volume (-v in cmd). Also, we add networks to connect the service to a shared Docker network so it can communicate with other services using their names.



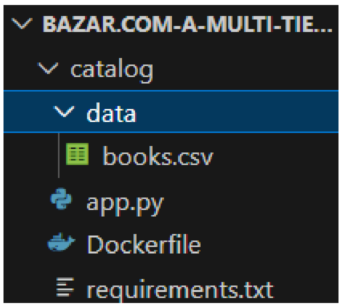
```
networks:
  bazar-network:
    driver: bridge
```

The same we did for the left two services, but we add depends on, like in order service we let it depends on catalog, to make sure the catalog service starts before order, since order depends on it to check stock.

Note: we use port 5000 for catalog service, 5001 for order, 5002 for frontend.

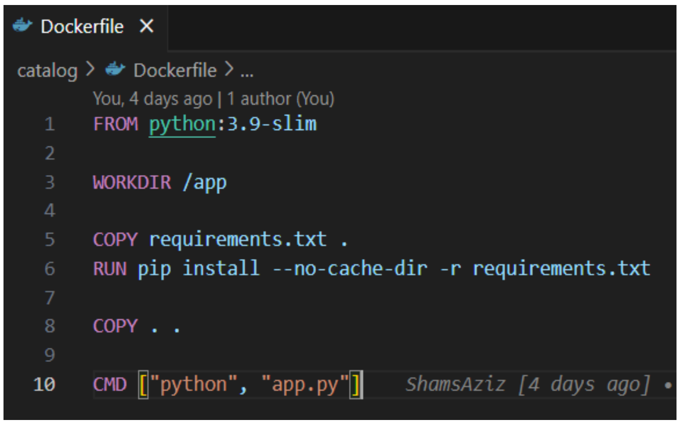
For the catalog service:

We define a data file of type CSV, that store books info, and we put the docker file that builds the Docker image for the catalog service (installs dependencies, sets up app), also python file called app, it is Flask file that defines the catalog service logic, routes and REST API, and the requirements txt file, it has lists Python packages needed (like Flask). Used during Docker build.


```
BAZAR.COM-A-MULTI-TIE...
  catalog
    data
      books.csv
    app.py
    Dockerfile
    requirements.txt
```

For docker file:

At first, we define the image the container will instance of it, it is python with v-3.9, because we use flask as a framework, then make the current directory in the container is app in root dir., then copy the requirements from host to the container to let the container run correctly, then run a command to install Python dependencies without keeping cache (saves space). Also copy all project files into the container. Then runs the app main file when the container starts.


Dockerfile

catalog > Dockerfile > ...

You, 4 days ago | 1 author (You)
```
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["python", "app.py"]    ShamsAziz [4 days ago] •
```

Like this file we made the 2 docker files for other two services (order, frontend), it is the same.

For app.py:

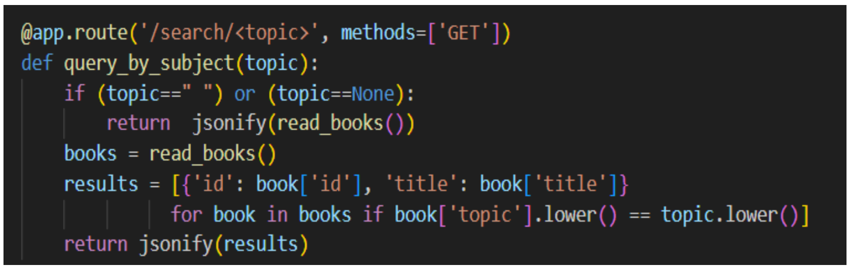
Here in catalog service, we route 3 requests in this service, search by book topic, search by book id, receive request to purchase from order service, to pay a book and let the qnt decrease in books database also to check if the book in stock, since the catalog service controls the books database.

The requests:
http://catalog:5000/search/<topic>

http://catalog:5000/info/<int: item_id>

http://catalog:5000/update/<int: item_id>

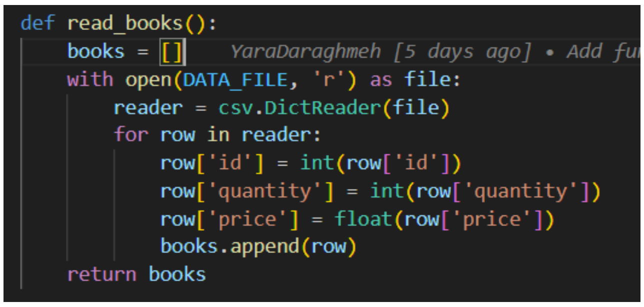
Here we define the first route, to let the user search on some topic and the result of list of books that read from CSV books file will be returned in Json object.



```
@app.route('/search/<topic>', methods=['GET'])
def query_by_subject(topic):
    if (topic==" ") or (topic==None):
        return  jsonify(read_books())
    books = read_books()
    results = [{'id': book['id'], 'title': book['title']}
               for book in books if book['topic'].lower() == topic.lower()]
    return jsonify(results)
```

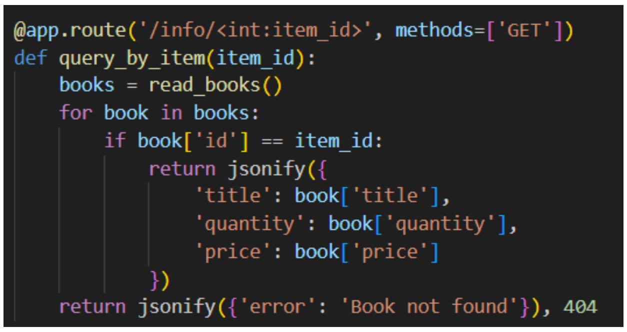
Since this is read function:

Also, we made a one to update books info, like qnt in stock



```
def read_books():
    books = []      YaraDaraghmeh [5 days ago] • Add fu
    with open(DATA_FILE, 'r') as file:
        reader = csv.DictReader(file)
        for row in reader:
            row['id'] = int(row['id'])
            row['quantity'] = int(row['quantity'])
            row['price'] = float(row['price'])
            books.append(row)
    return books
```

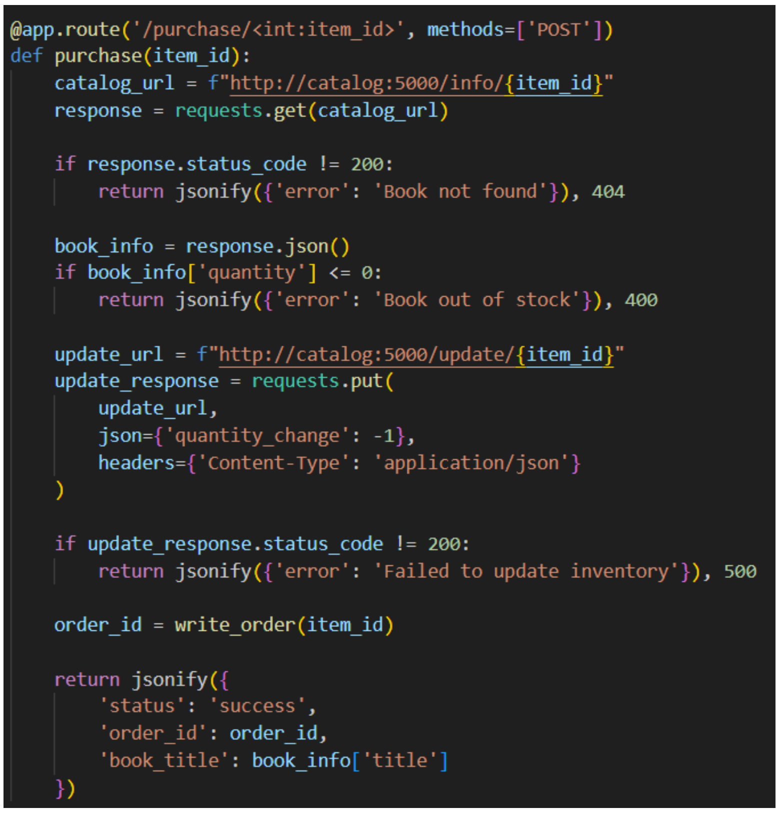
We did the same thing for other two routes, for the get book info by its id, and update the qnt in stock, but each one in its route has some functionality to do and some verifications. Like this picture of the right, to search some book info, if the book does not exist return false.



```
@app.route('/info/<int:item_id>', methods=['GET'])
def query_by_item(item_id):
    books = read_books()
    for book in books:
        if book['id'] == item_id:
            return jsonify({
                'title': book['title'],
                'quantity': book['quantity'],
                'price': book['price']
            })
    return jsonify({'error': 'Book not found'}), 404
```

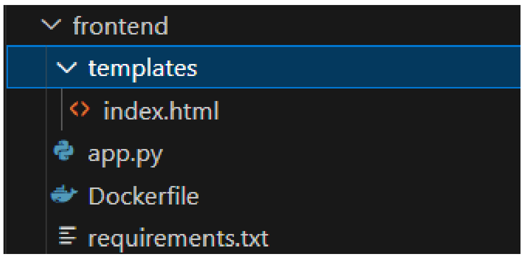
For order service:

It has the same files that exists in catalog folder, but here the CSV file contains the orders that being successed. It has a one route, because it gives the purchase service. As we see on the right, we made a route to purchase some book, at the first this service sent a request to the catalog to check if the book is exists, if not return error, then check if the qnt>0 to let the user buy the book, then send a request to the catalog service to update the qnt in stock, then add the order to the orders file, and return success message to the user.



```
@app.route('/purchase/<int:item_id>', methods=['POST'])
def purchase(item_id):
    catalog_url = f"http://catalog:5000/info/{item_id}"
    response = requests.get(catalog_url)

    if response.status_code != 200:
        return jsonify({'error': 'Book not found'}), 404

    book_info = response.json()
    if book_info['quantity'] <= 0:
        return jsonify({'error': 'Book out of stock'}), 400

    update_url = f"http://catalog:5000/update/{item_id}"
    update_response = requests.put(
        update_url,
        json={'quantity_change': -1},
        headers={'Content-Type': 'application/json'}
    )

    if update_response.status_code != 200:
        return jsonify({'error': 'Failed to update inventory'}), 500

    order_id = write_order(item_id)

    return jsonify({
        'status': 'success',
        'order_id': order_id,
        'book_title': book_info['title']
    })
```

Now, let us talk about frontend service:



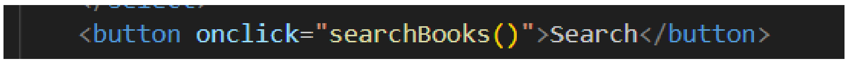
```
frontend
  templates
    index.html
  app.py
  Dockerfile
  requirements.txt
```

It has the same files exists in catalog service, I mean the same docker and requirements files, but here we change the content of app file, and we add index html page to make the GUI and let the user using services easily.

In the index.html file, we build a simple web interface with buttons and inputs that let users search for books, view details, and make purchases. Each button triggers a function that sends a request to the frontend service. This service, defined in app.py, handles the incoming requests using Flask routes, then communicates with the catalog and order services by REST APIs to fetch data or complete transactions. It's a clean flow from the browser to backend services.

In index.html:

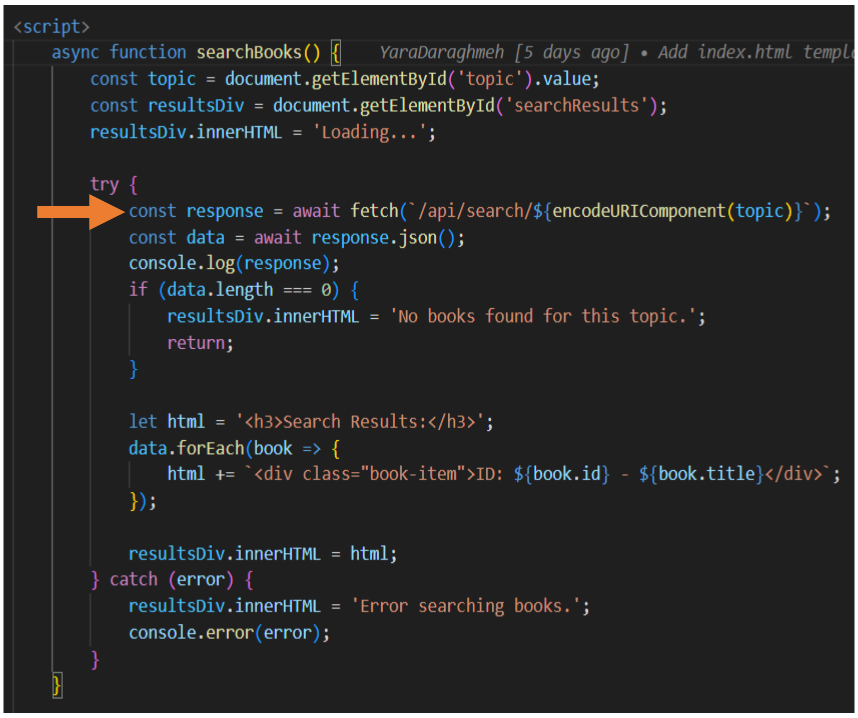
We write an html page, it contains a buttons, for example search button, when the user clicks on this button, then a JavaScript function will execute, it called searchBooks.



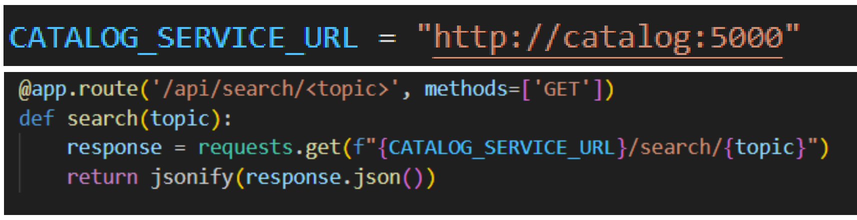
```
<button onclick="searchBooks()">Search</button>
```

In this function, the frontend sends a request to frontend service using fetch, and app.py which is backend code will handle the request using routes, and then send another search request to catalog service easily.

After that, if there is a response then the result will back from catalog service that runs in its container as Json object to frontend service, then it will be showed in web page as we see in the code on right side.



```
<script>
    async function searchBooks() {     YaraDaraghmeh [5 days ago] • Add index.html templ
        const topic = document.getElementById('topic').value;
        const resultsDiv = document.getElementById('searchResults');
        resultsDiv.innerHTML = 'Loading...';

        try {
            const response = await fetch(`/api/search/${encodeURIComponent(topic)}`);
            const data = await response.json();
            console.log(response);
            if (data.length === 0) {
                resultsDiv.innerHTML = 'No books found for this topic.';
                return;
            }

            let html = '<h3>Search Results:</h3>';
            data.forEach(book => {
                html += `<div class="book-item">ID: ${book.id} - ${book.title}</div>`;
            });

            resultsDiv.innerHTML = html;
        } catch (error) {
            resultsDiv.innerHTML = 'Error searching books.';
            console.error(error);
        }
    }
```



```
CATALOG_SERVICE_URL = "http://catalog:5000"
```

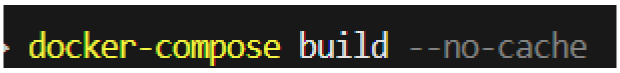
```
@app.route('/api/search/<topic>', methods=['GET'])
def search(topic):
    response = requests.get(f"{CATALOG_SERVICE_URL}/search/{topic}")
    return jsonify(response.json())
```

here is the code that exist in frontend backend service, in app.py, it makes a route to handle the requests that comes from web page, then send another request to other services like catalog for search a topic.

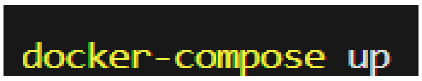
Note: we use port 5000 for catalog service, and 5001 for order and 5002 for frontend.

Outputs:

to run the full app, at the first on local terminal we run these commands:

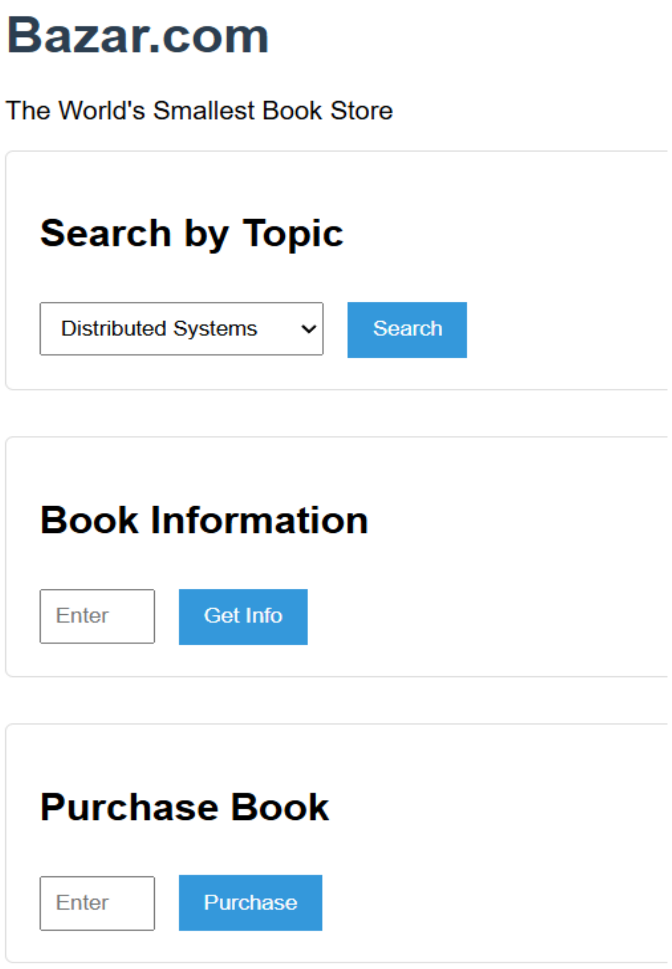
1- 

```
docker-compose build --no-cache
```

2- 

```
docker-compose up
```

3- Open 127.0.0.1:5002/ in the browser, then index html will load.



Bazar.com

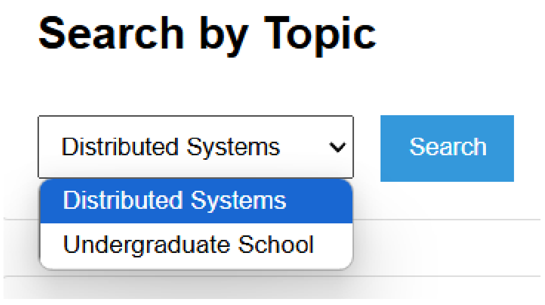
The World's Smallest Book Store

Search by Topic

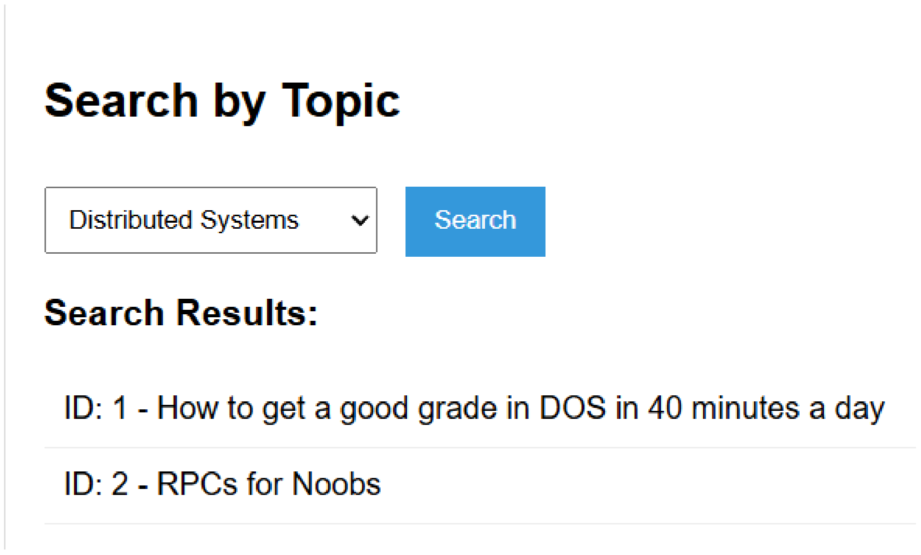
Book Information

Purchase Book

So here user can choose the topic of books that he wants, then click search.



And the result:


Search by Topic

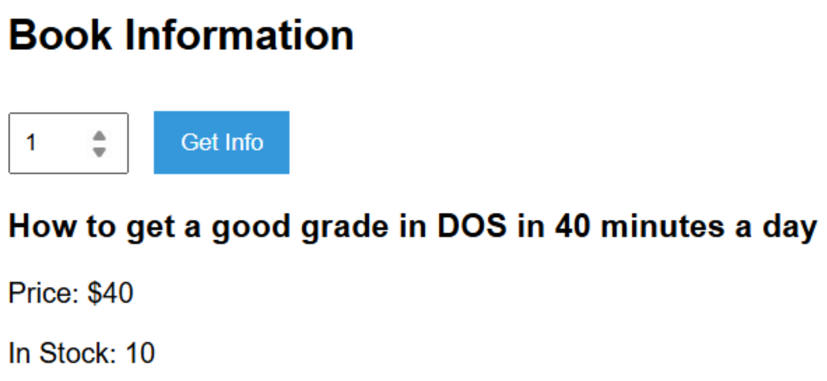
Search Results:

ID: 1 - How to get a good grade in DOS in 40 minutes a day

ID: 2 - RPCs for Noobs

Also, user can search book by its id (it has range to search in, to decrease the errors):

and the result is the information about the book.


Book Information

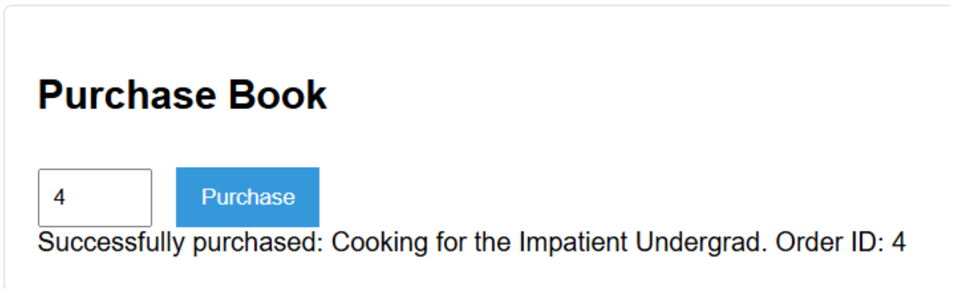
How to get a good grade in DOS in 40 minutes a day

Price: $40

In Stock: 10

Also, user can purchase a book just by entering the book id:

Then if the book in stock, a successful message will appear, and the order will list in orders CSV file, and the qnt in books info CSV file will decrease.

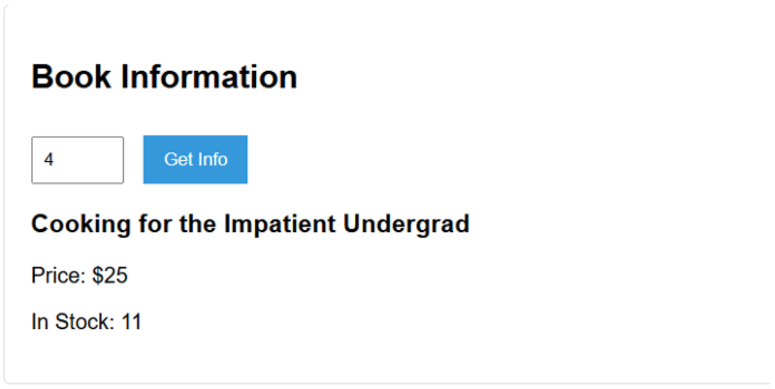


Purchase Book

4 Purchase

Successfully purchased: Cooking for the Impatient Undergrad. Order ID: 4

When we search the book with is 4, we see that the qnt decreased (it was 12 as the picture above in the report).



Book Information

4 Get Info

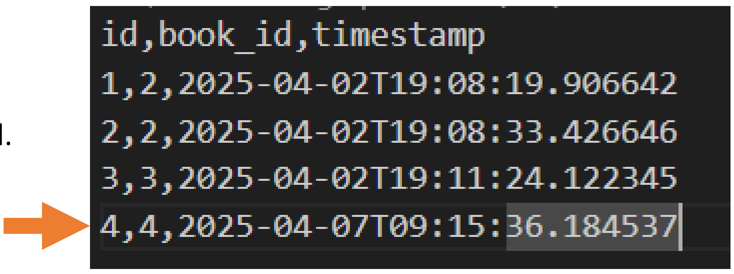
Cooking for the Impatient Undergrad

Price: $25

In Stock: 11

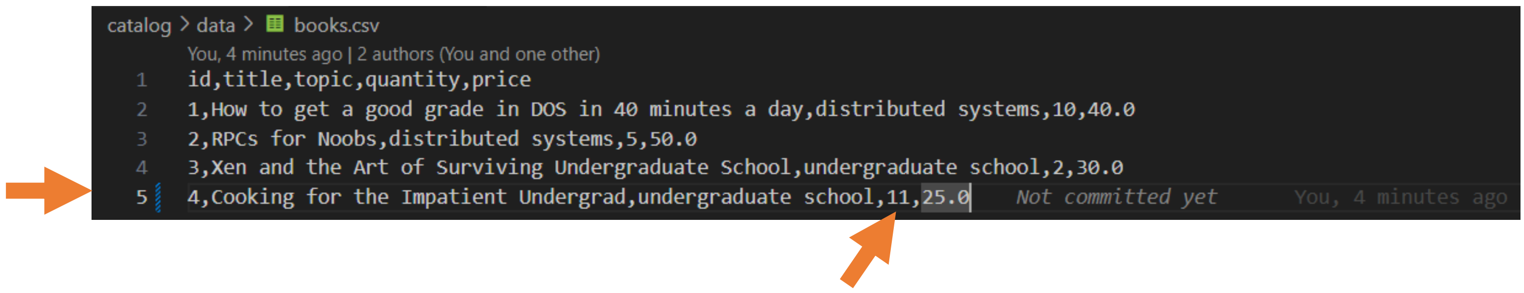
in orders file:

we see, the new order listed.



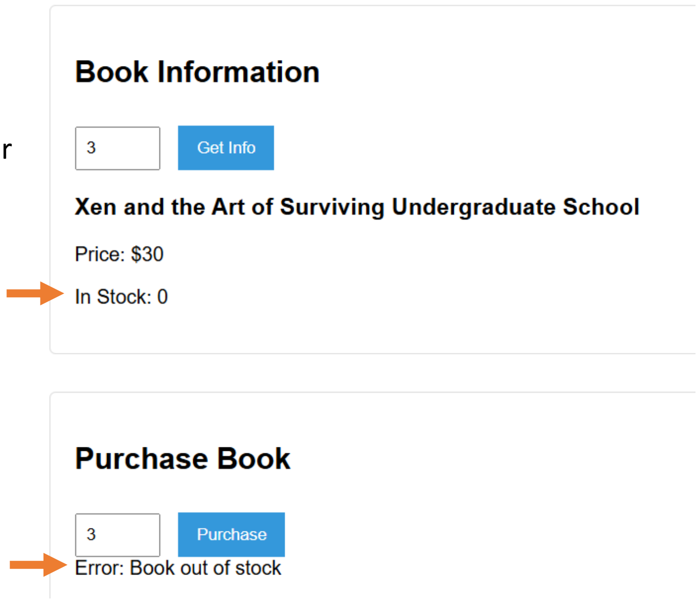
```
id,book_id,timestamp
1,2,2025-04-02T19:08:19.906642
2,2,2025-04-02T19:08:33.426646
3,3,2025-04-02T19:11:24.122345
4,4,2025-04-07T09:15:36.184537
```

And this is data books file:



```
catalog > data > books.csv
You, 4 minutes ago | 2 authors (You and one other)
id,title,topic,quantity,price
1,How to get a good grade in DOS in 40 minutes a day,distributed systems,10,40.0
2,RPCs for Noobs,distributed systems,5,50.0
3,Xen and the Art of Surviving Undergraduate School,undergraduate school,2,30.0
4,Cooking for the Impatient Undergrad,undergraduate school,11,25.0    Not committed yet    You, 4 minutes ago
```

if the book is out of stock, then error message will appear, and the order didn't complete.



Book Information

3 | Get Info

Xen and the Art of Surviving Undergraduate School

Price: $30

In Stock: 0

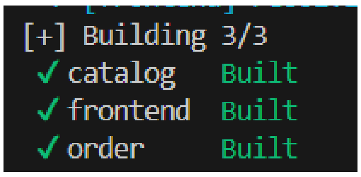
Purchase Book

3 | Purchase

Error: Book out of stock

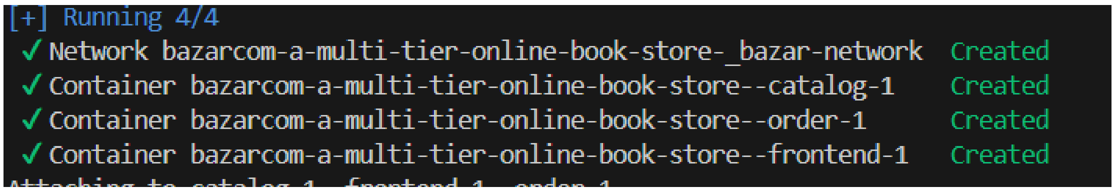
When we back to the terminal we see,

All 3 containers built.



```
[+] Building 3/3
 ✓ catalog    Built
 ✓ frontend   Built
 ✓ order      Built
```

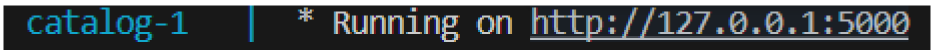
All 3 containers created and run.



```
[+] Running 4/4
 ✓ Network bazarcom-a-multi-tier-online-book-store-_bazar-network  Created
 ✓ Container bazarcom-a-multi-tier-online-book-store--catalog-1    Created
 ✓ Container bazarcom-a-multi-tier-online-book-store--order-1      Created
 ✓ Container bazarcom-a-multi-tier-online-book-store--frontend-1   Created
```

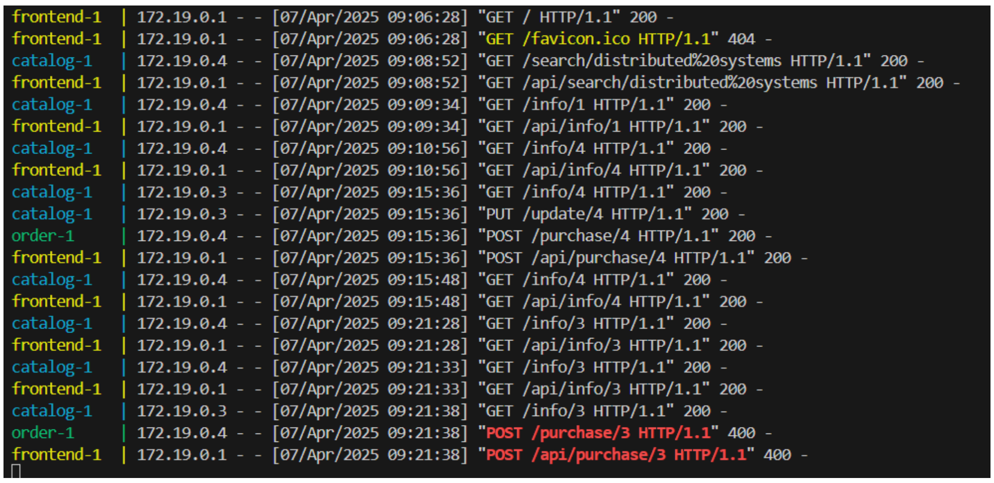
And this is indicating that everything going well.

also, it gives the URL to reach each service:



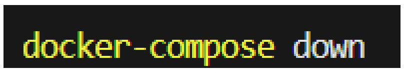
```
catalog-1   |  * Running on http://127.0.0.1:5000
```

finally, it gives all request that made and the transformations between services:

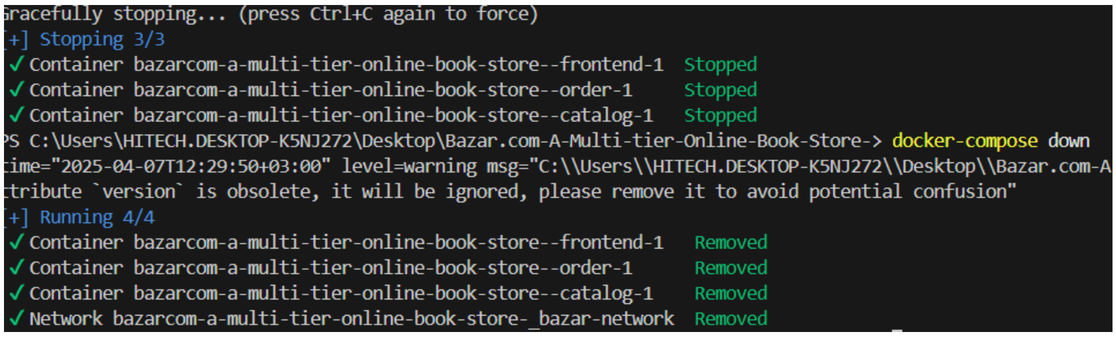


```
frontend-1  | 172.19.0.1 - - [07/Apr/2025 09:06:28] "GET / HTTP/1.1" 200 -
frontend-1  | 172.19.0.1 - - [07/Apr/2025 09:06:28] "GET /favicon.ico HTTP/1.1" 404 -
catalog-1   | 172.19.0.4 - - [07/Apr/2025 09:08:52] "GET /search/distributed%20systems HTTP/1.1" 200 -
frontend-1  | 172.19.0.1 - - [07/Apr/2025 09:08:52] "GET /api/search/distributed%20systems HTTP/1.1" 200 -
catalog-1   | 172.19.0.4 - - [07/Apr/2025 09:09:34] "GET /info/1 HTTP/1.1" 200 -
frontend-1  | 172.19.0.1 - - [07/Apr/2025 09:09:34] "GET /api/info/1 HTTP/1.1" 200 -
catalog-1   | 172.19.0.4 - - [07/Apr/2025 09:10:56] "GET /info/4 HTTP/1.1" 200 -
frontend-1  | 172.19.0.1 - - [07/Apr/2025 09:10:56] "GET /api/info/4 HTTP/1.1" 200 -
catalog-1   | 172.19.0.3 - - [07/Apr/2025 09:15:36] "GET /info/4 HTTP/1.1" 200 -
catalog-1   | 172.19.0.3 - - [07/Apr/2025 09:15:36] "PUT /update/4 HTTP/1.1" 200 -
order-1     | 172.19.0.4 - - [07/Apr/2025 09:15:36] "POST /purchase/4 HTTP/1.1" 200 -
frontend-1  | 172.19.0.1 - - [07/Apr/2025 09:15:36] "POST /api/purchase/4 HTTP/1.1" 200 -
catalog-1   | 172.19.0.4 - - [07/Apr/2025 09:15:48] "GET /info/4 HTTP/1.1" 200 -
frontend-1  | 172.19.0.1 - - [07/Apr/2025 09:15:48] "GET /api/info/4 HTTP/1.1" 200 -
catalog-1   | 172.19.0.4 - - [07/Apr/2025 09:21:28] "GET /info/3 HTTP/1.1" 200 -
frontend-1  | 172.19.0.1 - - [07/Apr/2025 09:21:28] "GET /api/info/3 HTTP/1.1" 200 -
catalog-1   | 172.19.0.4 - - [07/Apr/2025 09:21:33] "GET /info/3 HTTP/1.1" 200 -
frontend-1  | 172.19.0.1 - - [07/Apr/2025 09:21:33] "GET /api/info/3 HTTP/1.1" 200 -
catalog-1   | 172.19.0.3 - - [07/Apr/2025 09:21:38] "GET /info/3 HTTP/1.1" 200 -
order-1     | 172.19.0.4 - - [07/Apr/2025 09:21:38] "POST /purchase/3 HTTP/1.1" 400 -
frontend-1  | 172.19.0.1 - - [07/Apr/2025 09:21:38] "POST /api/purchase/3 HTTP/1.1" 400 -
```

After that, I must enter this command to down the containers, after click ctrl+c to stop eveything:



```
docker-compose down
```



```
racefully stopping... (press Ctrl+C again to force)
+] Stopping 3/3
✓Container bazarcom-a-multi-tier-online-book-store--frontend-1  Stopped
✓Container bazarcom-a-multi-tier-online-book-store--order-1     Stopped
✓Container bazarcom-a-multi-tier-online-book-store--catalog-1   Stopped
S C:\Users\HITECH.DESKTOP-K5NJ272\Desktop\Bazar.com-A-Multi-tier-Online-Book-Store-> docker-compose down
ime="2025-04-07T12:29:50+03:00" level=warning msg="C:\\Users\\HITECH.DESKTOP-K5NJ272\\Desktop\\Bazar.com-A
tribute `version` is obsolete, it will be ignored, please remove it to avoid potential confusion"
+] Running 4/4
✓Container bazarcom-a-multi-tier-online-book-store--frontend-1   Removed
✓Container bazarcom-a-multi-tier-online-book-store--order-1      Removed
✓Container bazarcom-a-multi-tier-online-book-store--catalog-1    Removed
✓Network bazarcom-a-multi-tier-online-book-store-_bazar-network  Removed
```

Conclusion:

This project involves building a multi-tier online bookstore with **microservices**. The frontend communicates with the catalog and order services using **REST APIs** to search books, get details, and process purchases. The catalog manages book info, while the order service handles purchases and updates stock. The entire system is deployed using **Docker** containers for easy management and communication between services.