CS 677 S21 Lab 2 - Bookstore

Design Document

Submitted By

Jagriti Singhal, Bhumika Kalavadia

Date: 5 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**. The script will first configure and start all the microservices (frontend, catalog and order) on the specified port based on command line arguments. The port for each microservice is fixed: Frontend - 8080, Catalog - 8081 and Order - 8082. The hostname could be specified in the hostname.conf file as per the test case. The code will read the hostname.conf and pick the required hostname while making HTTP requests.

To run all microservices and client on same server(here 3 is the num of clients) - **sh run.sh all 3**

To run only catalog microservice - **sh run.sh catalog**

To run only order microservice - **sh run.sh order**

To run only frontend microservice - **sh run.sh frontend**

To run clients (here 3 is the num of clients) - **sh run.sh client 3**

The above microservices are implemented as standalone maven modules. 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.

[2] Design and Implementation

Ninja is based on MVC design pattern. The framework creates a few packages based on conventions. The Java classes are categorized under conf, controllers, models, and services directories in src/main/java. The two routing components viz. Router and Controller specify the routes for microservice endpoints and logic to process the request respectively. The route defines a mapping between URL request and the associated controller which processes the request and returns the appropriate response. Models and services package is used to implement the underlined business logic of the microservice.

a) **Frontend**

The front end server supports following three operations which are implemented in ApplicationController of the microservice:

- search(topic) - get request which allows the user to specify a topic and returns all entries belonging to that category (a title and an item number are displayed for each match).

http://localhost:8081/search/<topic> [GET]

- lookup(item_number) - get request which allows an item number to be specified and returns details such as number of items in stock and cost
  http://localhost:8081/lookup/<bookNumber> [GET]

- buy(item_number) - post request which specifies an item number for purchase.
  http://localhost:8081/buy/<bookNumber> [POST]

The first two operations trigger queries on the catalog server. The buy operations triggers a request to the order server. The logic for the same is implemented under service directory.

b) Catalog

The catalog server supports following three operations which are implemented in ApplicationController of the microservice:

- queryByItem(item_number) - all relevant details of the specified items are returned. This operation is invoked from both the frontend server and the order server.
- queryBySubject(topic) - server returns all the matching book entries of specified topic.
- update(item_number) - decrements the number of items in stock after successful buy
- updateCost(item_number,cost) - updates the cost of specified book

The server maintains the HashMap of books indexed on item number and topic for serving the queries effectively. The data is stored in the database table(books.db) and populated to the hashmap when the first request is made to the server. All the subsequent requests are handled via in-memory data structure. Whenever a buy request is successful the database entry is updated to decrement the count of books available. Also there is a provision to update the cost and number of items of any book.

c) Order

The order server supports a single operation: buy(item_number). Upon receiving a buy request, the order server first makes a get request(queryByItem) on the catalog server to check the availability of the item. After confirming the availability, it makes a post request(update) to the catalog server to decrement the number of items in stock by one. The buy request can fail if the item is out of stock and restock request is sent to the catalog server to replenish the item.

[3] Assumptions

- 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.
- The data of books is already present in the database table. This data is fetched by the catalog server into in-memory data structure.

[4] Design Considerations

- Each microservice is implemented as a separate module which could be easily migrated to a distributed platform.
- The information about the books is maintained by only the catalog server and not replicated on other servers. Frontend and Order server fire query to catalog server to get the information.
- The server accepts multiple concurrent requests and handles 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.