An-Najah National University
Faculty of Engineering and IT

جامعة النجاح الوطنية
كلية الهندسة وتكنولوجيا المعلومات

Distributed Operating Systems

10636456

1st Semester | 2025-2026

Bazar-com Project | Part 1

Jana Moe'n Tahayni
12220650

Bazar.com, a small multi-tier online bookstore. The system follows the multi-tier architecture: a two-tier architecture with a **frontend service** and a **backend tier** that consists of a **catalog service** and an **order service**, all exposed via HTTP REST interfaces.

The store manages four books, each with an ID, title, topic, price, and quantity in stock.

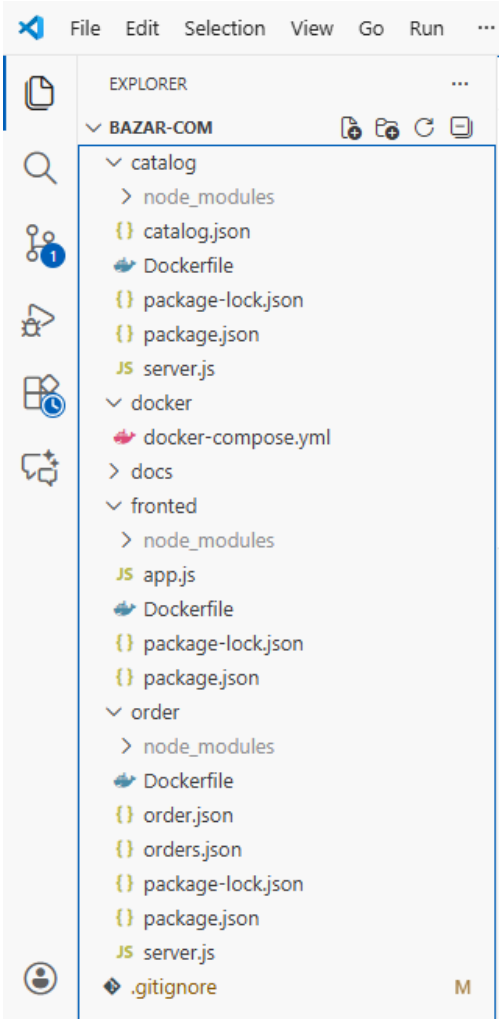
The system is implemented using **Node.js** and **Express**. It follows a microservices-style architecture with three independent services:

1. **Frontend service (port 3000)**
    - Entry point for clients.
    - Provides REST endpoints: /search/:topic, /info/:id, /purchase/:id.
    - Delegates work to the catalog and order services via HTTP calls using axios.
2. **Catalog service (port 4000)**
    - Manages the book catalog.
    - Stores data in catalog/catalog.json.
    - Exposes:
        - GET /search/:topic
        - GET /info/:id
        - PATCH /update/:id
        - GET /books (development/debug endpoint)
3. **Order service (port 5000)**
    - Manages purchase requests and order history.
    - Stores data in order/orders.json.
    - Exposes:
        - POST /purchase/:id
        - GET /orders
        - GET /orders/:id

Each service has its own **Dockerfile**, and the docker-compose.yml file in the docker directory builds and runs all three containers. The Docker setup also mounts the service

directories as volumes so that persistent JSON files are shared between the containers and the host.

The following photo shows the hierarchy of the system files:



Now, for the components of each service and how it works:

1) Catalog Service.
   The data is stored at **catalog.json** file, I designed two functions [load, save] for loading the books from the catalog file and to store any updated.

```
// Loading the catalog data from catalog.json file
const loadCatalog = () => {
    try {
        const data = fs.readFileSync(path.join(__dirname, 'catalog.json'), 'utf8');
        return JSON.parse(data);
    } catch (err) {
        console.error('Error loading catalog:', err);
        return [];
    }
};


const saveCatalog = (catalog) => {
    try {
        fs.writeFileSync(path.join(__dirname, 'catalog.json'), JSON.stringify(catalog, null, 2));
    } catch (err) {
        console.error('Error saving catalog:', err);
    }
};
```

This service is listening on port 4000.

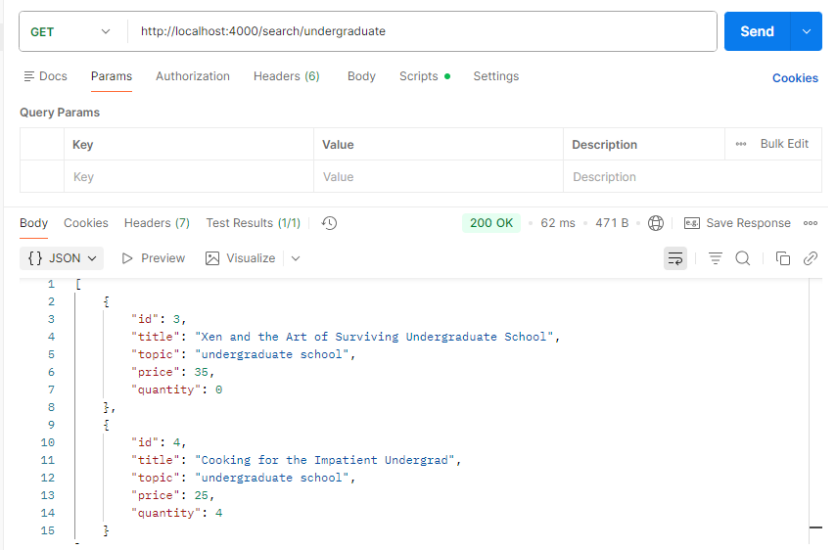
```
const PORT = 4000;
app.listen(PORT, () => console.log(`📚 Catalog service running on port ${PORT}`));

```

For the operations done through Catalog service:

- GET /search/:topic

```
// search by topic => search/topic
app.get('/search/:topic', (req, res) => {
    const topic = req.params.topic.toLowerCase();
    const catalog = loadCatalog();

    const results = catalog.filter(book =>
        book.topic.toLowerCase().includes(topic)
    );

    console.log(`📚 Search for "${topic}": Found ${results.length} books`);
    res.json(results);
});

```

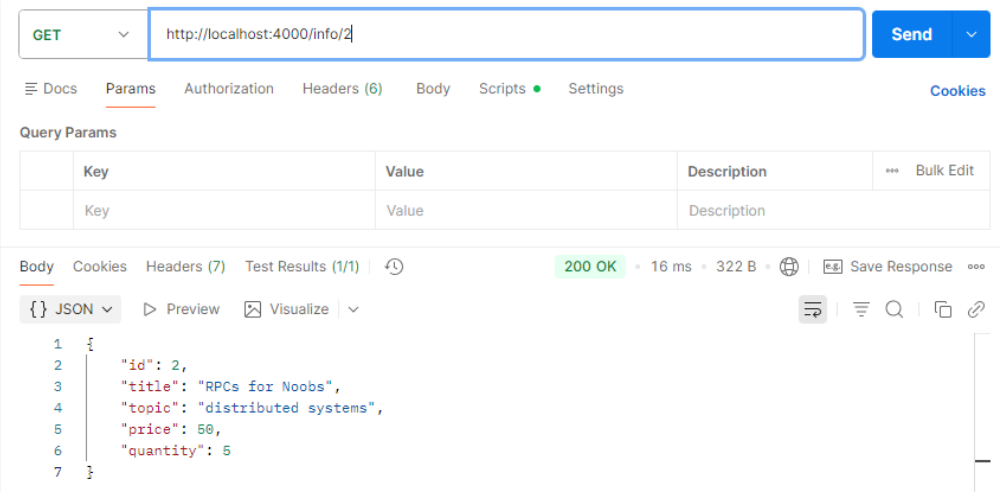
The result for **“https://localhost:4000/search/undergraduate”**



- GET /info/:id

```
// search by id => info/id
app.get('/info/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const catalog = loadCatalog();

    const book = catalog.find(b => b.id === id);

    if (!book) {
        return res.status(404).json({ error: 'Book not found' });
    }

    console.log(`ℹ️ Info request for book ID: ${id} - "${book.title}"`);
    res.json(book);
});
```

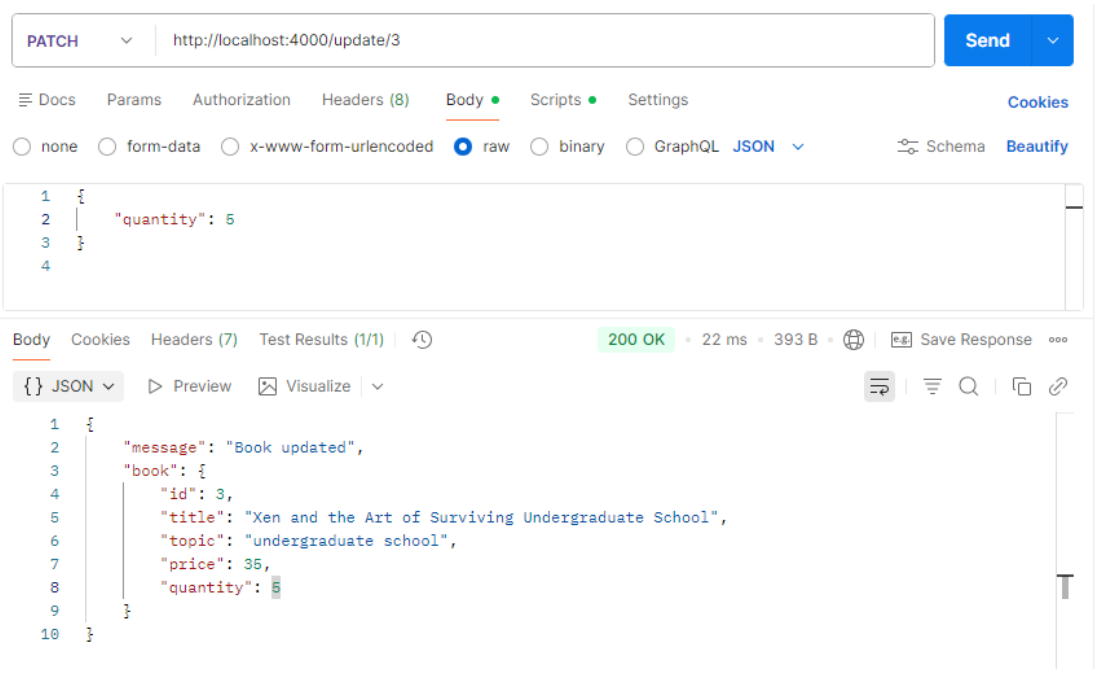
The result for **“http://localhost/4000/info/2”**



- PATCH /update/:id

```
// update existing book, quantity or price
app.patch('/update/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const { quantity, price } = req.body;

    let catalog = loadCatalog();
    const bookIndex = catalog.findIndex(b => b.id === id);

    if (bookIndex === -1) {
        return res.status(404).json({ error: 'Book not found' });
    }

    if (quantity !== undefined) {
        catalog[bookIndex].quantity = quantity;
    }
    if (price !== undefined) {
        catalog[bookIndex].price = price;
    }

    saveCatalog(catalog);

    console.log(`🔄 Updated book ID: ${id} - Quantity: ${catalog[bookIndex].quantity}, Price: ${catalog[bookI
    res.json({ message: 'Book updated', book: catalog[bookIndex] });
});
```

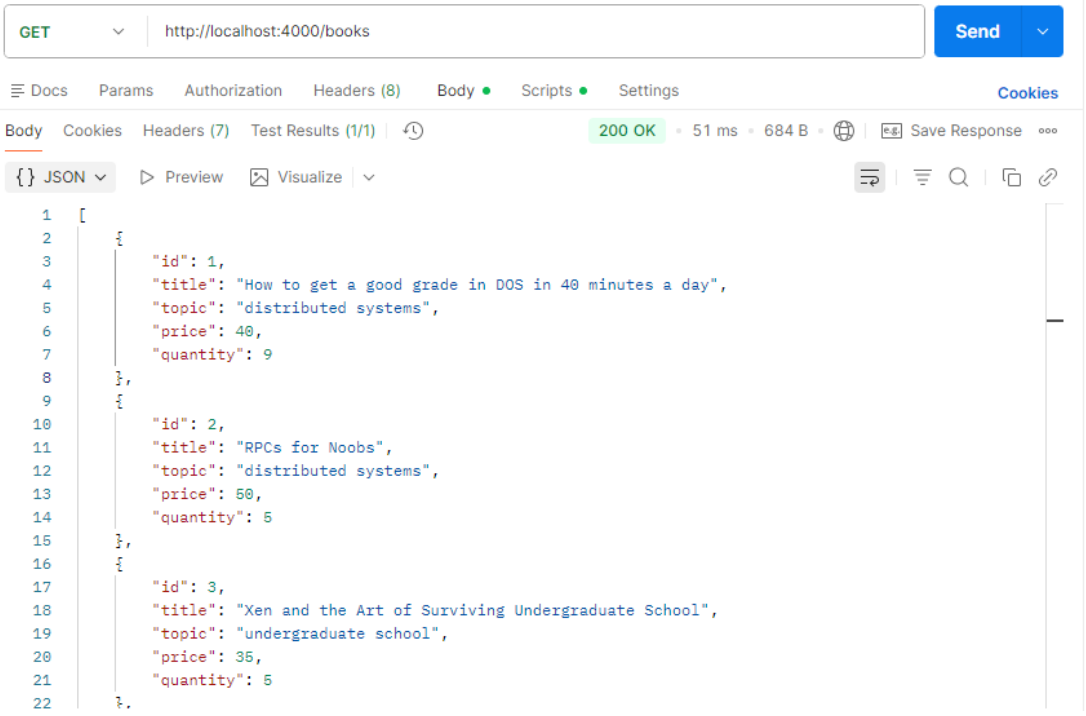
The result for **“http://localhost/4000/update/3”**



- GET /books

```
// get all books
app.get('/books', (req, res) => {
    const catalog = loadCatalog();
    res.json(catalog);
});
```

The result for **“http://localhost/4000/books”**



2) Order Service.
   The orders data stored in **orders.json** file. On startup, the service calls loadOrders() function to read any previous orders from orders.json file and the function saveOrders() is called when new order is created.

   The service is listening on port 5000.

```
100   const PORT = 5000;
101   app.listen(PORT, () => console.log(`🛒 Order service running on port ${PORT}`));
```

   The service is connected to catalog service through REST
   - GET {CATALOG_URL}/info/:id to read the current book state.
   - PATCH {CATALOG_URL}/update/:id to decrement quantity when a purchase succeeds.

```
8   const CATALOG_URL = process.env.CATALOG_URL || 'http://localhost:4000';   // catalog url
```

   Operations:
   - POST /purchase/:id

Steps:

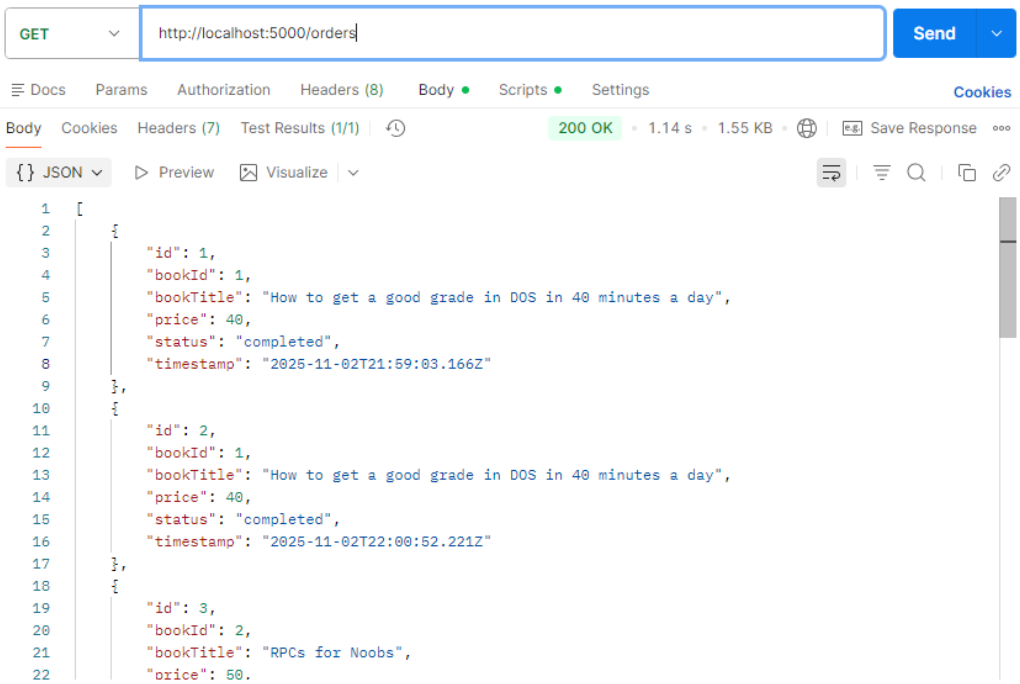
1. Parse bookId from req.params.id.
2. Call the catalog's /info/:id to get current quantity and price.
3. If quantity <= 0, return an error: book out of stock.
4. Otherwise, compute newQuantity = quantity - 1 and send a PATCH /update/:id to the catalog.
5. Create a new order object with:
   - id
   - bookId
   - bookTitle
   - price
   - status: 'completed'
   - timestamp
6. Append the order to the in-memory orders array and call saveOrders(orders) to persist.
7. Return a JSON object including the order and the remaining quantity.

The order service thus ensures that every purchase: checks stock, updates the catalog, logs an order entry.

- GET /orders

```
86   // get all orders list
87   app.get('/orders', (req, res) => {
88       res.json(orders);
89   });
```

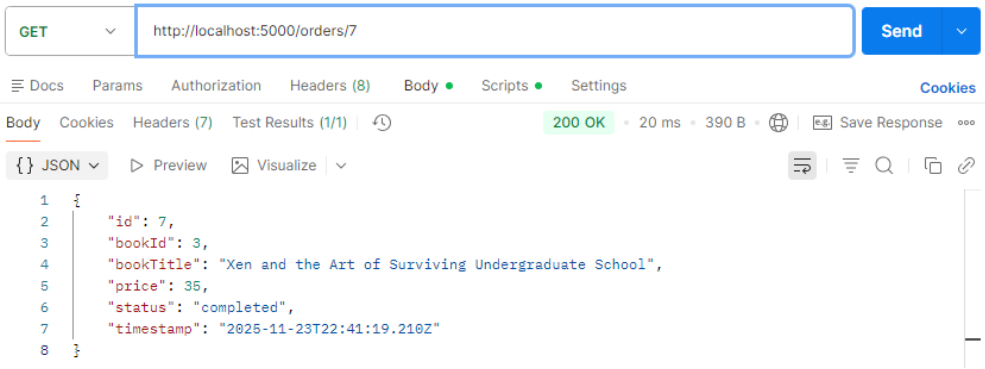
The result of **“http://localhost/5000/orders”**



- GET /orders/:id

```
91   // search for specific order by id
92   app.get('/orders/:id', (req, res) => {
93       const order = orders.find(o => o.id === parseInt(req.params.id));
94       if (!order) {
95           return res.status(404).json({ error: 'Order not found' });
96       }
97       res.json(order);
98   });
```

The result of “http://localhost/5000/orders/7”



3) Frontend Service.

The frontend service acts as a gateway for clients. It hides the internal addresses of catalog and order services and exposes a simpler API:

```
6   const CATALOG = process.env.CATALOG_URL || 'http://localhost:4000';
7   const ORDER = process.env.ORDER_URL || 'http://localhost:5000';
```

This service is listening on port 3000.

```
81  const PORT = process.env.PORT || 3000;
82  app.listen(PORT, () => console.log(`🎯 Frontend service running on port ${PORT}`));
```

Operations:

- GET /search/:topic

```
9   // search by topic
10  app.get('/search/:topic', async (req, res) => {
11    try {
12      const topic = req.params.topic;
13      console.log(`🔍 Frontend: Search request for topic: ${topic}`);
14
15      const response = await axios.get(`${CATALOG}/search/${encodeURIComponent(topic)}`);
16
17      console.log(`✅ Frontend: Search successful, found ${response.data.length} books`);
18      res.json(response.data);
19
20    } catch (err) {
21      console.error('❌ Frontend: Search error:', err.message);
22      res.status(500).json({ error: 'Internal server error' });
23    }
24  });
```

Logs the incoming request, calls GET {CATALOG}/search/:topic. Then, forwards the JSON response back to the client.

Samples of output will be on the other file of documentation.

- GET /info/:id

```
26  // search by id
27  app.get('/info/:id', async (req, res) => {
28    try {
29      const id = req.params.id;
30      console.log(`ℹ️ Frontend: Info request for book ID: ${id}`);
31
32      const response = await axios.get(`${CATALOG}/info/${id}`);
33
34      console.log(`✅ Frontend: Info retrieved for: "${response.data.title}"`);
35      res.json(response.data);
36
37    } catch (err) {
38      console.error('❌ Frontend: Info error:', err.message);
39
40      if (err.response && err.response.status === 404) {
41        return res.status(404).json({ error: 'Book not found' });
42      }
43
44      res.status(500).json({ error: 'Internal server error' });
45    }
46  });
```

Calls GET {CATALOG}/info/:id. If the catalog returns 404, the frontend returns 404 to the client. Otherwise, returns the book details.

- POST /purchase/:id

```
48  // purchase book
49  app.post('/purchase/:id', async (req, res) => {
50    try {
51      const id = req.params.id;
52      console.log(`🛒 Frontend: Purchase request for book ID: ${id}`);
53
54      const response = await axios.post(`${ORDER}/purchase/${id}`);
55
56      console.log(`✅ Frontend: Purchase successful - Order ID: ${response.data.order.id}`);
57      res.json(response.data);
58
59    } catch (err) {
60      console.error('❌ Frontend: Purchase error:', err.message);
61
62      if (err.response) {
63        return res.status(err.response.status).json(err.response.data);
64      }
65
66      res.status(500).json({ error: 'Internal server error' });
67    }
68  });
```

Calls POST {ORDER}/purchase/:id, forwards success or error responses (e.g., out-of-stock) back to the client.

Docker Deployment:

The project was containerized using Docker. Each microservice (catalog, order, and fronted) contains its own **Dockerfile** that defines how the service is built and executed inside an isolated environment. All three services are orchestrated using a docker-compose.yml file located in the *docker* directory of the project.

Based on the configuration in the docker-compose.yml file, Docker Compose automatically builds and runs each microservice from its own folder (catalog, order, and fronted). Each service is started inside a separate container using the Dockerfile found in its directory. The file also defines the dependencies between services: the frontend service depends on both the catalog and the order services, and the order service depends on the catalog. This ensures that the backend services are fully running before the frontend starts, allowing correct communication between all parts of the system.

docker-compose.yml

```yaml
version: "3.9"

services:
  catalog:
    build:
      context: ../catalog
      dockerfile: Dockerfile
    ports:
      - "4000:4000"
    volumes:
      - ../catalog:/app

  order:
    build:
      context: ../order
      dockerfile: Dockerfile
    ports:
      - "5000:5000"
    volumes:
      - ../order:/app
    environment:
      - CATALOG_URL=http://catalog:4000
    depends_on:
      - catalog

  fronted:
    build:
      context: ../fronted
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    volumes:
      - ../fronted:/app
    environment:
      - CATALOG_URL=http://catalog:4000
      - ORDER_URL=http://order:5000
    depends_on:
      - catalog
      - order
```

How to run my project using docker:

1- Open docker desktop.
2- Open the terminal on docker folder.
3- Build and start containers => **docker-compose up –build**