CS 677 S21 Lab 3 - Bookstore

Design Document

Submitted By

Jagriti Singhal, Bhumika Kalavadia

Date: 30 April, 2021

The project has been implemented in Java and can be deployed using a single bash script.

[1] Project Setup

The project has been implemented in Java using Ninja Framework and can be deployed using a single bash script: **run.sh**.

First we will start 5 containers - Frontend, Catalog1, Order1, Catalog2, Order2 using docker compose file. The port for each microservice is fixed: Frontend - 8080, Catalog1 - 8081 and Order1 - 8082, Catalog2 - 8083 and Order2 - 8084. Once all the containers are up and running, we will start our client.
The number of clients can be configured via the run.sh script. By default, 2 clients will start.

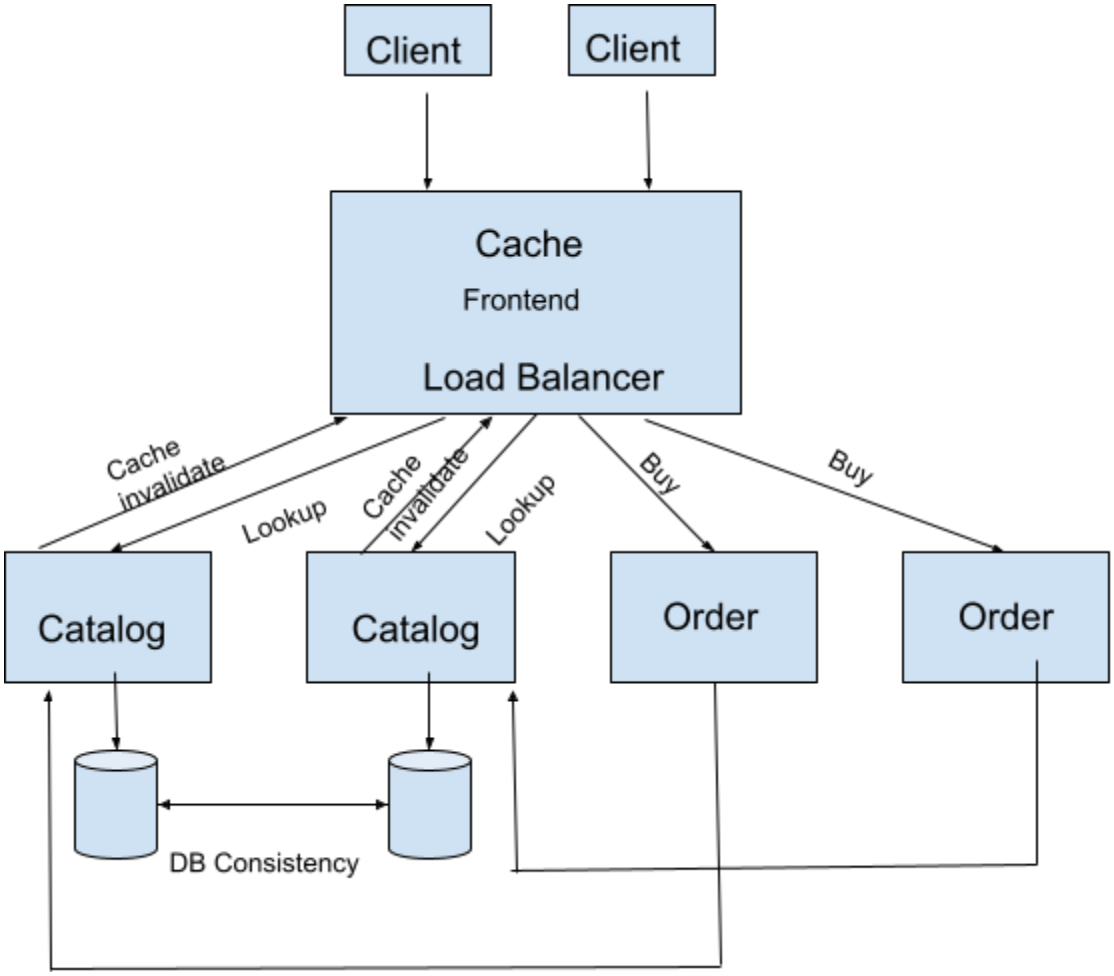
The hostnames for all services are specified in the hostname.conf file. For the first milestone, all hostnames point to localhost. The code will read the hostname.conf and pick the required hostname while making HTTP requests.
To start the system (5 containers and 2 clients) - **sh run.sh**
The above microservices are implemented as standalone maven modules in different containers. The client process will make the HTTP request: lookup, search or buy at random with random parameters to the front-end server and get the response as a JSON object.

Once this system runs for a fixed period of time, we will shut down the system by stopping the containers and deleting the images.

[2] Design and Implementation

This project has been built on top of Lab 2: Bookstore.
Lab2: Bookstore existing design in brief -

- It has been developed using Ninja which is based on MVC design pattern.
- It supports three microservices:
  Frontend
  The front end server supports three operations: search(topic) , lookup(item_number) and buy(item_number)
  Catalog
  The catalog server supports three operations: queryByItem(item_number), queryBySubject(topic), updateInventory(item_number), updateCost(item_number,cost)
  Order
  The order server supports a single operation: buy(item_number).



PART 1

1. **Load Balancing**

   Load balancing has been implemented on the frontend server.

   - It uses the **round robin** algorithm on a **per request basis**.
   - Since there are two replicas each of catalog and order microservices, the frontend server toggles between the two replicas.
   - When the frontend server receives a request from the client for lookup or search, it decides between the two catalog servers. Similarly, when frontend server receives a request for buy, it decides between the two order servers available and forwards the request

2. **Caching**

   Caching has been implemented on the frontend server.

   - **In-memory caching** has been implemented on the frontend server. No separate server has been used for caching.
   - It uses the **Least Recently Used (LRU)** caching scheme to replace entries in the cache. LinkedHashMap has been used to implement this. The size of the cache has been kept configurable. For every **cache hit**, the order of entries in cache is changed so that the least recently used entry can be removed.
   - Cache is only used for lookup and search requests
   - If a request cannot be served by cache (**cache miss**), the frontend server forwards the request to catalog server and caches the response for subsequent requests

3. **Cache Consistency**

   Server push technique has been used to **invalidate a cache entry.** Whenever the catalog server receives an update request (to update cost or inventory), it sends a cache invalidate request to the frontend server using REST API. On receiving this request, the front end server removes that particular entry from its cache.

4. **DB Consistency**

   Since there are multiple replicas of the catalog server and each server maintains its own copy of the database, consistency between all databases is required.

   - When the catalog server receives a request to update the database, it makes a sync REST API call to the other replica with the necessary parameters
   - On receiving a sync API call, the server will execute the same query on its database
   - This leads to all databases being **eventually consistent**

PART 2

All the microservices have been dockerized. This has been done using docker and docker compose.

Docker compose file (yml) has the following information to start the container

- Dockerfile location for each service to build and run the system
- Ports which must be exposed in the microservices
- The order in which the microservices must start
- The name of the container for each microservice.
- The volume which must be mounted to stream logs from container to host machine

Dockerfile has the following information to build the image

- Environment (JDK, Maven)
- Source code
- Environment variables
- Build and execute instructions

Logs

Once the containers are up and running, all logs will be streamed from the containers to the host machine, so that once the system is shut down, all logs are preserved.

PART 3

1. **Failure Detection**

In order to detect the backend server failure we have used heartbeat protocol. Catalog and Order servers send the heartbeat message to the frontend server every 2 sec to declare that it is alive and can accept requests. We have used the ninja scheduler in the backend to send heartbeat messages in a separate thread. Frontend server uses the health check API to detect if there is any server not responding. The timeout for the heartbeat message is set as 3 sec after which frontend marks the corresponding server as FAILED.

2. **Fault Tolerance**

If the server is marked as FAILED, all the incoming requests will be switched to the non-faulty replica. A client request is sent to the backend server only if it is in RUNNING status. The messages(intransit) that the frontend issued to the backend server before crash detection is again sent to the non faulty replica for processing. Thus the system does not halt on failure of any backend server in presence of non- faulty replicas.

3. **Recovery**

When the crashed process comes up in the system, it starts to send heartbeat messages to the frontend server. Frontend marks the server as “RUNNING” again and starts sending client requests. Before executing the first client request after failure, the catalog server sends a replica request to replica to synchronize the DB state. This ensures replication consistency and correctness of the system.

[3] Assumptions

- The data of books is already present in the database table. This data is fetched by the catalog server into in-memory data structure on server startup.
- The number of items of each book is fixed. When a buy request is made and the number of items for the specified book is found 0, stock is replenished again. So just the current buy request would fail.

[4] Design Considerations

- Each microservice is implemented as a separate module which could be easily migrated to a distributed platform.
- There are **two replicas of catalog** microservice and **two replicas of order** microservice. First replica of order microservice is connected to the first replica of catalog microservice and the second replica of order microservice is connected to the second replica of catalog microservice
- The information about the books is maintained on all catalog servers.
- All microservices accept **concurrent requests** and handle them asynchronously
- **SQLite** is used to persist the information about books and make the system fault tolerant. Each server maintains the log of requests and its output.
- The update operation is synchronized to maintain the data consistency.
- We have followed the standard MVC design pattern and object oriented principles while implementing the project

[5] Improvements

- Port information for each microservice is hardcoded.