2024

Distributed operating system

Lab 1: Bazar.com: A Multi-tier Online Book Store

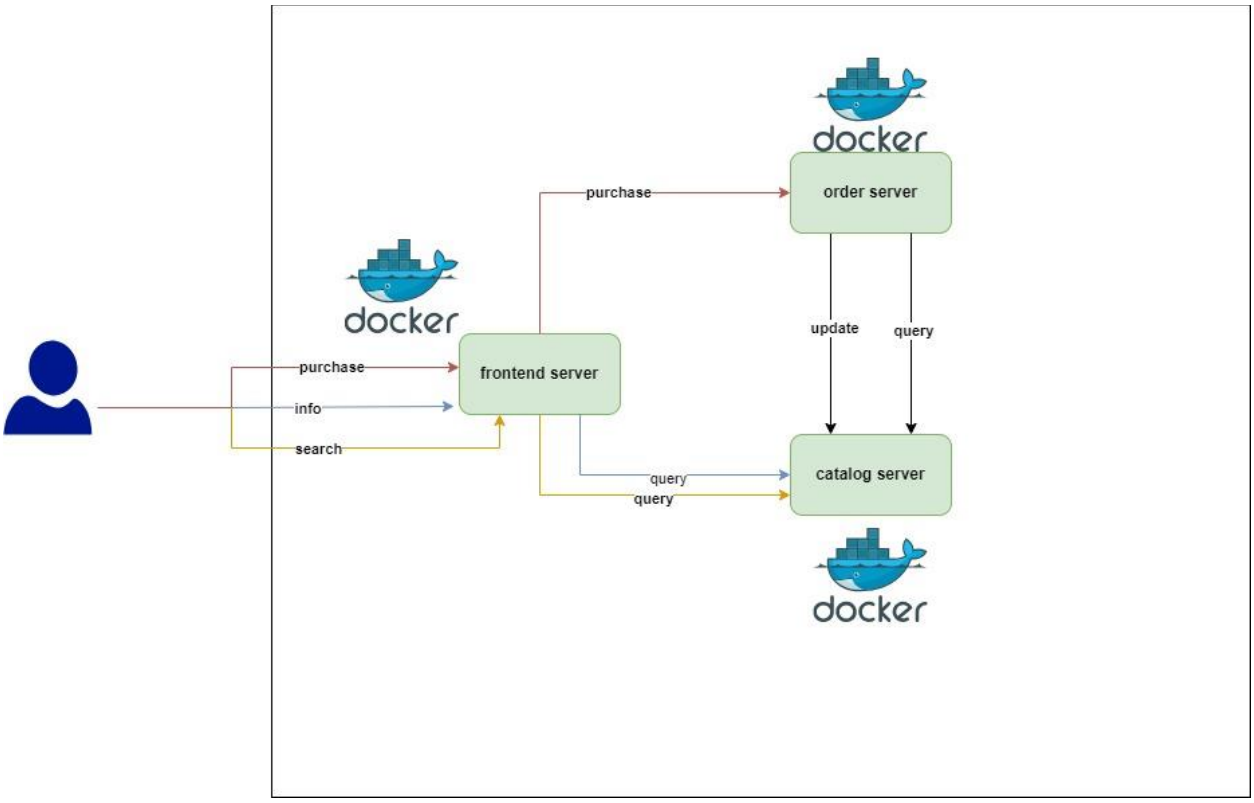
Naser Tayeh

Ameer mialeh

Objectives

The primary objective of this project is to design, implement, and deploy Bazar.com, the World's smallest bookstore, with a focus on efficient and scalable two-tier web architecture utilizing micro services. The specific goals include:

Overall design



Design Description

In this project, a containerized approach using Docker is adopted to facilitate the efficient deployment and management of the three crucial components: the **front-end server**, **order server**, and **catalog server**. Docker provides a lightweight and portable solution to package each microservice along with its dependencies into a standardized unit called a container. For each server, a dedicated Dockerfile is crafted, outlining the necessary steps to create a reusable and isolated environment.

Why do we need a container for each service ?

By encapsulating the application and its dependencies within containers, potential conflicts between different environments are mitigated, ensuring consistent behavior across various deployment scenarios.

What is the need for docker-compose ?

The use of Docker Compose further streamlines the orchestration of these containers by defining the services, network configurations, and interdependencies in a single, easily maintainable YAML file. This approach not only simplifies the deployment process but also enhances scalability, as additional instances of each microservice can be effortlessly spun up to meet growing demand

How the program work ?

The program operates seamlessly upon execution of the 'docker-compose build' command, initiating the orchestrated deployment of three distinct services within Docker containers.

1. **Front-End Server:**
   The front-end server, encapsulated within **frontend-container**, is designed to handle user requests for purchasing, information retrieval, and search operations. It operates on port 8000 and resides within its own local network in the Docker environment – this to make sure the 3 services is connected with each other - , having a it's own IP address. The front-end service, implemented as a microservice, plays a crucial role in the user interface, providing a seamless experience for customers interacting with Bazar.com. It communicates with both the catalog and order servers to fulfill user requests, establishing dependencies on these backend services for smooth operation. The service is initiated by the 'docker-compose up' command and gracefully stopped by 'docker-compose down', ensuring controlled lifecycle management ,note that this service is depend on order and catalog server , this mean this docker container will not run until it's sure that the order and catalog server is running .

2. **Order Server:**

   The order server, a distinct microservice running in order-container, specializes in processing purchase requests called from the front-end server. This service operates on port 8002 and is assigned its own IP address within the internal Docker network environment – this to make sure the 3 services is connected with each other -. The order server maintains a dependency on the catalog server, ensuring accurate verification of item availability before completing a purchase. Its functionality is critical in managing and updating the order database, contributing to the overall responsiveness and reliability of the book store's transactional processes. The service is initiated by 'docker-compose up' and gracefully stopped by 'docker-compose down'.,note that this service is depend on catalog server , this mean this docker container will not run until it's sure that the catalog server is running .

3. **Catalog Server:**

The catalog server, implemented as a microservice within catalog-container, serves as the backbone for processing both purchase and query requests. Operating on port 8001, this service has its own IP address and is part of an internal Docker network. The catalog server is responsible for accessing an SQLite database to retrieve and update book-related data, including stock for each book. It supports query operations, responding to requests based on either book name or book number. The front-end and order servers depend on the catalog server to ensure accurate and up-to-date information for user interactions, making it a central component in the overall functionality of Bazar.com. Similar to the other services, it is started with 'docker-compose up' and stopped with 'docker-compose down', providing a streamlined approach to the container lifecycle.

Design tradeoffs considered and made

- Microservices Architecture vs. Monolithic Design:
  The decision to adopt a microservices architecture introduces benefits such as modularity and scalability. However, it also introduces increased complexity in terms of service communication and potential overhead. The tradeoff was made to prioritize the advantages of microservices for flexibility and scalability while managing the associated complexities through effective communication patterns.

- Containerization with Docker:
  Choosing Docker for containerization provides portability and consistency across different environments. However, this decision comes with the tradeoff of increased resource consumption due to the need for separate containers for each service. The benefits of isolation and reproducibility, particularly in deployment, were deemed more significant than the incremental resource overhead.

- SQLite Database for Catalog Server:
  The selection of SQLite as the database management system for the catalog server involves a tradeoff between simplicity and scalability. While SQLite is lightweight and requires minimal setup, it may have limitations in handling extensive concurrent transactions compared to more robust database systems. This tradeoff was accepted to streamline development and deployment while recognizing that the book catalog is relatively small, making SQLite suitable for the task.

- Docker Compose for Orchestration:
  While Docker Compose simplifies the orchestration of multiple services, it introduces a tradeoff in terms of the learning curve for users unfamiliar with the tool. The decision to use Docker Compose was based on its ability to streamline the deployment process and manage the interdependencies between services efficiently, outweighing the initial learning curve.

Possible improvements and extensions

- **Introduction of Caching Mechanisms:**
  Implement caching mechanisms, both at the front end and the catalog server, to optimize response times. Caching frequently accessed data can reduce the load on backend services and enhance overall system responsiveness.

- **Integration of a Robust Database System:**
  Consider transitioning from SQLite to a more robust and scalable database system, especially if the book catalog is expected to grow significantly. Database options like MySQL or PostgreSQL could provide better support for concurrent transactions and scalability.

- **Security Enhancements:**
  implement security best practices, including data encryption, secure communication protocols (HTTPS), and regular security audits. This ensures the protection of user data and prevents potential vulnerabilities.

How we can run the program

using a docker-compose only

you will find this in readme.txt

using docker-compose just to build the images

here you can use these command , to build the images , then build each container sperately

run# docker-compose build // to build the 3 images

run# docker images // to show you 3 images

docker network create --subnet=172.18.0.0/16 internal-network

run# docker run -it -d -v .\frontend-server\src:/app/src -p 8000:8000 --network internal-network --ip 172.18.0.6 --name frontend-container testing-frontend-server

run# docker run -it -d -v .\order-server\src:/app/src -p 8002:8002 --network internal-network --ip 172.18.0.8 --name order-container testing-order-server

run# docker run -it -d -v .\catalog-server\src:/app/src -p 8001:8001 --network internal-network --ip 172.18.0.7 --name catalog-container testing-catalog-server

Conclusion

In conclusion, the design and implementation of Bazar.com have resulted in a microservices-based architecture that balances efficiency, scalability, and simplicity. Through careful consideration of design tradeoffs, the program successfully utilizes Docker containerization, microservices orchestration with Docker Compose, and a lightweight SQLite database for catalog management. The system, initiated with 'docker-compose up' and gracefully stopped with 'docker-compose down', provides a responsive and reliable platform for users to search, obtain information, and purchase books. While meeting current objectives, opportunities for improvement and extension, such as scaling strategies, feature enhancements, and security considerations, pave the way for future developments and the continued evolution of Bazar.com.