

DESIGN SPECIFICATION

This project implements a simple book store through a client server model that is capable of handling a higher workload and uses REST API's to communicate between layers.

How it works

There are two tiers in the web-app: a front-end and a back-end.

The front end accepts incoming requests from clients and accordingly processes them, by forwarding the calls to relevant servers. The backend servers are replicated and the frontend server implements a load balancer to balance the load between the replicas of the respective backend servers.

The backend consists of two servers, each of which has two replicas:

1. Catalog server A and B: Each replica of the catalog server maintains a catalog (which currently consists of four books) and the relevant information corresponding to those books. Following is the schema for the catalog server.

```
{
    id: int (primary key),
    title: text
    cost: text
    count: text
    topic: text
}
```

2. Order Server A and B: Each replica of the order server maintains a list of all orders received for the books.

Following is the schema for order server:

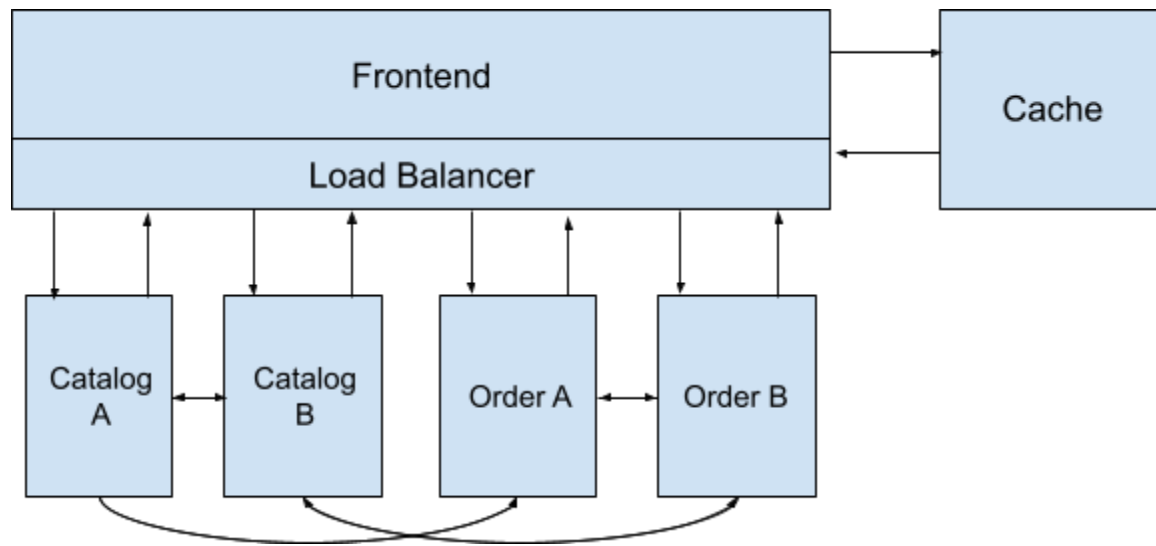
```
{
    id: int (primary key),
    item_id: int,
    Created: date
}
```

The front end server also implements a caching mechanism to cache the results of

The front end server supports three operations:

1. search(topic) - 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).
2. lookup(item_number) - which allows an item number to be specified and returns details such as number of items in stock and cost
3. buy(item_number) - which specifies an item number for purchase.

The way the servers interact with each other is shown below:



IMPLEMENTATION

Catalog server

Each replica of the catalog server exposes a GET API ('item') which allows for books to be looked up from its database, either by id, or by topic. The catalog server also exposes a PUT API on 'item' which allows for the altering the items database in the catalog server. For example, This can be used when a book is purchased or if the cost of a book needs to be updated. Each catalog server also has another PUT API that is called by other replicas of the catalog server for propagation of updates to the catalog database. This is how the catalog servers maintain consistency in the database. Whenever a copy of a catalog server makes an update to its own database, it also calls this PUT API on the other copies of the server to propagate this update.

Order Server

Each replica of the order server exposes a GET API('buy') method which looks for a book corresponding to a certain ID and then buys it from the catalog server. When a client invokes the buy function on the front end server to buy a book by an ID number, the frontend server calls the endpoint on the order server with this ID number to buy a book. The order server first queries the respective replica of the catalog server to ensure that the book to be bought is available in stock, and then executes the buy operation and updates the database to reflect it. Each order server also has another PUT API that is called by other replicas of the order server for propagation of updates to the order database. This is how the order servers maintain consistency in the database. Whenever a copy of an order server makes an update to its own database, it also calls this PUT API on the other copies of the server to propagate this update.

Frontend Server

The front end server also allows the user to look for a book either by its ID or by its topic. The lookup function allows the user to look up the details of a particular book by its ID. A query is made to the catalog server and specific details of the book corresponding to the ID is returned. Similarly, to search for all books belonging to a particular topic, the search function is implemented by the front end server which makes a call to the catalog server which returns a list of all the books belonging to that topic. The frontend server also allows the client to buy a specific item from the catalog via the GET API buy.

Load Balancing

Load balancing is implemented at the frontend server. We have used a simple round robin technique for load balancing. Since there are two copies each of order server and catalog server, this means that each request alternately goes to one of the two order/catalog servers.

Caching

We have implemented caching so that lookup requests can be served more quickly. The frontend has three methods that a client can access, search, lookup and buy. For search and lookup a client gets returned either a list of books corresponding to the topic searched for, or return the details of the book corresponding to the id that is looked up. First, the frontend server will check its own cache, and if the results of the query are present simply return that instead of sending the request to the backend server. However, everytime the buy method is called, the entire cache is invalidated.

Replication

The frontend server has not been replicated.

For our project we have assumed there to be two replicas each of the order server and the catalog server. Each server is replicated along with its code and database files. When an update is made to the database on any server, it first updates the local copy of the database, and then propagates this update to all the replicated copies of the database by calling a method with PUT API on the other copies of the server. This is done to ensure that there is a consistency between the different copies of the database among the replicated backend servers.

Every copy of the order server, thus must have knowledge of the ports and IP addresses of the other copies of order servers so that it may propagate any update to its database to the other copies. It must also have knowledge of the corresponding catalog server that it must connect to. For the purpose of our project we have assumed that order A connects with catalog A and similarly order B connects with catalog B. Similarly, catalog server needs to have network information about its replicas as well.

Dockerization

In the current implementation, each server will be deployed as a docker container which will be communicating with each other for serving the client request. Each folder in the *pygmy* repository contains a Dockerfile which is used to deploy the images containing source code and required environment variables to the docker hub. Currently the docker images has been pushed to the public repository *himgupta1996/pygmy*.

1. **Frontend server docker image location with tag:** himgupta1996/pygmy:frontend
2. **Order server docker image location with tag:** himgupta1996/pygmy:order
3. **Catalog server docker image location with tag:** himgupta1996/pygmy:catalog

To pull the images from the repository and run the docker containers on the machines specified in the *env.cfg*, run the file "*runme.sh*" present in the main folder using the command: `./runme.sh`. This will deploy the docker server containers to respective machines and trigger the *client.py* to instantiate the traffic and test the servers. Logs of all the servers can be found in the directory *logs/* inside the main folder *pygmy*. Client log can be found in the main folder itself.

API Specification

Conventions

- **Frontend**- Front End server
- **Order** – Order Server
- **Catalog**- Catalog Server
- **Status** - HTTP status code of response.
- All the possible responses are listed under 'Responses' for each method. Only one of them is issued per request server.
- All response are in JSON format.
- All request parameters are mandatory unless explicitly marked as optional.

Status Codes

All status codes are standard HTTP status codes. The below ones are used in this API.

- 2XX** - Success of some kind
- 4XX** - Error occurred in client's part
- 5XX** - Error occurred in server's part

Status Code	Description
-------------	-------------

Used	
200	OK
201	Created
400	Bad request
401	Unauthorized
422	Unprocessable Entity
404	Resource not found
500	Internal Server Error

Methods for the Front End Server

1. Buy

Call the buy method on the order server to buy an item with given ID

Request

Method	URL
GET	/buy

Query Parameters	Optional	Value Type	Description
id	False	String	The id of the book that needs to be

			bought
--	--	--	--------

Example: localhost:8010/buy?id=1

Response

Status	Response
200	<pre>{ "message": "Item with id 1 bought successfully.", "order": { "id": 196 }, "status": "Success", "validation_code": 200 }</pre>
404	<pre>{ "message": "The item with id 1 is no longer present in the catalog server", "status": "Failed", "validation_code": 404 }</pre>

2. Search

Search for all the books belonging to a certain topic

Request

Method	URL
GET	/search

Query Parameters	Optional	Value Type	Description
------------------	----------	------------	-------------

topic	False	String	Topic of books to search for
-------	-------	--------	------------------------------

Example localhost:8010/search?topic=distributed systems

Responses

Status	Response
200	<pre>{ "item": [{ "cost": 10, "count": 20, "id": 1, "title": "How to get a good grade in 677 in 20 minutes a day.", "topic": "distributed systems" }, { "cost": 10, "count": 15, "id": 2, "title": "RPCs for Dummies.", "topic": "distributed systems" }], "message": {}, "status": "Success", "validation_code": 200 }</pre>

3. Lookup

To lookup a particular item in the server

Request

Method	URL
--------	-----

GET	/lookup
------------	---------

Query Parameters	Optional	Value Type	Description
id	False	int	Id of the book to search for

Example localhost:8010/lookup?id=3

Response

Status	Response
200	<pre>{ "item": [{ "cost": 10, "count": 0, "id": 3, "title": "Xen and the Art of Surviving Graduate School.", "topic": "graduate school" }], "message": {}, "status": "Success", "validation_code": 200 }</pre>

Methods for the Catalog Server

1. Item

Look for Items in the database either by topic, ID, or return all books if no parameters are passed

Request

Method	URL
GET	/item/<id_>
GET	/item

Query Parameters	Optional	Value Type	Description
id	True	String	The id of the book to look up
topic	True	String	Return all books with given topic

Example: localhost:8011/item?topic=distributed systems

Response

Status	Response
200	<pre>{ "item": [{ "cost": 10, "count": 20, "id": 1, "title": "How to get a good grade in 677 in 20 minutes a day.", "topic": "distributed systems" }, { "cost": 10,</pre>

	<pre> "count": 15, "id": 2, "title": "RPCs for Dummies.", "topic": "distributed systems" }], "message": {}, "status": "Success", "validation_code": 200 } </pre>
--	---

2. Update By ID

Update either the cost, or the count of the book corresponding to the given ID. Also propagates this update by calling the propagate by ID method of other replicas of catalog server.

Request

Method	URL
PUT	/item/<id_>

Query Parameters	Optional	Value Type	Description
id	False	String	The ID of the book whose count or cost needs to be updated. Count and cost must be passed as payload.

Responses

Status	Response
201	<pre>{ "item": {}, "message": {}, "status": "Success", "validation_code": 201 }</pre>

3. Propagate By ID

Propagate an update to either the cost, or the count of the book corresponding to the given ID

Request

Method	URL
PUT	/item/propagate/<id_>

Query Parameters	Optional	Value Type	Description
id	False	String	The ID of the book whose count or cost needs to be updated. Count and cost must be passed as payload.

Responses

Status	Response
201	<pre>{ "item": {}, "message": {}, "status": "Success", "validation_code": 201 }</pre>

	}
--	---

Methods for the Order Server

1. Buy

Check if the book with given ID is available, and then buy it. Upon successful transaction, propagate the update to other replicas of the order server.

Request

Method	URL
GET	/buy/<item_id>

Query Parameters	Optional	Value Type	Description
id	False	String	The id of the book to buy

`http://localhost:8012/buy/1`

Response

Status	Response
--------	----------

200	<pre>{ "message": "Item with id 1 bought successfully.", "order": { "id": 195 }, "status": "Success", "validation_code": 200 }</pre>
404	<pre>{ "message": "The item with id 1 is no longer present in the catalog server", "status": "Failed", "validation_code": 404 }</pre>

2. Add to order

Propagate an update to the database of the order server. This ensures consistency between the databases of each replica of the order server

Request

Method	URL
PUT	/update

Responses

Status	Response
201	<pre>{ "item": {}, "message": {successfully updated order id 2}, "status": "Success", "validation_code": 201 }</pre>

Book-Store

This repository contains implementation of the Lab 3 Project of COMPSCI 677 course at UMass Amherst.

Contributors -

Abhishek Lalwani (alalwani@umass.edu)

Himanshu Gupta (hgupta@umass.edu)

Kunal Chakrabarty (kchakrabarty@umass.edu)

System requirement

Local VM (Linux), Local VM OS(ubuntu), Ec2 servers (Linux)

Installing Required Packages

Installing Python and required dependencies:

1. `sudo apt-get update`
2. `sudo apt-get install python`
3. `sudo apt-get install python-pip`
4. Go to the main folder `pygmy` and run the command: `pip install requirement.txt`

Installing Docker

1. Follow the steps specified in the link:
<https://docs.docker.com/engine/installation/>

Important Source File Descriptions

1. `catalog/catalog.py` implements the catalog server with the relevant GET and PUT methods.
2. `frontend/frontend.py` implements the front end server with the buy, search and lookup methods.

3. `order/order.py` implements the order server with the buy method.
4. `Docs` contains the design documentation and the test case documentation.
5. `requirements.txt` contains the python libraries required.
6. `env.cfg` contains the `PUBLIC_IP` and `PORT` of the machines where the catalog, order and frontend server has to be run. It also contains the reference to the `pem` file that is required to `ssh`, and `scp` to the remote machines. Modify the file according to your requirements.
7. `runme.sh` is a single script to automatically deploy catalog, order and frontend docker servers on specified machines in `env.cfg`, run the `client.py` and get the logs from all the servers.
8. `const.py` contains information about the books in the catalog server.
9. `Docs` contains the design documentation

Please find the instructions below for testing the implementation.

Instructions

To run the server locally

1. Define the ip and ports of order, catalog and front end server by editing the `env.cfg` file. For running locally make the IPs for the server as `http://<public_ip_of_local_vm>`.
2. Now, on your local machine, run the `runme.sh` file which deploy all the dockers and then trigger the `client.py` for starting the traffic. USAGE: `./runme.sh`.
3. You can observe the results of this run in different log files that should be accumulated under the folder `logs`. `client.log` in the main folder will contain the logs of `client.py` script.

To run servers remotely

1. Deploy an Ubuntu VM. We have used the AMI: `ami-013f17f36f8b1fefb` to deploy instances and test our code. You can use any other Ubuntu image as per your convenience. Make sure to install docker engine on the instances. You can follow the link: <https://docs.docker.com/engine/installation/> for the same.
2. Get the private `.pem` file which will be used to communicate to the remote servers by the local machine.

3. Edit the security group to ensure that the ports required by the peers to communicate are open.
4. Set up password-less ssh from the local machine to the ec2 servers by running the following command from the local terminal: `ssh -i <pem_file_path> ubuntu@<ec2_public_ip>` with the private pem file and the public IP address of the EC2 instance that has been set up. (This will add the ec2 server to the `known_hosts` file so that you can ssh from the script without the need of a password). Please note that `ubuntu` is default username of an AWS Ubuntu VM. You can alter it according to your need.
5. Define the ip and ports of order, catalog and front end server by editing the `env.cfg` file. Change the port IDs of the servers as pleased. Also provide the path of the pem file. Instructions to change the file is specified in the same.
6. Now, on your local machine, run the `runme.sh` file which deploy all the dockers and then trigger the `client.py` for starting the traffic. USAGE: `./runme.sh`.
7. You can observe the results of this run in different log files that should be accumulated under the folder `logs`. `client.log` in the main folder will contain the logs of `client.py` script.