# Distributed and Operating Systems

## Development of an Online Bookstore using Node.js and Docker

**Groub Memebers : Shahd Ismail ( 90%  from the work  )**

**Hadeel Jumaa (10% from the work ➔ only part from the report )**

**Github repo link: https://github.com/shahdism/DOS_LAB1**

## Introduction:

This project involved the design, development, and deployment of an online bookstore using **Node.js** and Docker. The bookstore is a web application that allows users to browse, search, and purchase books. It is built using a microservices architecture, with a front-end application written in JavaScript and a back-end API written in **Express.js** that used to handle http requests. The application is containerized using Docker for easy deployment and scalability.

**Part 1:**

**Data Base**

**We use SQLite to implement the data base , it contains these fields :**

**( id,  title ,  topic ,  cost ,stokItem ) , the data base contain information about a book**

**There is two topics in this project : distributed systems (first two**

**books) and undergraduate school (the last two books).**

1. How to get a good grade in DOS in 40 minutes a day.
2. RPCs for Noobs.
3. Xen and the Art of Surviving Undergraduate School.

4. Cooking for the Impatient Undergrad.

**The stock item represent the number of the books that available in the store**

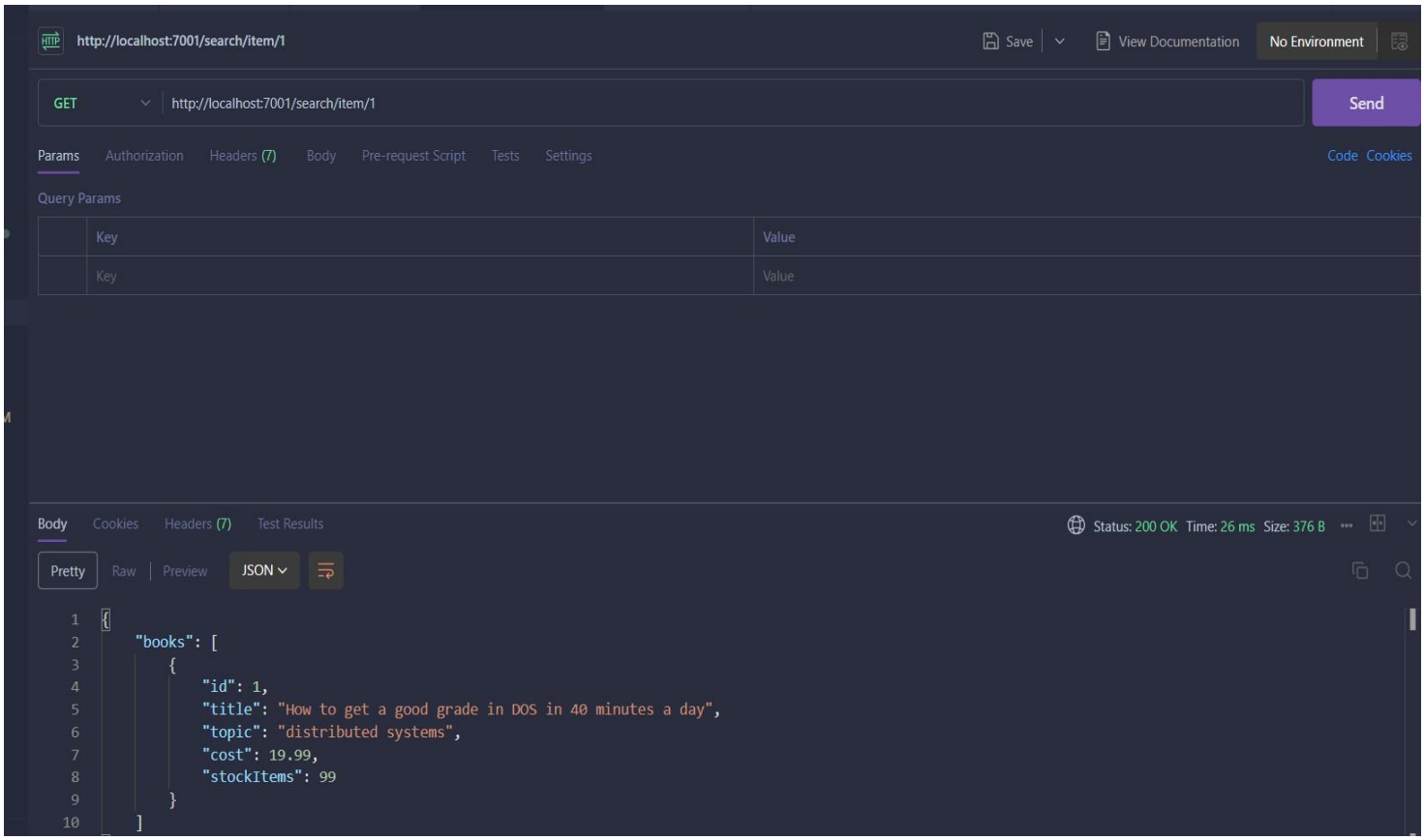
**Part 2:**

**Catalog server :**

The catalog server supports two operations: query and update. Two types of queries are supported: query-by-subject and query-by-item. In the first case, a topic is specified and the server returns all matching entries. In the second case, an item is specified and all relevant details are returned. The update operation allows the cost of an item to be updated or the number of items in stock to be increased or decreased

**Here are some results for the End points using PostMan :**

1. in this case the end point that used is /serach/item/id : search for an item by its id , the server run on port 7001 , the response is jason file contain the book information :



2. in this End point , search for the book by its topic , search/subject/distributed systems , it returns two books that have the topic distributed systems

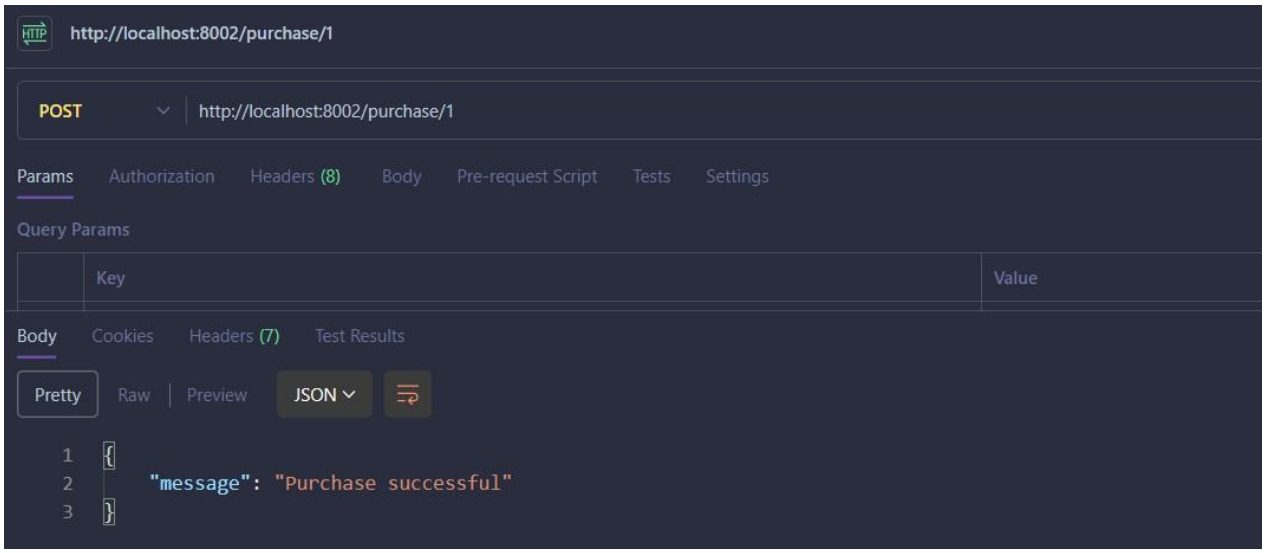

3. this End point Work when the user make the purchase operation from order server , the items in stock updated when the user buy a book it decrement by one /updateStock/2

# Part 3 :

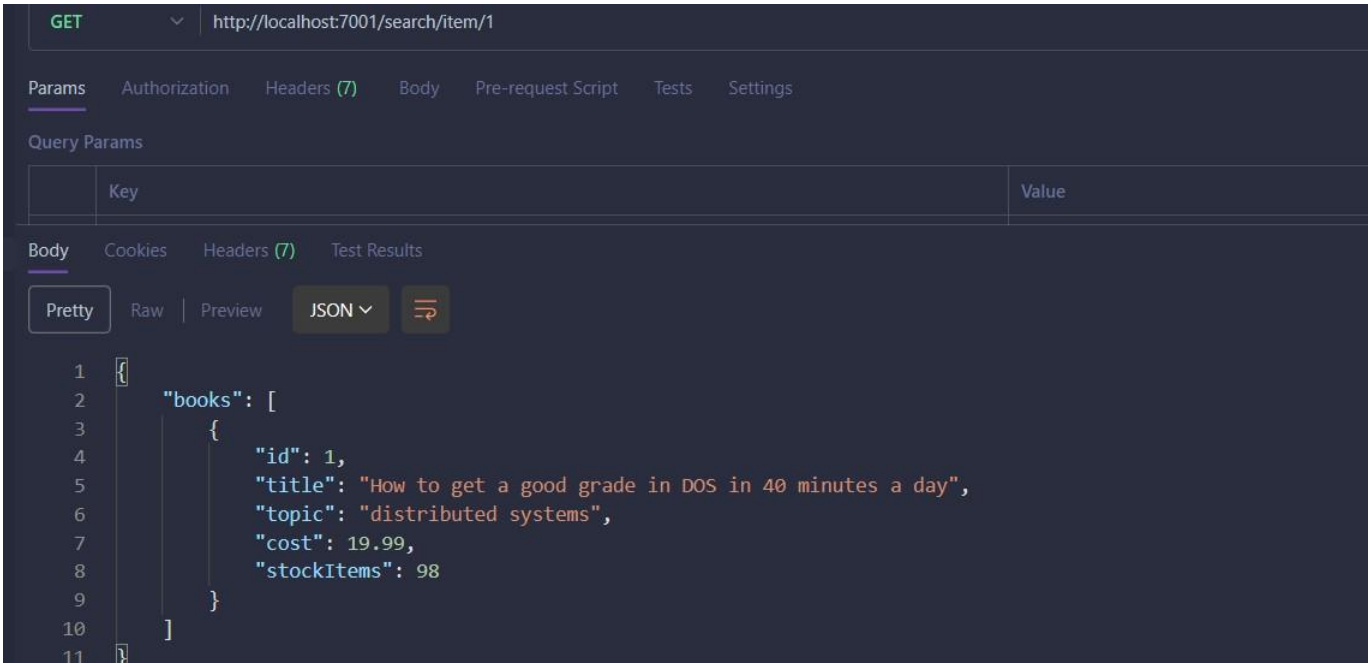
## Order Server

The order server supports a single operation: purchase(item_number). Upon receiving a purchase request, the order server must first verify that the item is in stock by querying the catalog server and then decrement the number of items in stock by one. The purchase request can fail if the item is out of stock. The server run on port 8002

1. Purchase :




As we see here the purchase done successfully, if we check the book information using the previous end point from the catalog server , we see that the stock item decreased by one  from 99 to 98




## How do we apply that ?

To connect between the two services order and catalog , we used the axios library it used to communicate external services together , in our case ,The axios used to send request to catalog server when the order server apply purchase operation , to make sure that the stock item have been decreased by one.

## Part 4 :

## Front End:

The front end server supports three operations:

• search(topic) - which allows the user to specify a topic and returns all entries beloging to that category (a title and an item number are displayed for each match.

• info(item_number) - which allows an item number to be specified and returns details such as number of items in stock and cost

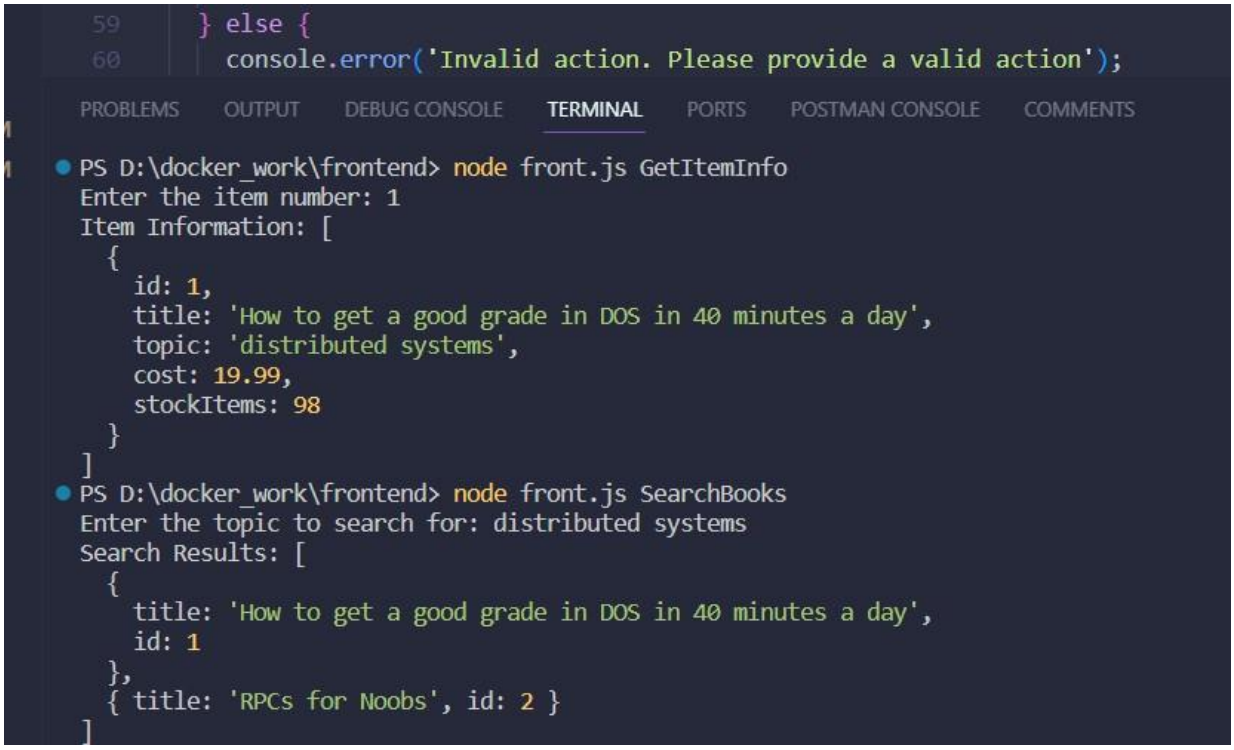
• purchase(item_number) - which specifies an item number for purchase.

It contains three function , each function make request

**Dependencies used:**

- Axios: to make http request
- Readline: to read input from the command line.

**Here are some result from the front end server:**



# LAB2

In this part we have completed the project by using the Caching and replication here are some results

For the catalog server , Ihave created the cach middleware , using yhe node-cache library to handel the cacheng Process , it checkd the cached data before handling the routes , here are the code :

```
const Cache = require('node-cache');

const db = new sqlite3.Database(dbFilePath, (err) => {
  if (err) {
    console.error('Error opening database:', err);
  } else {
    console.log('Connected to the database');
  }
});
const cache = new Cache();

app.use(express.json());

const cacheMiddleware = (req, res, next) => {
  const cacheKey = req.url;
  const cachedData = cache.get(cacheKey);
  if (cachedData) {
    console.log('Data found in cache');
    return res.json({ books: cachedData });
  }

  next();
};
```

This used when we handel a route , first we check in the cach if the data is already stored , if its not , then the data is fetched from database , then stored in cache , if we repaet the request , the data will be stored in it , no need to fetch from the data base another time.

```
app.get('/search/subject/:topic', cacheMiddleware, (req, res) => {
  const { topic } = req.params;
  const query = 'SELECT title, id FROM books WHERE topic = ?';
  db.all(query, [topic], (err, rows) => {
    if (err) {
      res.status(500).json({ error: 'An error occurred while querying the database' });
    } else {
      // Store the fetched data in the cache above
      cache.set(req.url, rows);
      console.log('Data added to cache');
      res.json({ books: rows });
    }
  });
});
```

This is the catalog server , same thing made in the ordder server :

```
const cacheMiddleware = async (req, res, next) => {
  const { item_number } = req.params;
  const cacheKey = `item_${item_number}`;
  const cachedItem = cache.get(cacheKey);

  if (cachedItem) {
    console.log('Item found in cache');
    return res.json({ item: cachedItem });
  }

  try {
    const catalogResponse = await axios.get(`http://localhost:7001/search/item/${item_number}`);
    console.log('Catalog server response:', catalogResponse.data);
    const item = catalogResponse.data.books[0];

    if (!item || item.stockItems === 0) {
      return res.status(404).json({ error: 'Item not available or out of stock' });
    }

    const updatedStock = item.stockItems - 1;
    const updateResponse = await axios.put(`http://localhost:7001/updateStock/${item_number}`, {
      stockItems: updatedStock,
    });

    if (updateResponse.status === 200) {
      cache.set(cacheKey, item);
      console.log('Catalog server response after update:', item);
      return res.json({ item });
    } else {
      return res.status(500).json({ error: 'Failed to update stock' });
    }
  } catch (error) {
    console.error('Error during purchase:', error.message);
    return res.status(500).json({ error: 'Internal server error' });
  }
```

```
app.post('/purchase/:item_number', cacheMiddleware);

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

The front end server : if we try to find item by its id , it will give this result ,, for the first time it will be add to the cach
If we repeat the same request , it will befounded in it :

```
]
PS D:\docker_work\frontend> node front.js GetItemInfo
Enter the item number: 2
Item Information: [
  {
    id: 2,
    title: 'RPCs for Noobs',
    topic: 'distributed systems',
    cost: 78.99,
    stockItems: 138
  }
]
PS D:\docker_work\frontend> node front.js GetItemInfo
Enter the item number: 2
Item Information: [
  {
    id: 2,
    title: 'RPCs for Noobs',
    topic: 'distributed systems',
    cost: 78.99,
    stockItems: 138
  }
]
```

```
Data found in cache
PS D:\docker_work\catalog> node catalog
Catalog server is running on port 7001
Connected to the database
Data added to cache
Data added to cache
Data found in cache
```

## Replication:

**The code provided represents a simple load balancer using Node.js and Axios to handle incoming HTTP requests. The method used here is handleRequest, which manages incoming requests and distributes them across multiple servers defined in the serverEndpoints array.**

```
alance > JS load.js > ...
const axios = require('axios');

const serverEndpoints = [
  'http://localhost:8002', // Order server
  'http://localhost:7001', // Catalog server
];
let currentIndex = 0;

const handleRequest = async (req, res) => {
  const currentServer = serverEndpoints[currentIndex];

  try {
    const response = await axios.get(currentServer + req.url);
    res.writeHead(200, { 'Content-Type': 'application/json' });
    res.end(JSON.stringify(response.data));

    currentIndex = (currentIndex + 1) % serverEndpoints.length;
  } catch (error) {
    console.error('Error proxying request:', error.message);
    res.writeHead(500, { 'Content-Type': 'text/plain' });
    res.end('Internal Server Error');
  }
};

module.exports = { handleRequest };
```