CS 677 S21 Lab 2 - Book Store

Design Document V2

Submitted By
Yelugam Pranay Kumar
Date: 5 April, 2021

The project has been implemented in Python using Flask and can be run using a single bash script.

[1] Project Setup:

- To execute this project locally, execute **run.sh**

- To execute this project on AWS, execute **sh aws_run.sh $(cat config.txt).** Before executing this script, make sure that the **config.txt** file has the path to the **KP**(key pair) file and the **DNS** for the category, order and the front-end are set up.

  An example config.txt would look like this:

  ```
  ./677kp.pem
  <ec2-3-83-245-8.compute-1.amazonaws.com>
  ec2-52-207-153-47.compute-1.amazonaws.com
  Ec2-54-208-155-245.compute-1.amazonaws.com
  ```

- The client will run on the users local system and the output will be stored in the local system. The test cases will also run on the user's local machine and the output will be stored in the local system.

The script will run all the processes and carries out the API calls in the client process.
Each test case will run and the results from the client and the tests files will be stored in the **ClientOutput.txt** and **tests.txt** in their corresponding folders.

The project structure is as follows:
Book-Store
- Frontend-server
    - frontend.py
    - requirements.txt
- Catalog-server
    - catalog.py
    - Requests.txt (Storage for the update requests)
    - requirements.txt
- Order-server
    - order.py
    - requirements.txt

- Client process
    - Client.py
    - Client-output.txt
- run.sh (Runs the servers on the local machine)
- Tests
    - Tests.py (test cases checking for search, lookup and buy requests)
    - tests.txt (Contains the outputs from the tests.py)
- Config.txt (contains the path to the key-pair and the dns of the servers)
- aws_run.sh (includes commands to run the servers on different AWS EC2 instances)

The servers: Catalog, Order and frontend operate on ports 8080, 8082 and 8081 respectively.

[2] Assumptions

The project is designed based on the following assumptions:

- Items are never restocked and the servers will respond with no result once the stock reaches 0.
- To achieve fault tolerance in case of a failure the update requests on the catalog server are stored in a log file (request.txt)
- Sqlite database is used to store the data in the catalog server.
- The database is assumed to handle the concurrent read/write operations.

[3] Design and Workflow

Each process/server is written in a separate folder and have the following properties:

Client-server:
Client contains the API request calls for the front-end server. It logs the requests and the corresponding responses in *ClientOutput.txt*.

Front-end server:
App implements the following routes:

- Search(topic): The requests coming on this route calls the Catalog server to query for the topic, collects the response from the Catalog server, and sends the response if the topic is present in the database. If the topic is not present in the database, it returns an empty list with message *Item not present for the topic*.

- Lookup(itemNumber): The requests coming on this route calls the Catalog server to query with the itemNumber, collects the response from the Catalog server, and sends the response if the itemNumber is present in the database. If the itemNumber is not present in the database, it returns empty json with message *Item not present for the item number*.
- Buy(itemNumber): The request coming on this route calls the Order server to query with the itemNumber, collects the response from the Order server, and sends the response if there is enough stock for the item. If there's no stock, the route responds with a message *The item is out of stock*.

Catalog-server:

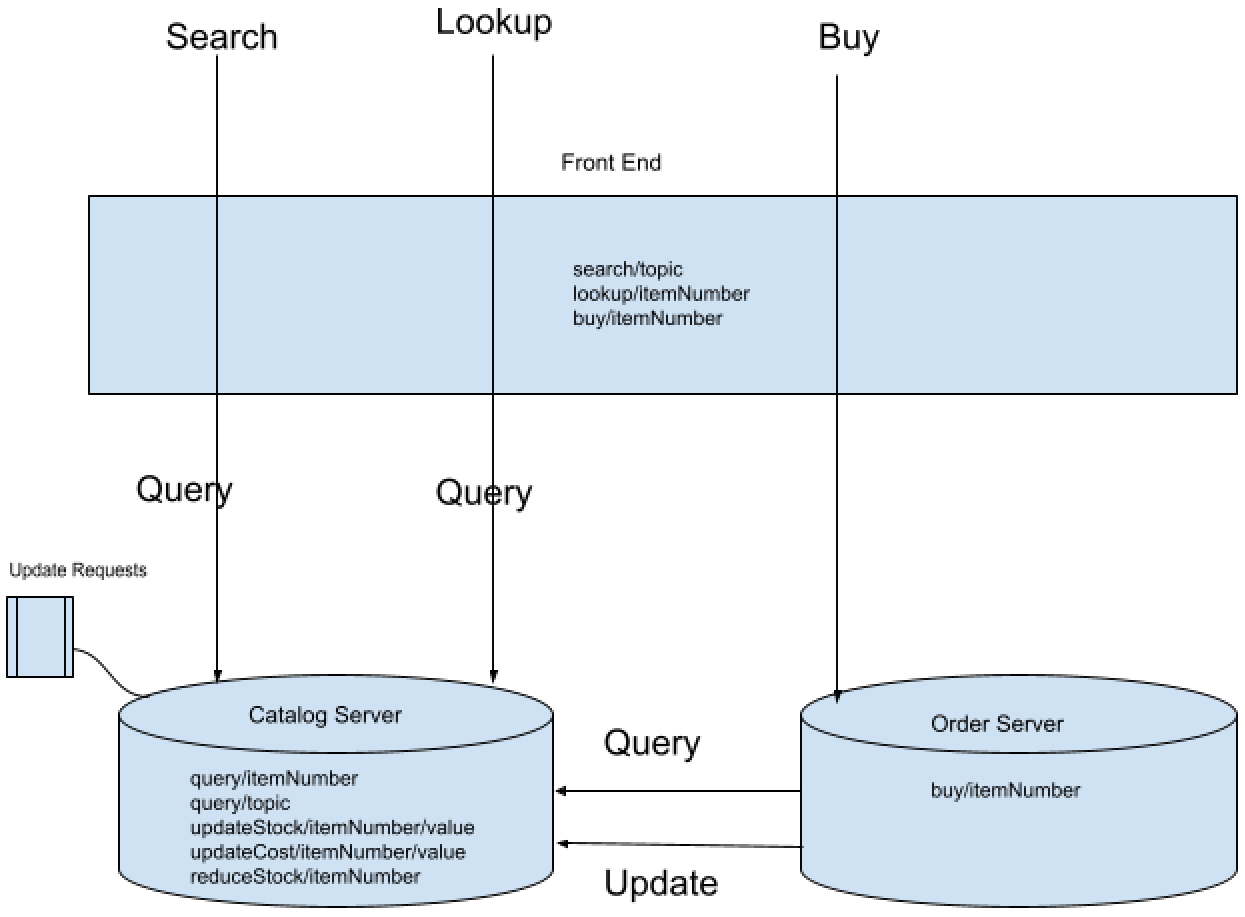
Catalog implements the following routes:

- */query/<int:itemNumber>:* Queries the database to get the item with the corresponding item number, serializes it and returns the serialized result to the response if the number of result items is greater than 0.
- */query/<string:topic>*: Queries the database to get the item with the input topic, serializes it and returns the serialized result to the response if the number of result items is greater than 0.
- */updateStock/<int:itemNumber>/<int:value>:* Given the item number and the stock value, this route finds the item with the itemNumber and updates its stock value with the value from the input. It returns the updated item from the database.
- */reduceStock/<int:itemNumber>:* Given the item number of the catalog item, this route's functionality is to reduce the stock of the item by 1 and then return the updated item from the database.
- */updateCost/<int:itemNumber>/<int:value>:* Given the item number and the cost value, this route finds the item with the itemNumber and updates its cost value with the value from the input. It returns the updated item from the database to the response.

Order server:

Order server implements the following routes:

- /buy/<int:itemNumber>: Given the item number as input, this route queries the Catalog server for the item and gets the response from the Catalog server of the matching itemNumber. If the stock of the matching number is greater than or equal to 1, it sends an updateStock request to the Catalog server and gets the response from the catalog server. The Order server then returns the response from the catalog updateStock request to its response.



Fault Tolerance & Logging

- Every update request is logged in the requests.txt. In case of a failure, the requests.txt can be used to update the database and get the corresponding response for the new request.
- The initial data is present in a file called data.json to retrieve the data into the database using this snapshot of the data.

[4] Design Considerations

- I have used Flask to implement the REST apis.
- The three servers start up at 3 different ports. 8080,8081,8082.
- The routes in the servers are type casted.
- The logging of the requests are done in a text file and is stored on the catalog server.
- Included requirements.txt in every server to install the dependencies on all the servers when it is run on AWS

[5] Improvements and Extensions

- The logging of the requests could be improved to perform WAL and maintain fault tolerance.
- Usage of yaml for storing the configuration.
- Could use dockerization of the servers to help ease the deployment process.