677 Design Documentation

Nishant Raj, Noel Varghese, Rishika Bharti

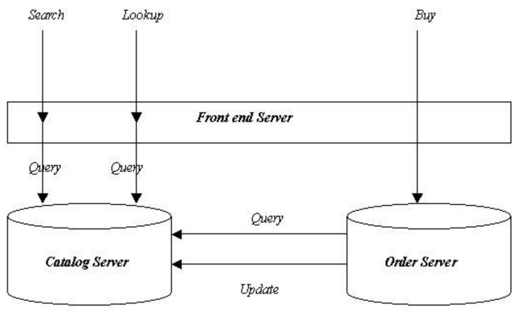
14 March 21

1 System Design

This project aims to design a two-tier web application for an online book store called Pygmy.com. The bookstore sells the following books.

Item Number	Book	Topic
1	How to get a good grade in 677 in 20 minutes a day.	Distributed Systems
2	RPCs for Dummies.	Distributed Systems
3	Xen and the Art of Surviving Graduate School.	Graduate School
4	Cooking for the Impatient Graduate Student.	Graduate School

The two tiers of the application are the front-end and back-end and each tier consists of various components as depicted below in the image.



The entire system that we propose in this design documentation handles GET and POST requests and has been deployed and tested on local as well as AWS using FLASK in python to run the microservices. We have also performed the dockerization of the entire framework. All the components i.e. Frontend, Order, Catalog have been dockerized separately.

1.1 Frontend

A client makes a request to the front-end application. The front-end application is designed to support user-requests in the form of three operation which are listed below:

- ***search(topic)*** Using the search operation, the user of the web-application can specify a topic and the operation will query the servers in the back-end to retrieve all the books that come under that particular topic. The returned output contains the title and the item number associated with the title.

  We have implemented this using the *search()* function which uses a *GET* call and the route that it uses is given by */search?topic="Topic"*, where *"Topic"* represents topic of the book mentioned in the database.

- ***lookup(item_number)*** This operation allows the user to pass a item number that belongs to a particular book and the front-end then returns the details associated with the book having the item number. The details include the number of books of that type available in stock and the unit cost of the book.

  We have implemented this using the *lookup()* function which uses a *GET* call and the route that it uses is given by */lookup?id="book id"¿*, where *"book_id"* represents item_number of the book mentioned in the database.

- ***buy(item_number)*** The buy operation allows the user to specify an item number that they wish to purchase. All the aforementioned operations that hit the front-end server query the servers that are set up on the back-end.

  We have implemented this using the *buy()* function which makes a *POST* call.The route would be and data in JSON format for the post call would be {*"id":item_number*}. Internally it makes a call to the *buy_product()* function which then hits the order service endpoint and has a route */purchase.*

1.2 Back-end

The back-end consists of two components which are the catalog server and order server.

1.3 Catalog Server

The operations from the front-end query different servers fall on the back-end. The catalog server maintains the catalog for the various books that are sold in the bookstore. For every entry, the catalog server stores the number of items in stock, the cost and the topic of the book in the database.We can query items from catalog based by two ways which are through the topic of the books and by the item number. A query can be made using a *topic* or using an *id*. There is an *update()* functionality as well that makes a POST call and helps in making an update to the "stock" and "cost" the items present in our database. The routes for the two types of queries are given below -

- Query using *topic* - The route followed is *query_product_details/query?topic="topic"*
- Query using *id* - The route followed is *query_product_details/query?id="book_id"*

1.4 Order Server

The operations from the front-end query different servers fall on the back-end. The back-end server is responsible for the incoming buy requests. The route for the order server is *purchase_product/query?id="book_id"* where *book_id* is the id associated with the book that is purchased. Order server leverages a POST call to implement a *purchase()* functionality. If the *id* is fetched and the purchase is made, then the stock will get reduced by an amount equal to -1.

2 Design Tradeoffs and Proposed Improvements

2.1 Design Tradeoffs

- Instead of using ".csv" or ".txt" files, we selected to use SQLite as Flask supports it.
- We had two options to choose as a lightweight micro web framework: Flask for Python or Spark/Ninja for Java. We decided to proceed with Flask in Python as it is easier to work and manage virtual environments in Python.Also, Flask is a very lightweight framework that let's us define REST APIs with minimal effort.
- In order to ensure that dependency management is easy, we decided to leverage docker containers. The ease of deployment using *docker-compose* command makes it a very suitable option for our usecase.

2.2 Proposed Improvements

- Using a better database instead of sqlite should reduce latency and enable us to setup replication and split reads and writes in a master-slave architecture.

– Currently we are running only one container per tier. Instead of this we can replicate them and load balance the requests at each tier, further removing bottlenecks and improving the latency.

– Deployment can be made easier if we used a container orchestration tool like kubernetes.

3 How to run the system on AWS?

– cd to deploy-to-aws directory inside project directory

```
cd deploy-to-aws
```

– To deploy the code to ec2 instances we need to change the docker-compose files and post that run the command shown

So we need 4 ec2 machines each hosting frontend, order, catalog and client in this order. Before executing the below script edit docker-compose1.yml and docker-compose2.yml. Change the CATALOG_SERVICE_ENDPOINT and ORDER_SERVICE_ENDPOINT to the corresponding pvt ips of catalog-service-ec2-public-dns and order-service-ec2-public-dns respectively. This is how the services will know how to reach themselves internally within AWS. i.e

CATALOG_SERVICE_ENDPOINT=http://catalog-ec2-instance-pvt-ip:8080
ORDER_SERVICE_ENDPOINT=http://order-ec2-instance-pvt-ip:8080

```
./deploy_code.sh path-to-private-key
↪  frontend-service-ec2-public-dns
↪  <order-service-ec2-public-dns>
↪  <catalog-service-ec2-public-dns>
↪  <client-testing-ec2-public-dns>
```

– Once the ips are changed, the deploy_code.sh script can be run as shown in the example. The first parameter to the script is the path to the private key which you use to ssh to the ec2 instances.

– Check if services are running SSH to the first 3 ec2 servers and check if the docker-containers are running properly, using the below command.

```
docker ps
```

– Test the endpoints from the 4th ec2 instance post doing ssh to it

```
cd BookStore_Pygmy_Microservices
python3 run_api_tests.py -h <frontend-ec2-instance-pvt-ip> -p 8080
```

– 5. Further testing can be done by manually doing curl calls as highlighted in the readme Ensure to replace localhost with ip and port 8000 with 8080