Bazar.com Project Documentation

Bassam Tuffaha
University ID: 11924847

November 5, 2024

1 Introduction

Bazar.com is an online bookstore that consists of a two-tier web architecture with multiple microservices, each dedicated to handling specific tasks. This document provides an overview of the project, its components, and technical details regarding its implementation. The system includes a front-end service, a catalog service, and an order service. The front-end service handles user requests, the catalog service manages book information, and the order service manages purchase orders.

2 Overall Program Design

The Bazar.com system is designed using a microservice architecture, implemented with the Flask framework in Python. It consists of the following components:

- **Front-end Service**: Acts as an API gateway, forwarding user requests to the appropriate backend services.
- **Catalog Service**: Maintains the catalog of books, including details such as the title, topic, quantity, and price.
- **Order Service**: Handles book purchases, updates stock in the Catalog Service, and maintains a record of all orders placed.

Each service communicates using RESTful APIs. The system uses SQLite for data storage, providing simplicity and ease of use for the limited scope of this project.

3 Detailed Description of How It Works

The main functionalities provided by the Bazar.com system include the following endpoints:

3.1 Front-end Service Endpoints

- `/search/<topic>` (GET): Forwards the request to the Catalog Service to find books matching the given topic.
- `/info/<item_id>` (GET): Retrieves detailed information about a specific book by forwarding the request to the Catalog Service.
- `/purchase/<item_id>` (PUT): Initiates a purchase by forwarding the request to the Order Service.
- `/orders` (GET): Retrieves all orders by forwarding the request to the Order Service.

3.2 Catalog Service Endpoints

- `/search/<topic>` (GET): Returns all books belonging to the specified topic.
- `/info/<item_id>` (GET): Returns detailed information about a specific book, such as title, quantity, and price.
- `/update/<item_id>` (PUT): Updates the quantity or price of a book.

3.3 Order Service Endpoints

- `/purchase/<item_id>` (PUT): Initiates a purchase for a specified book ID by verifying availability, updating the Catalog Service, and recording the order.
- `/orders` (GET): Retrieves all orders that have been placed.

4 Design Trade-offs Considered and Made

- **Database Selection**: SQLite was chosen for its simplicity and suitability for small-scale applications like Bazar.com. Although it does not offer the scalability of more advanced databases (e.g., PostgreSQL or MySQL), it is sufficient for this project given the limited scope and the number of items managed.
- **Data Consistency**: Data consistency was prioritized when interacting between services to avoid race conditions. Thread safety mechanisms were implemented using Flask's built-in capabilities to handle concurrent requests.
- **Microservice Communication**: RESTful APIs were used for inter-service communication, providing a simple and stateless mechanism for data exchange.

5 Possible Improvements and Extensions

- **Database Scalability**: Transitioning from SQLite to a more robust database, such as PostgreSQL, would support larger datasets and concurrent access. Adding replication and sharding strategies could improve data availability and load balancing across multiple nodes.
- **Microservices Scalability**: Enable horizontal scaling by replicating microservices across multiple instances. Using a load balancer can distribute requests evenly, ensuring high availability and responsiveness under heavy loads.
- **Security and Rate Limiting**: Implement user authentication to secure endpoints and apply rate limiting to prevent abuse. This protects the system from excessive requests and enhances security.
- **Monitoring and Logging**: Centralized logging and real-time health monitoring of services would enable proactive issue detection and streamlined debugging, ensuring system reliability.

6 Instructions on How to Run the System

1. **Prerequisites**: Install Docker and Docker Compose on your machine.
2. **Setup Steps**:
   - Ensure that the Docker Compose file (docker-compose.yml) is present in the project directory.
   - Run the following command to start all services (Front-end, Catalog, and Order):

     ```
     docker-compose up
     ```

   - Docker Compose will create and start the necessary containers for each service.
   - Use a tool like Postman or curl to test the service endpoints. Example request to purchase an item:

     ```
     curl -X PUT http://localhost:5002/purchase/4
     ```

7 Conclusion

The Bazar.com project demonstrates a simple implementation of a multi-tier online bookstore using microservices. By dividing responsibilities among the Front-end, Catalog, and Order services, the system maintains modularity and separation of concerns. The project meets the requirements of a small-scale bookstore, while future improvements can enhance scalability, security, and robustness.

Overall, the design and implementation of Bazar.com strike a balance between simplicity and functionality, making it suitable for learning and demonstrating basic distributed system principles.