DOS Project (Part 1)
Bazar.com

A Report by:
Bakr Yaish & Hamza Younes

GitHub

https://github.com/Baker-Yaeesh/books-bazar

Section 1 – Overall Program Design

The Books Bazar system is a small online bookstore built using a multi-tier, microservices-based architecture. The system is divided into two main layers: a Frontend Service, which handles user interaction, and a Backend Tier consisting of the Catalog Service and Order Service. This structure keeps the system modular, easy to maintain, and capable of running as distributed components.

The Frontend Service is a lightweight Flask application that serves as the public entry point. It accepts user requests—such as searching for books, retrieving book details, or making a purchase—and forwards them to the appropriate backend service. The frontend performs only simple validation and routing, keeping business logic inside the backend microservices.

The Catalog Service manages all book data, including ID, title, topic, price, and stock. It exposes REST endpoints for searching by topic, getting detailed information for a book, and updating inventory or pricing. The catalog is stored in a file that is loaded at startup and updated whenever changes occur. A thread lock is used to ensure safe, consistent access to the file during concurrent operations.

The Order Service handles purchases. To process an order, it requests the item's information from the Catalog Service, verifies that stock is available, and—if successful—decrements the stock through the Catalog Service. It also records each completed purchase in a persistent order log. Like the Catalog Service, it uses a thread lock to prevent corruption when writing to the log. The service does not keep its own copy of catalog data; it always relies on the Catalog Service as the source of truth.

All communication between services uses HTTP REST with JSON, allowing each service to run independently and be updated without affecting the others, as long as their API contracts are maintained.

For deployment, all three services run in separate Docker containers, coordinated with Docker Compose. The services share an internal network so they can reach each other by name (e.g., catalog-service). This setup allows the entire system to be started with a single command, making deployment and testing straightforward.

Section 2 – How the Program Works

The Books Bazar system operates through coordinated interactions among three microservices, with each component responsible for a specific part of the workflow. This section explains, in practical terms, how the system behaves from the moment a user makes a request to the moment a response is returned.

1) User Interaction via the Frontend

All user operations begin at the Frontend Service, which exposes three primary HTTP endpoints

- GET /search/<topic>
- GET /info/<item_id>
- POST /purchase/<item_id>

The frontend parses and validates user requests, then forwards them to the backend, doing no business logic to stay simple and stateless.

2) Searching for Books

When a user makes a search request:

- Frontend receives the search query.
- Request is forwarded to Catalog Service using a REST call to GET /search/<topic>.
- Catalog Service returns all books matching the topic.
- frontend passes this list directly back to the user.

Section 2 – How the Program Works (cont…)

3) Retrieving Book Information

- Frontend receives GET /info/<item_id>.
- Request is forwarded to Catalog Service's GET /info/<item_id> endpoint.
- Catalog Service returns a JSON object containing the book's title, price, and current stock.
- Frontend relays the response back to the user.

4) Purchasing a Book

Purchasing is the most complex workflow, as it is coordinated between the Order and Catalog services:

- User sends: POST /purchase/<item_id>
- Frontend forwards the request to the Order Service.
- Order Service verifies stock by calling GET /info/<item_id> on the Catalog Service
- If stock is available Order Service sends POST /update/<item_id> with {"stock_delta": -1} to decrement inventory. Order Service writes a purchase entry to its order log file.
- If stock is not available Order Service immediately returns an "out of stock" error.

5) Persistence and Thread Safety

- Catalog Service stores book metadata.
- Order Service stores order logs

6) Distributed Execution Through Docker

At runtime, each service operates inside its own Docker container. Docker Compose:

- Starts all services and Connects them on a shared internal network
- Allows them to discover each other by service name (e.g., catalog-service:5001)
- Ensures the system can be run with a single command (docker-compose up)

Section 3 – Design Tradeoffs

The design of the Books Bazar system required several tradeoffs between simplicity, correctness, performance, and scalability. Because this project is small and educational, the design favors clarity and ease of implementation over advanced optimizations.

Microservices vs Monolith

The bookstore is split into three microservices instead of a single program.

Pros

clear separation of concerns, independent updates, and realistic distributed design.

Cons

more deployment complexity, extra networking overhead, and inter-service failures becoming possible.

Thread Locks vs Performance

Because data is stored in files, each service uses a threading.Lock for safe reads and writes. This prevents corruption but also forces the service to handle updates one at a time, reducing concurrency. This is acceptable for a small demo but would not scale under real load.

Synchronous REST Calls vs Asynchronous Design

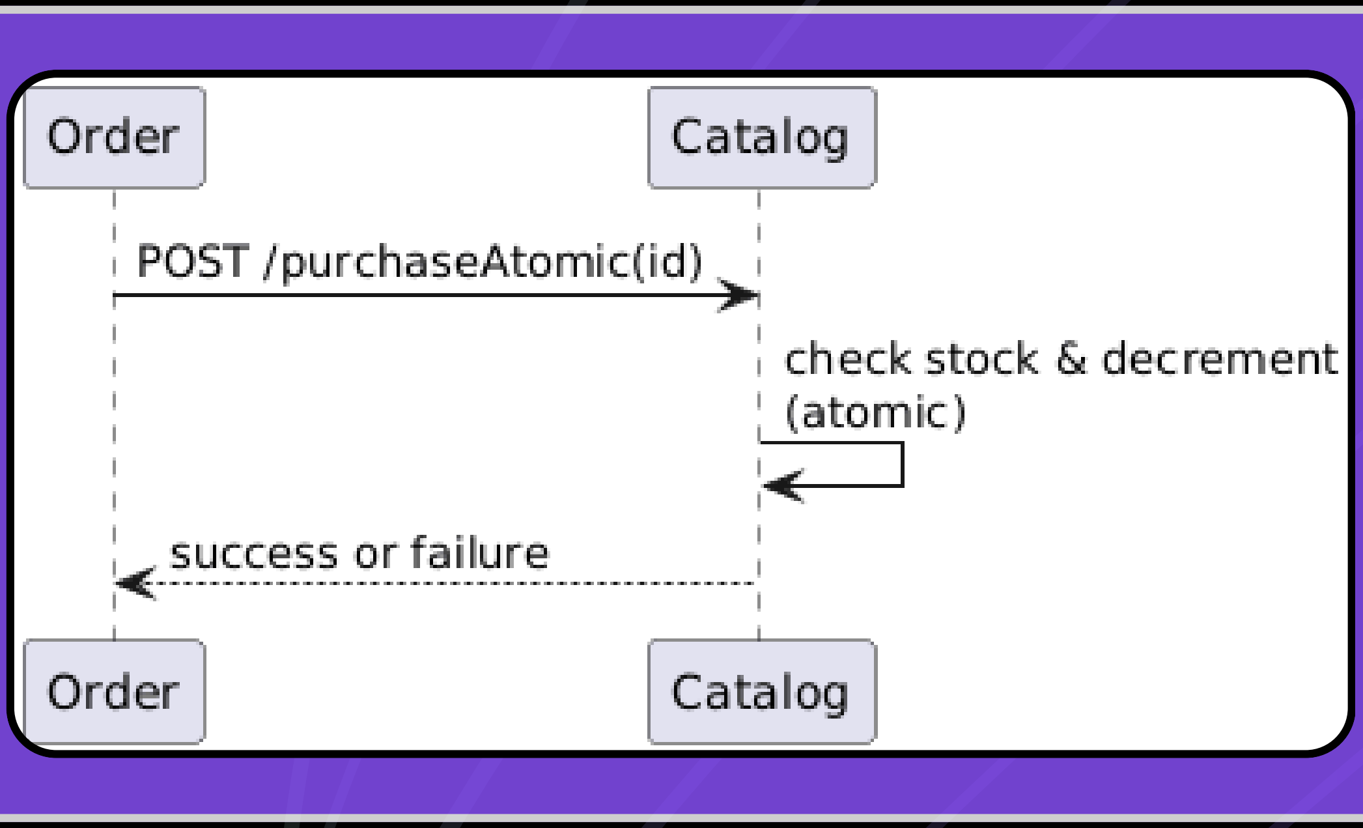
Services communicate using blocking REST calls, making the workflow easy to understand and debug. The tradeoff is that one slow service delays others, and no retries or fault tolerance exist. An asynchronous or message-queue design would be more robust but was beyond the project scope.

Section 4 – Improvements & Extensions

The Books Bazar system works well as a small educational microservice application, but several improvements can make it more robust, scalable, and feature-rich. Below are the most important enhancements and how they might be integrated.

Atomic Purchase Operation

The purchase process currently checks stock then decrements it. This creates a small race condition. Improving this requires combining both into one atomic endpoint in the Catalog Service.



Improved Caching for Frequent Reads

Search and book-info requests are read-heavy. Introducing a simple in-memory cache (e.g., Python dict, Redis, or Flask caching) would speed up repeated requests.

Add a Shopping Cart Microservice

A more realistic bookstore includes carts, not single-item purchases. A new Cart Service can track user items before checkout.

Section 5 Limitations

several known limitations and edge cases can cause incorrect behavior, especially under concurrent or unexpected conditions

Race Condition During Purchases

During purchases, the system checks and updates stock in two separate steps, two simultaneous purchase requests can both see “1 item left” and both decrement it. This may cause:

- Double-selling an item
- Negative stock
- Out-of-sync catalog data

This flaw is rare but inherent in the non-atomic two-step design.

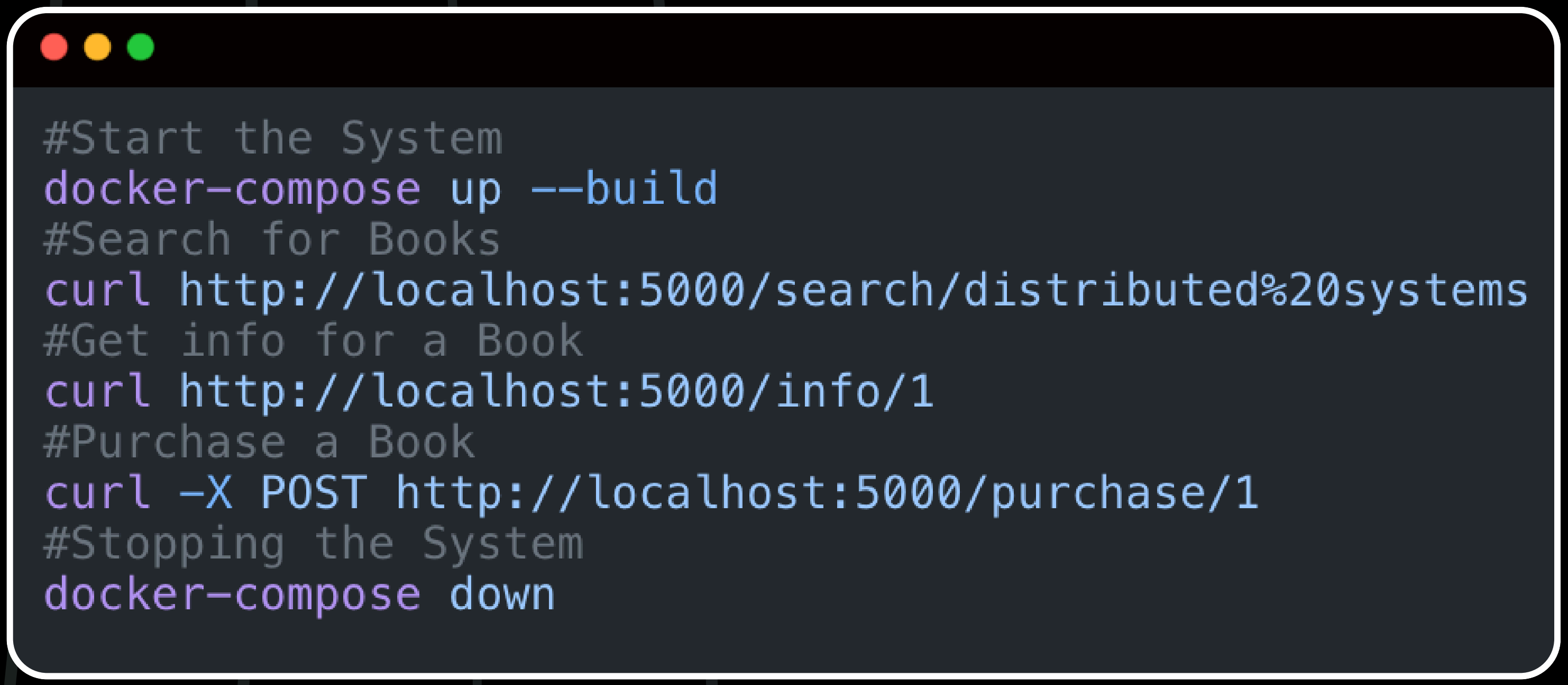
Service Fault Tolerance Issues

If one microservice becomes slow or fails, the whole system can hang because there are no retries, timeouts, or failover mechanisms—an important limitation in distributed systems.

Section 6 How to Run

Running the system is straightforward because all three services (Frontend, Catalog, Order) are packaged and orchestrated using Docker and Docker Compose

Run the following commands inside the project folder; when the containers start, the Frontend is available on localhost:5000 for all bookstore operations.



```
#Start the System
docker-compose up --build
#Search for Books
curl http://localhost:5000/search/distributed%20systems
#Get info for a Book
curl http://localhost:5000/info/1
#Purchase a Book
curl -X POST http://localhost:5000/purchase/1
#Stopping the System
docker-compose down
```