DESIGN DOCUMENT

COMPSCI-677

LAB 2 - PYGMY.COM

Overview

This document gives a detailed description of the design of Pygmy.com - the world's smallest bookstore. The system consists of distributed microservices, capable of running on different machines. It is also capable of handling HTTP GET requests and gives a response in JSON. The system consists of the following three components:

1. Front End Server
2. Catalog Server
3. Order Server

The entire system is built using Python, with the Flask library to run the microservices. Additionally, we are using Flask-SQLAlchemy for our catalog and order database, which uses a variant of sqlite to store the data. The following sections go into more detail about each of the components.

Front End Server

This server is responsible for handling the requests from the clients. The front end server supports the following operations:

- **Search** - This is implemented in the `search()` function. The route for this is `/search/<topic>`, where `topic` is the topic of the book. A query to this hits the catalog server, which queries the catalog database for all the books with the passed topic. The query finally returns a JSON with all the details of the books matching the topic.
- **Lookup** - This is implemented in the `lookup()` function. The route for this is `/lookup/<item_number>` where `item_number` is the item id for the book. A query to this hits the catalog server, which queries the catalog database for the book with the passed item id. The query finally returns a JSON with all the details of the book matching the item id.
- **Buy** - This is implemented in the `buy()` function. The route for this is `/buy/<item_number>` where `item_number` is the item id for the book. A query to this hits the order server, which queries the catalog server for the book with the passed item id. If the book has enough stock available, the buy is successful, otherwise it fails. The query finally returns a JSON with all the details of the book matching the item id.

Catalog Server

This server is responsible for handling the requests from the front end and the order servers. It supports the following operations:

- **Query_by_subject** - This is implemented in the `query_by_subject()` function. The route for this is `/query_by_subject/<topic>` where `topic` is the topic of the book. This receives requests from the `search()` function of the front end server, queries the catalog database and returns a JSON with the details of the books matching the topic.
- **Query_by_item** - This is implemented in the `query_by_item()` function. The route for this is `/query_by_item/<item_number>` where `item_numbe`r is the item id of the book. This receives requests from the `lookup()` function of the front end server and the `buy()` function of the order server, queries the catalog database and returns a JSON with the details of the book matching the item id.
- **Update** - This is implemented in the `update()` function. The route for this is `/update/<item_number>` where `item_number` is the item id of the book. This receives requests from the `buy()` function of the order server, queries the catalog database, and decrements the stock by one if the stock is greater than 0 and returns a JSON with a success message. If the stock is 0, it will return a JSON with a failed message.

Order Server

This server is responsible for handling the buy requests from the front end server. This is done through the `buy()` function. The route for this is `/buy/<item_number>` where `item_number` is the item id of the book. This function sends a `query_by_item` request to the catalog server to check if the book is in stock. If it is in stock, it sends an `update` request to the catalog server to decrement the count of the book by 1. It also creates a new order in the orders database if the order is successful and returns the order details in the form of a JSON.

How It Works

The clients can send the search, lookup and buy requests. All of these are handled by the front end server. The following presents a general flow of all the three cases:

1. Search -
    a. The client sends a search request to the front end server, specifying a topic.
    b. The front end server receives the requests in its `/search` route and routes the request to the catalog server.

    c. The catalog server receives the request in its `/query_by_subject` route and queries the catalog database for all the books with the matching topic. The details are returned in the form of a JSON.
    d. The front end server receives the JSON file from the catalog server, and returns it to the client.
2. Lookup -
    a. The client sends a lookup request to the front end server, specifying an item number.
    b. The front end server receives the requests in its `/lookup` route and routes the request to the catalog server.
    c. The catalog server receives the request in its `/query_by_item` route and queries the catalog database for the book with the matching item number. The details are returned in the form of a JSON.
    d. The front end server receives the JSON file from the catalog server, and returns it to the client.
3. Buy -
    a. The client sends a buy request to the front end server, specifying an item number.
    b. The front end server receives the requests in its `/buy` route and routes the request to the order server.
    c. The order server receives the request in its `/buy` route and queries the catalog server to check if the book is in stock.
    d. The catalog server receives the request in its `/query_by_item route` and queries the catalog database for the book with the matching item number. The details are returned in the form of a JSON.
    e. The order server checks the result;
        i. if the book is in stock, it sends back a JSON with a “failed” response.
        ii. Otherwise, it sends an update request to the catalog server to decrement the count of the book by one. After this, it creates a new entry in the orders database and returns the order details in the form of a JSON.
    f. The front end server receives the JSON file from the order server, and returns it to the client.

All 3 servers constantly log the request ID for each request as well as the time taken to service the request, and save them in a file on disk

How To Run The System

1. Create a public-private key pair.
    a. On the client (local) machine, run the following command:
    `ssh-keygen -t rsa`
    b. After this, you will be given the following prompt:
    `Enter file in which to save the key (/home/demo/.ssh/id_rsa):`
    Press Enter here.
    c. After this you will be given the following prompt:
    `Enter passphrase (empty for no passphrase):`
    Press Enter here.
    d. Copy the public key to the remote machine. For this you will need the `ssh-copy-id` command. It is installed by default in most linux variants. It won't be installed on a Mac. Use the following command to install it if you are on a Mac:
    `brew install ssh-copy-id`
    e. Copy the public key using the following command:
    `ssh-copy-id <username>@elnux.cs.umass.edu`
    Where <username> is your edlab username. Alternatively, you can also use the following command to paste the key:
    `cat ~/.ssh/id_rsa.pub | ssh<username>@elnux.cs.umass.edu "mkdir -p ~/.ssh && chmod 700 ~/.ssh && cat >> ~/.ssh/authorized_keys"`
    f. Enter your password, if prompted.
    g. Now you should be able to login without using a password. Please test that you are able to do so before running the application.
2. Clone the git repo on your edlab machine as well as your local machine.
3. Inside the src/ directory **in EdLab machine**, run the following command:
    `source venv/bin/activate`
    `pip3 install -r requirements.txt --user`
4. In the `run_pygmy.sh` script **in the local machine**, make the following changes:
    a. In the `user` variable, assign your edlab username.
    b. In the `remote_path` give the remote directory where you cloned the repo. This can be done using the `pwd` command. Make sure to give a '/' at the end of the path.
    Example: `/nfs/elsrv4/users1/grad/aayushgupta/cs677/`
5. Run the script run_pygmy.sh **on your local machine** using the following commands:

```
chmod +x run_pygmy.sh
./run_pygmy.sh
```

This will spawn:

- The front end server on `http://elnux3.cs.umass.edu:34600`
- The order server on `http://elnux2.cs.umass.edu:34601`
- The catalog server on `http://elnux1.cs.umass.edu:34602`
- The client on your local machine.
- Once the tests are complete, you will be able to view the output and performance metric files for all the clients on your local machine, and the server logs will be stored under 'front_end_server.txt', 'order_server.txt' and 'catalog_server.txt' on your edLab machine in the src sub directory

Design Tradeoffs and Future Improvements

For the design tradeoffs, we considered the following:

1. We ended up choosing Python-Flask microservice library instead of Java-Spark, because we believed that Python will provide an easier dependency management as well as isolation using virtual environments.
2. We used Sqlite for our database instead of a text or csv file, because Flask has an inbuilt support for sqlite using flask-sqlalchemy. Also, this is more robust, as it uses transactions.

For future improvements, we considered the following:

1. We can have user specific sessions where the user can log in using their credentials. This will help isolate the different orders based on the users.
2. The catalog server is a bottleneck in our design, as it has to handle all the requests from both the order and the front end servers. We can distribute the catalog server by two methods:
    a. Load balancing using domain name: We can have another domain name which routes to the same catalog server. In this way, the number of requests won't be sent to a single address.
    b. Replication of server: We can replicate the catalog server. However, in this case, the database is still a bottleneck.
    c. Replication of database: We can replicate the database, but this will require consistency control, which will complicate the design.