Bazar.com Design Document

D. Samer Arandi

Rand Johari	**12027653**
Abeer Kharouf	**12028125**

1. Overview of the Program Design

We designed and implemented **Bazar.com**, a lightweight multi-tier online bookstore, using the microservices architecture. Our system is split into three independent services:

- **Catalog Service**: Maintains book data such as title, topic, quantity, and price.
- **Order Service**: Handles purchase requests by checking stock availability and updating book data.
- **Frontend Service**: Provides the interface through which users interact with the system.

Each service runs in a separate Docker container and communicates via RESTful HTTP requests. This design promotes modularity, simplicity, and distributed deployment.

---

2. How the System Works

When a user sends a request, the process flows through the following path:

- **Search Request**: The user queries by topic via the Frontend Service. The Frontend calls the Catalog Service's /search/:topic endpoint and returns the matched book list. -Get method-
- **Info Request**: The user requests book details by ID through the Frontend Service. It relays the request to the Catalog Service's /info/:id endpoint and returns the response. -Get method-
- **Purchase Request**: The user initiates a purchase via the Frontend Service. The Frontend sends a POST request to the Order Service's /purchase/:id endpoint. The Order Service then: -post method-
    1. Sends a GET request to the Catalog Service to retrieve book info.
    2. Verifies stock availability.
    3. Send a PUT request to the Catalog Service to update the quantity.

Log messages such as "bought book RPCs for Noobs" are printed during successful purchases to reflect proper functionality.

---

3. Design Trade-offs and Justifications

a. CSV vs. Database

We used a **CSV file** to store book data instead of a database. This simplifies deployment and meets lab requirements, avoiding the complexity of managing SQL databases for a small-scale project.

b. HTTP Method Choices

- We used POST for the /purchase/:id route, as a purchase is an action that changes the system state and creates a new transaction.
- PUT is used for /update/:id since it updates existing resources.

c. Microservice Boundaries

Each service has a **single responsibility**, helping isolate changes, scale independently, and enable easier debugging.

---

4. Possible Improvements & Future Extensions

- **Switch from CSV to SQLite**: A lightweight database like SQLite can offer more robustness and concurrency control.
- **Order Logging**: Implement persistent order logging to track all purchases.
- **Frontend Enhancement**: Introduce a simple UI using HTML/CSS to make interaction easier than using Postman.
- **Input Validation**: Add validation for topics, item IDs, and quantities to prevent invalid requests.
- **Concurrency Handling**: Add locking or transactions to handle simultaneous purchases safely.

---

5. Known Issues and Limitations

- Currently, there is **no race condition protection** in the purchase logic. If two users buy the same book simultaneously, the stock might not update correctly.

- **No frontend UI**: The system uses only API calls; GUI features are not implemented.
- **No authentication or security**: All endpoints are publicly accessible, which is acceptable for the lab scope but not production-level.

---

6. Running the Application

Requirements:

- Docker and Docker Compose must be installed.

Steps:

1. Clone the repository from GitHub.
2. Navigate to the project root directory.
3. Run the system:

```
docker-compose up --build
```

Accessing the Services:

- **Frontend Service** (on port 3002):
  - **Search:** http://localhost:3002/search/distributed systems
  - **Info:** http://localhost:3002/info/2
  - **Purchase:** http://localhost:3002/purchase/2
  - **Update (Internal):** curl -X PUT http://catalog:3000/update/2 \ -H "Content-Type: application/json" \ -d '{"quantity": 4, "price": 40 }'
    *Log messages will be printed to each service's terminal to show detailed operations.