DESIGN SPECIFICATION

This project implements a simple book store through a client server model using REST APIs to communicate between layers.

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 consists of two servers:
1. Catalog server: 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
}
```

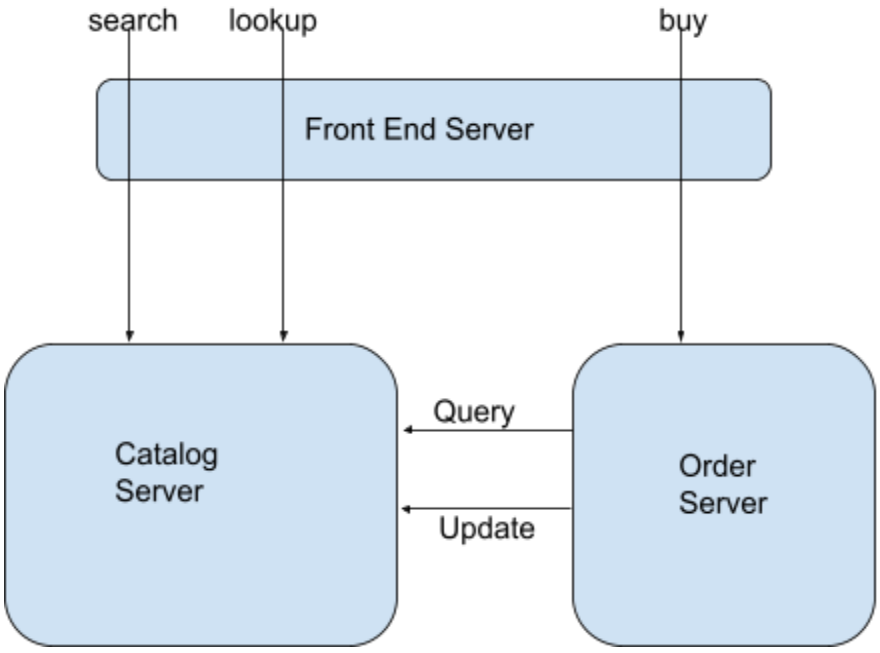
2. Order Server: The order server maintains a list of all orders received for the books.
Following is the schema for order server:

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

The front end server 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 three servers interact with each other is shown below.



For this milestone, we have implemented all three servers to be running on the same machine, but on different ports.

IMPLEMENTATION

Catalog server

The catalog server exposes a GET API (‘item’) which allows for books to be looked up from it’s 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.

Order Server

The order server exposes a GET API(‘buy’) method which looks for a book corresponding to a certain ID and then buy 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 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.

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 it’s 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.

Client Process
A client process is responsible for making calls to the frontend server. This process takes in as argument a parameter based on which, it iteratively calls the lookup, search and buy methods of the front end server (the count for calling these is passed as a parameter to the function) and logs the results of these operations.

Each server is able to locate the IP address and ports of the other two servers from the const.py file and connect with them accordingly.

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 Used	Description
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	{
	  "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
	}

3. Lookup

To lookup a particular item in the server

Request

Method	URL
GET	/lookup

Query	Optional	Value Type	Description

Parameters			
id	False	int	Id of the book to search for

Example localhost:8010/lookup?id=3

Response

Status	Response

200	{
	"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
	}

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	{
	  "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
}
```
 |

2. Update By ID

Update either the cost, or the count of the book corresponding to the given ID

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>

Methods for the Order Server

1. Buy

Check if the book with given ID is available, and then buy it

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>

STEPS TO RUN

System requirement

Local machine (Windows),

Ec2 servers (Linux)

Python module dependencies that we used: Details in `requirements.txt`

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. `runme.py` **is a single script to automatically deploy servers and run frontend APIs.**
7. `const.py` **contains information about the type, IP and Port for all the servers. Change the IP and port of the servers as per requirement.**

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** `const.py` **file. For running locally make the IPs for the server as** `http://12.0.0.1`**.**
2. **Run** `python runme.py -n <number of iterations>`**. For example, if you want the service to run for 20 iterations (where each iteration contains one frontend API of each search,lookup and buy) you use** `python client.py -n 20`**. Default value is 5 iterations. Please note that the servers keep running after the completion of all iterations and have to be stopped manually by the user.**
3. **You can observe the results of this run in the log files.** `client.log` **will contain the logs of** `runme.py` **script. For server specific logs, you can refer to the logs inside the specific server folders named order, catalog and frontend.**

To run servers remotely

1. **Use the custom public AMI:** `ami-07f5352ed5fd844e3` **to deploy instances. The image contains all the library and source code for the respective flask servers to run. If you want to use another image, you have to follow the commands written in the** `Remote Linux setup Commands` **section.**
2. **Create EC2 instances with key pair value, and get the private .pem file.**
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> ec2-user@<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_host file so that you can ssh from the script without the need of a password).

5. Copy the public IP of the instances and add them to the `const.py` file. Change the port IDs of the servers as pleased.
6. Now, on your local machine, run the `runme.py` file which will ssh on the remote machine and start the flask servers. It will then trigger frontend APIs to test the functionalities. USAGE: `python runme.py -pem <pem file used to ssh to all the machines> -n <number of iterations>`. The argument -pem is compulsory in the case the servers are running in remote environment/machines.
7. You can observe the results of this run in different log files. `client.log` will contain the logs of `runme.py` script. For server specific logs, you can refer to the logs inside the specific server folders in remote machines. The location for the same will be: `/home/ec2-user/src/<servername>/<servername>.log`

Remote Linux setup Commands

sudo yum install python3

curl -O https://bootstrap.pypa.io/get-pip.py

python3 get-pip.py --user

pip install flask

pip install gunicorn pip install requests

Command to run the servers:
cd src/<server_name> && gunicorn -b 0.0.0.0:8010 <server_name>:app