# DESIGN DOCUMENT

COMPSCI-677

LAB 3 - 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.

The system also implements the following:

- The system uses in-memory caching implemented on the front-end server, caches responses and responds if the same request is received. The cache is implemented using the Flask-Caching library. After a write operation for a particular data item, the cache is invalidated.
- The system implements replication - by replicating the catalog and the order servers. Each server has two replicas. To guarantee consistency, we are using primary based consistency protocol. In this protocol, each replica is responsible for performing writes for a particular data item, and forwarding that write to the other replica.
- The system also implements load balancing, making use of the replicated servers. We are using the "round-robin" protocol.
- The system implements fault tolerance. We are using a heartbeat mechanism to implement this. The front end server has a dedicated heartbeat thread for each of the order and catalog replica, which periodically ping them to check their status. As soon as a server is down, a separate thread is responsible for respawning the servers. If it is a catalog server, it is also responsible for making sure that the server resyncs with the other replica so that its database is up to date.
- Finally, we have dockerized the entire application. Each of the components are capable of running in a separate docker container, without any issues.

## 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 replica decided using the load balancing algorithm, 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 replica decided using the load balancing algorithm, 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 replica decided using the load balancing algorithm, which queries the catalog server primary 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. If the order server is down, it tries again after a certain time.
- **Caching -** This is implemented using the Flask-Caching library. The `@cache.memoize()` decorators above the `search()` and `lookup()` functions are responsible for this. There is also a provision to invalidate the cache in the `buy()` function.
- **Fault Tolerance -** This is implemented in the `heartbeat()` and the `respawn_servers()` functions. Dedicated threads use the `heartbeat()` function to ping their respective servers and upon failure detection set a shared flag. Another thread periodically reads this shared flag; if it is set, it will use the `respawn_servers()` function to respawn the failed server.
- **Load Balancing -** We are using the “round robin” protocol to balance the load.

## 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_number` 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.
- **Resync Database -** This is implemented using `resync_catalog_db()` and `request_restore_catalog_db()` functions. This is initiated by the front end server when the server is respawned.

## 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 primary catalog server for the book to check if the book is in stock. If it is in stock, it sends an `update` request to the same 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. If the server is down, it sends a failure response to the front end server, which tries again.

## 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 checks which catalog server has its turn to process the request. If that server is up and running, it routes the request to the catalog server, otherwise the request is forwarded to the replica which is running.
    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 checks which catalog server has its turn to process the request. If that server is up and running, it routes the request to the catalog server, otherwise the request is forwarded to the replica which is running.
    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 checks which order server has its turn to process the request. If that server is up and running, it routes the request to the order server, otherwise the request is forwarded to the replica which is running.
    c. The order server receives the request in its `/buy` route and queries the primary catalog server to check if the book is in stock. If the catalog server is down, it sends back a failure response.
    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 not 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. The catalog server is responsible for updating the other replica. 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. If the response results in a failure, the front end server tries again.

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 between your local machine and Edlab.
    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. Create a public-private key pair between Edlab and Edlab:
    a. Log in to Edlab as follows:
    ```
    ssh elnux1.cs.umass.edu
    ```
    b. Follow the steps from 1.a. (previous step) with the remote directory as `elnux2.cs.umass.edu`

    (This is done to enable respawning of servers)
3. Clone the git repo on your edlab machine as well as your local machine.
4. Inside the src/ directory **in EdLab machine**, run the following command:
    ```
    source venv/bin/activate
    pip3 install -r requirements.txt --user
    ```
5. 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/`
6. 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 servers on `http://elnux2.cs.umass.edu:34601` and `http://elnux2.cs.umass.edu:34611`
- The catalog servers on `http://elnux1.cs.umass.edu:34602` and `http://elnux1.cs.umass.edu:34612`
- 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

## Docker execution:

1. Clone the repository on your local machine
2. Cd into the Docker directory - this contains the run_pygmy_docker.sh script
3. The docker folder also consists of a tests directory, with each directory containing a different test (testing a different aspect of the system - caching, consistency and fault tolerance), and a run_client script that calls all these tests.
4. In order to execute the dockerized version of the application simply execute the run_pygmy_docker.sh script. This will automatically pull all the images for all the containers from docker-hub onto the local machine , spin them up and execute all the tests
5. Once all the tests have been executed, corresponding output files as well as server output files and log files will be generated within each sub-directory for that respective test

## 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.
3. We are using Flask-Caching for in-memory caching. This doesn't allow us to specify a specific cache eviction algorithm, like LRU or LFU.
4. We are using a primary based consistency protocol for writes to ensure strong consistency. However, this doesn't guarantee availability, as the requests for a particular data item whose primary is down will have to wait for the replica to be up and running. This is just for write requests; read requests will still be serviced by the remaining functional replica.

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. Instead of the primary based protocol, we can use something like a distributed locking mechanism to ensure availability.