Lab 3 – Pygmy, The Book Store

Submission by:

1. Adarsh Kolya - akolya@umass.edu (Spire ID - 33018261)
2. Vignesh Radhakrishna - vradhakrishn@umass.edu (Spire ID - 32577580)
3. Brinda Murulidhara - bmurulidhara@umass.edu (Spire ID - 32578418)

Contents

1. Introduction

The objective of this project is to implement a multi-tier web application. The web application is a simple version of a book-store wherein users can search for the availability of books by topic or using the unique identifier assigned to every book. Users can then buy books of their choice depending on the availability. REST APIs are exposed to perform basic CRUD operations as part of the backend and frontend microservices. Flask, a light-weight web framework is used to create the distributed web application. SQLite is used as the backend database.

1.1 Goals

1. Build the backend and frontend components of a two-tier web application wherein microservices are used at each tier. Expose the search, lookup, and buy functionalities as REST APIs in each microservice, wherein the client hits the REST endpoint, and the server services the client request.

2. Replication – The order server and catalog servers are replicated, and the load is evenly distributed across these servers. This not only ensures that the time the system needs to process their requests is reduced but also provides a way to continue serving requests even when one or more servers go down.

3. Caching – The responses to lookup requests are cached in the frontend server. This reduces the time taken to respond to subsequent requests for the same book item.

4. Consistency – The data consistency across replicas is ensured through propagation of requests from primary (server receiving the write request) to replicas. The cache in the front-end server is invalidated every time there is a database update request on an item in the cache.

5. Fault tolerance – The system is made fault tolerant using replicas and heartbeat mechanism to identify failed nodes. Client requests are forwarded only to non-faulty nodes and the faulty nodes re-sync data by contacting working replicas once they begin operating normally after a crash.

6. Containerization and docker – The entire application is bundled and executed in an isolated environment. This ensures that the application with all the required libraries and other dependencies is deployed as one package.

2. Components and Interfaces

The front-end server implements the following interfaces in addition to the interfaces provided as part of Lab 2.

2.1 cache(itemNum)

This is an interface provided to the catalog server to invalidate a specific item in the front-end server's cache when it is updated in the catalog database. We use a server push model to remove stale cache entries. The catalog server invalidates the specific entry in the cache when there is a write request on the database. If the same item is looked up again, the front-end server caches the response. Subsequent lookup requests for the same item do not result in calls to the catalog server as the cached responses are returned. We design this using REST client/server architecture, and a high-level design is as below:

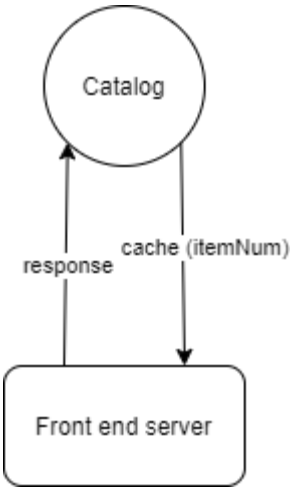


Figure 1 - Implementation details of the cache invalidation interface

2.2 Load balancer component

The front-end server acts as a load balancer and evenly distributes the incoming requests across the order and catalog servers using round-robin technique.

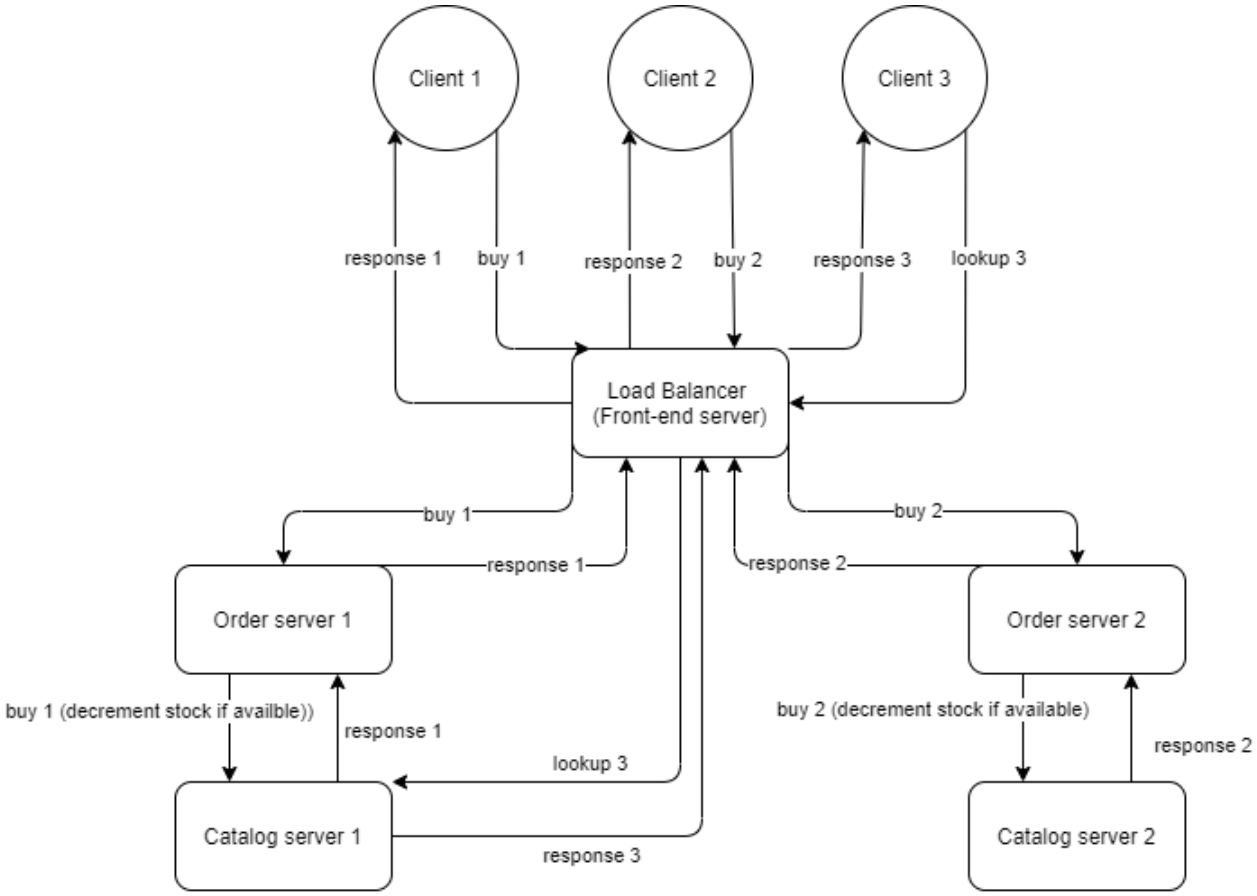


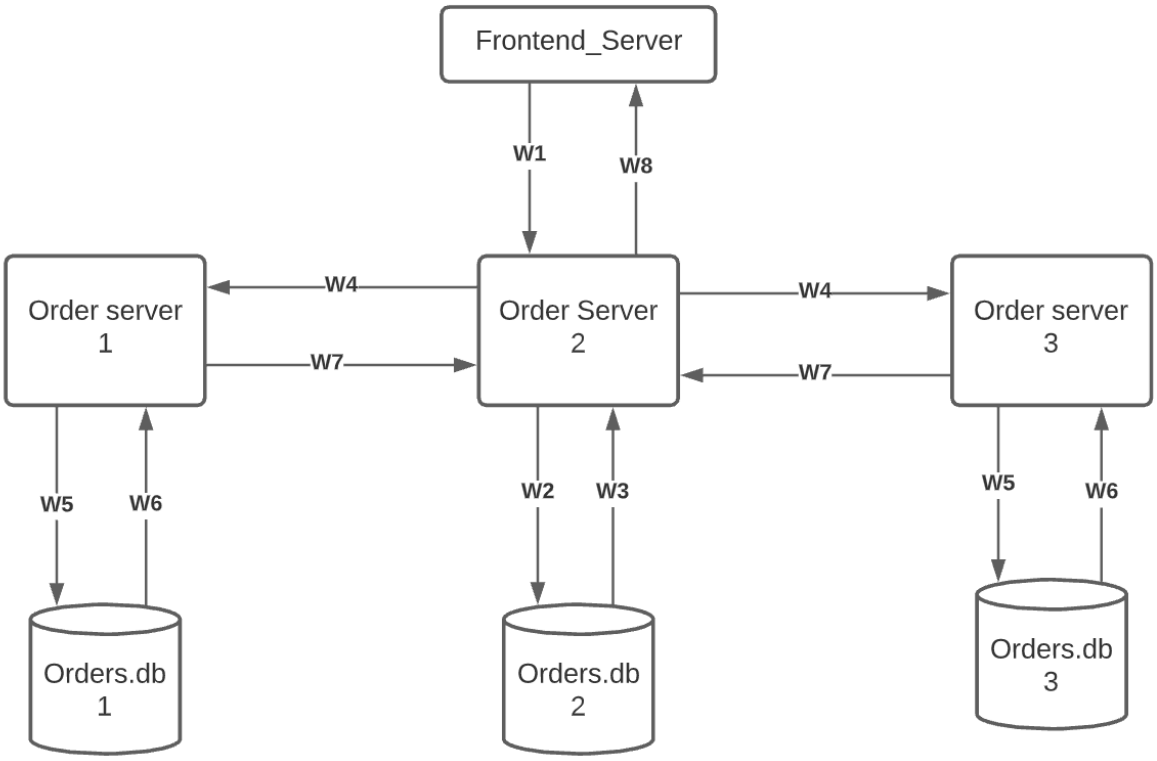
Figure 2 - Load balancer component

2.3 Fault Tolerance

The front-end server checks if the replicas are alive by sending heart beats requests at regular intervals. Depending on the status of the replica servers, the front-end server performs load balancing and forwards requests to non-fault replicas.

2.4 Ensuring consistency among the replicas

To ensure that the order server and catalog server replicas are in sync with each other, we have followed **Local Write with Replication** protocol among the replicas. In this model the primary copy migrates between the process that want to perform the write. Once the writes have been performed on the primary, this is propagated onto the other replicas as well as shown below:



W1 --> Buy request from the front-end server.
W2 --> Write to the new primary for orders.db i.e, Orders.db 2.
W3 --> Response for W2.
W4 --> Propagation of writes to the replica from the primary server.
W5 --> Writes on the replica databases.
W6 --> Response for W5.
W7 --> Response for W4.
W8 --> Response for W1.

Figure 3 - Local write with replication protocol to ensure consistency among replicas

2.5 Crash recovery

When a replica crashes and then later comes back online, we have a crash recovery mechanism that will help the crashed replica come in sync with the other replicas that were online. The backend servers will have 2 states – INIT and RUNNING. Every server will be in INIT state when it comes up for the first time. As soon as a write is performed on the server's DB, the state is changed to RUNNING. RUNNING state simply means that the server has performed some writes.

Crash recovery logic makes use of these states of the servers and a server synchronizes itself after a crash as described below:

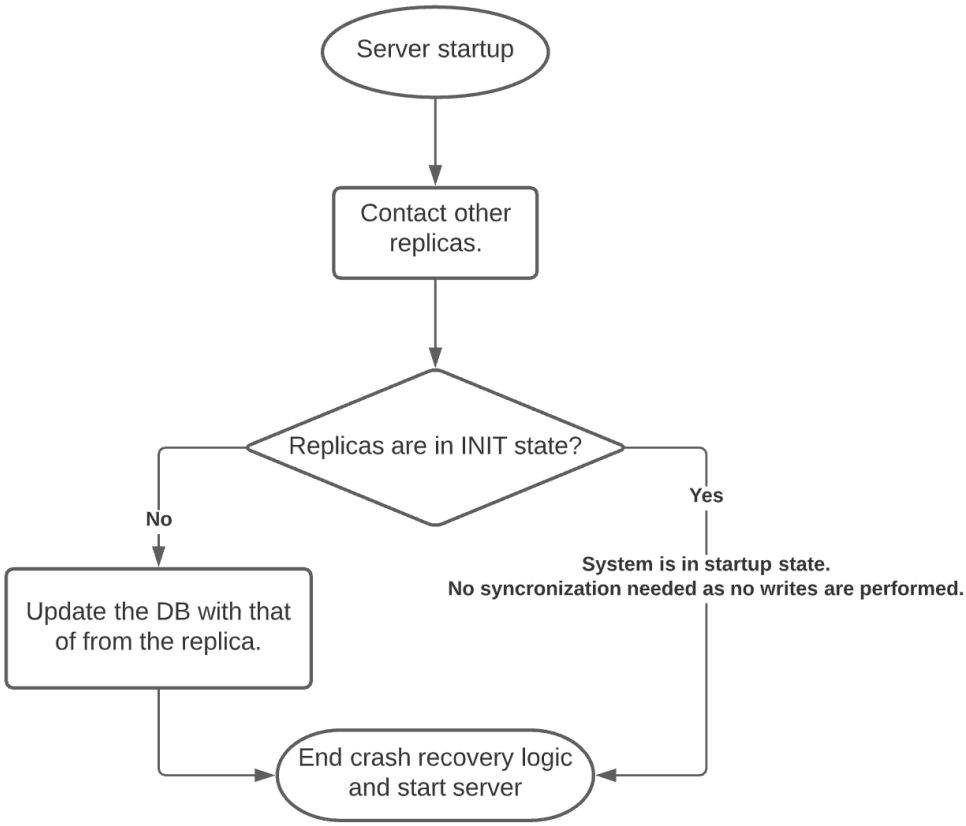


Figure 4 - Crash recovery mechanism

3. How to run the system?

To run the system on localhost

Run 2 catalog servers, 2 order servers and 1 frontend server on localhost

```
cd Lab-3-Pygmy-The-book-store
bash run.sh
```

To run the system on AWS EC2 instances

Create a config file. This file should have 3 lines.

1st line will contain comma separated list of DNS name/IP of servers where catalog server replicas should run.

2nd line will contain comma separated list of DNS name/IP of servers where order server replicas should run.

3rd line will contain DNS name/IP of server where frontend server should run.

Example :

```
cat config
ec2-52-204-17-35.compute-1.amazonaws.com,ec2-54-146-69-77.compute-1.amazonaws.com,ec2-54-
152-5-125.compute-1.amazonaws.com
ec2-54-157-158-136.compute-1.amazonaws.com,ec2-52-23-179-14.compute-1.amazonaws.com,ec2-
107-23-14-212.compute-1.amazonaws.com
ec2-54-237-55-198.compute-1.amazonaws.com
```

With the above config file 3 replicas of catalog server will run on servers listed on line 1. 3 replicas of order server will run on servers listed on line 2. And front end server will run on server on line 3.

Please ensure there is no extra space in between.

Login to a server that has passwordless ssh set up to all the above servers. Clone the repo and do the below :

```
cd Lab-3-Pygmy-The-book-store
bash run.sh config
```

Usage

CLI

Open a new shell and execute the below commands.

```
cd LAB-3-PYGMY-THE-BOOK-STORE
# For linux based
pip install virtualenv
virtualenv .venv
source .venv/bin/activate

pip install -r src/requirements.txt
source .venv/bin/activate

cd src/cli
```

Lookup

```
python main.py --frontend_server <ip_of_frontend_server> lookup <item number>
```

Note: --frontend_server is optional. If not provided, it takes localhost as default.

Example

```
python main.py --frontend_server ec2-35-175-129-185.compute-1.amazonaws.com lookup 1
```

Search

```
python main.py --frontend_server <ip_of_frontend_server> search --topic "<topic>"
```

Example

```
python main.py search --topic "distributed systems"
```

Buy

```
python main.py --frontend_server <ip_of_frontend_server> buy <item number>
```

Example

```
python main.py buy 1
```

Note: For more details refer “How to run the system?” section in the design document for Lab 2.

4. Fault tolerance

Goals of the Experiment

1. Show that when one of the catalog server is down and the frontend server sends request to it, the request is forwarded to an active catalog server by the load balancer.

2. Show that when one of the order server is down and the frontend server sends request to it, the request is forwarded to an active order server by the load balancer.

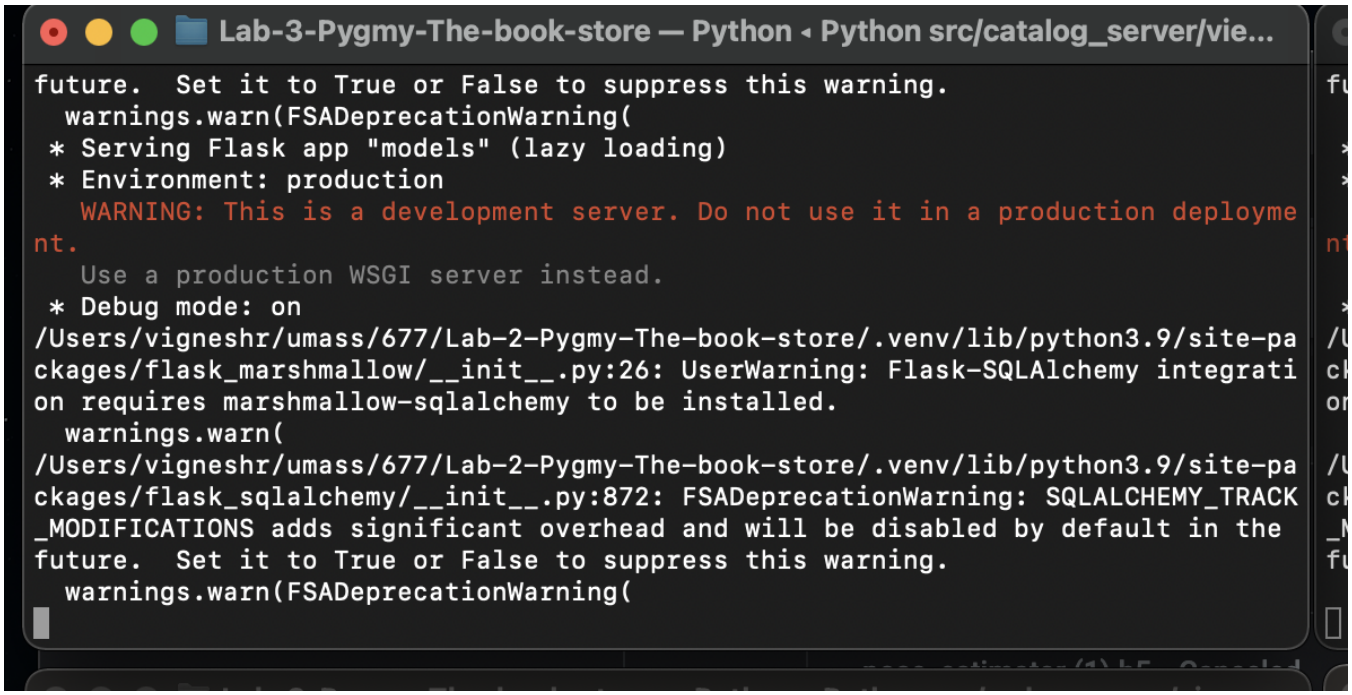
3. Show that when the crashed catalog server recovers, it syncs its data base with the other catalog servers.

4. Show that when the crashed order server recovers, it syncs its data base with the other order servers.

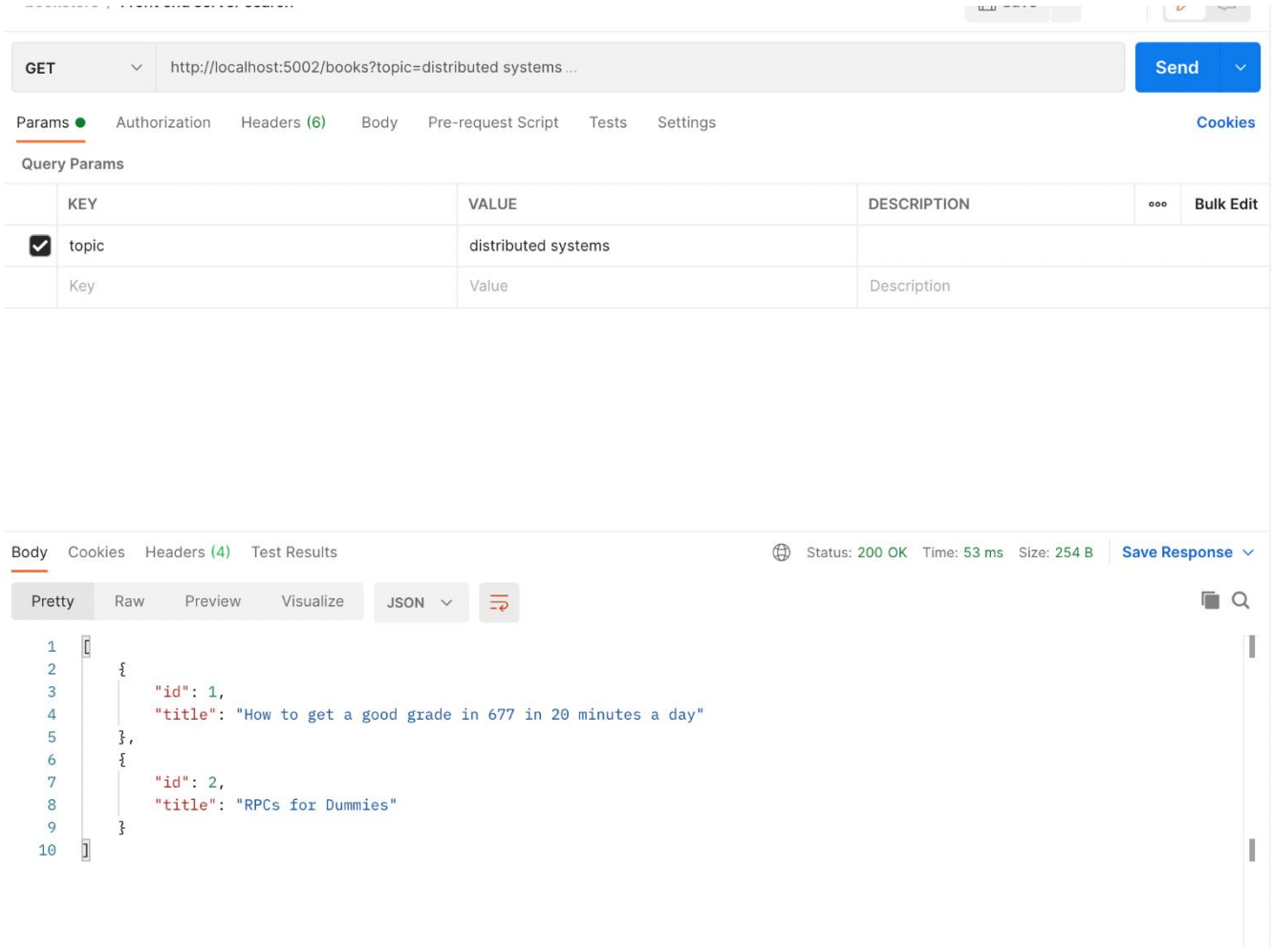
5. Show that the frontend server periodically checks the health of the backend servers.

Sequence of operations for 1:

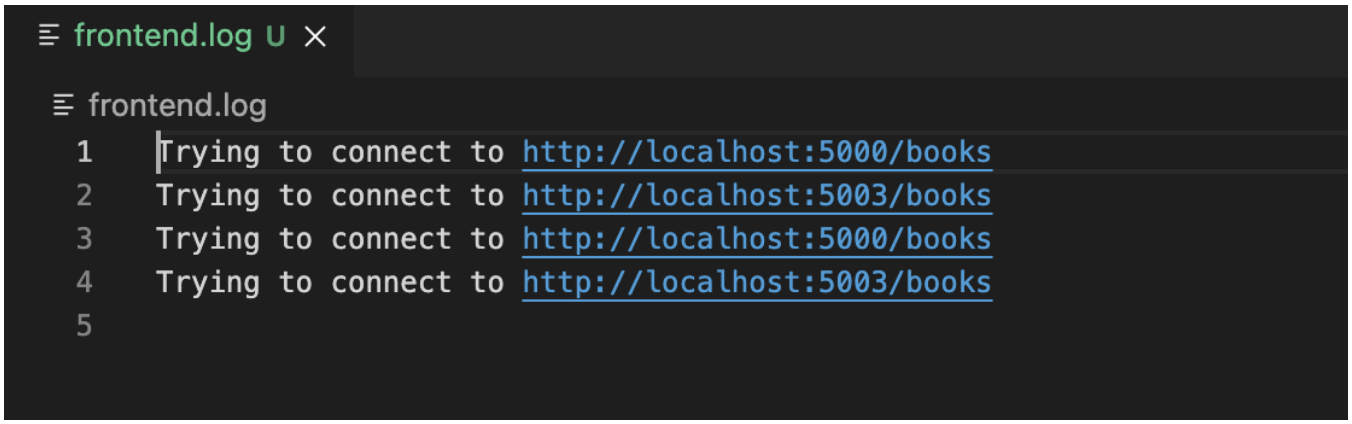
1.1 Ensure all the servers are running.



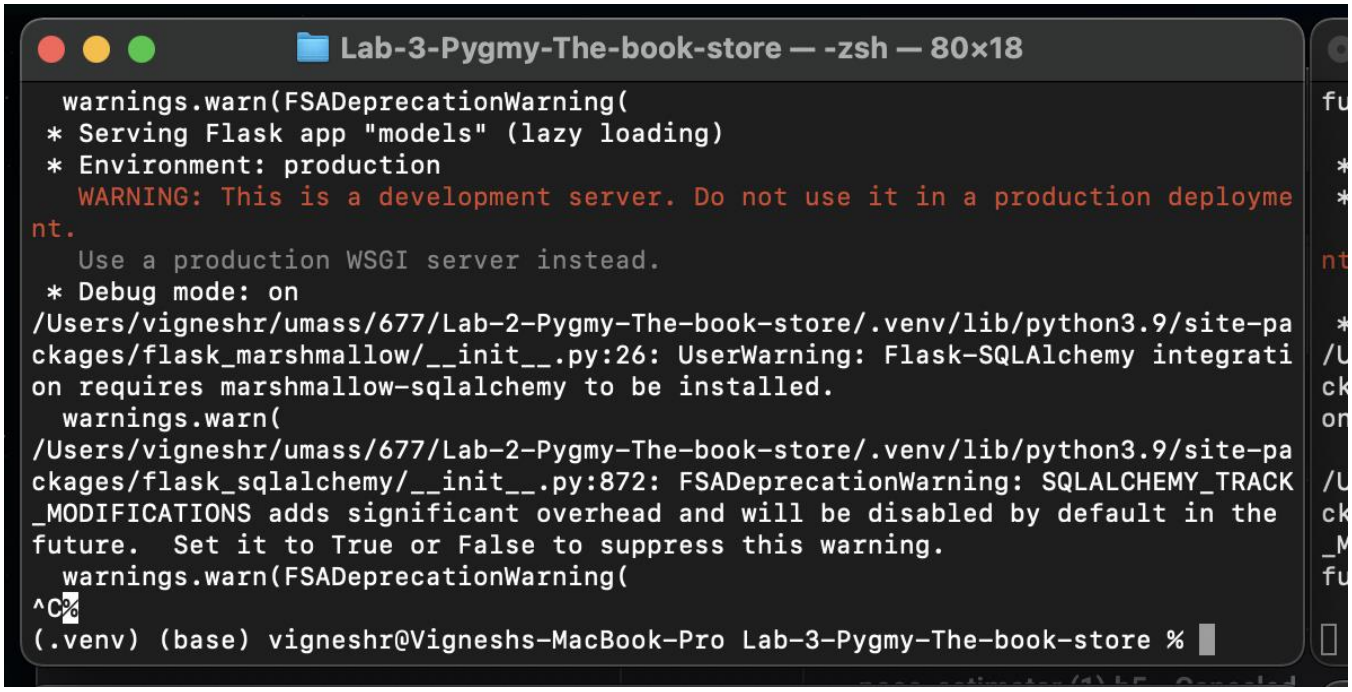
1.2 Trigger 4 search requests to the frontend server.



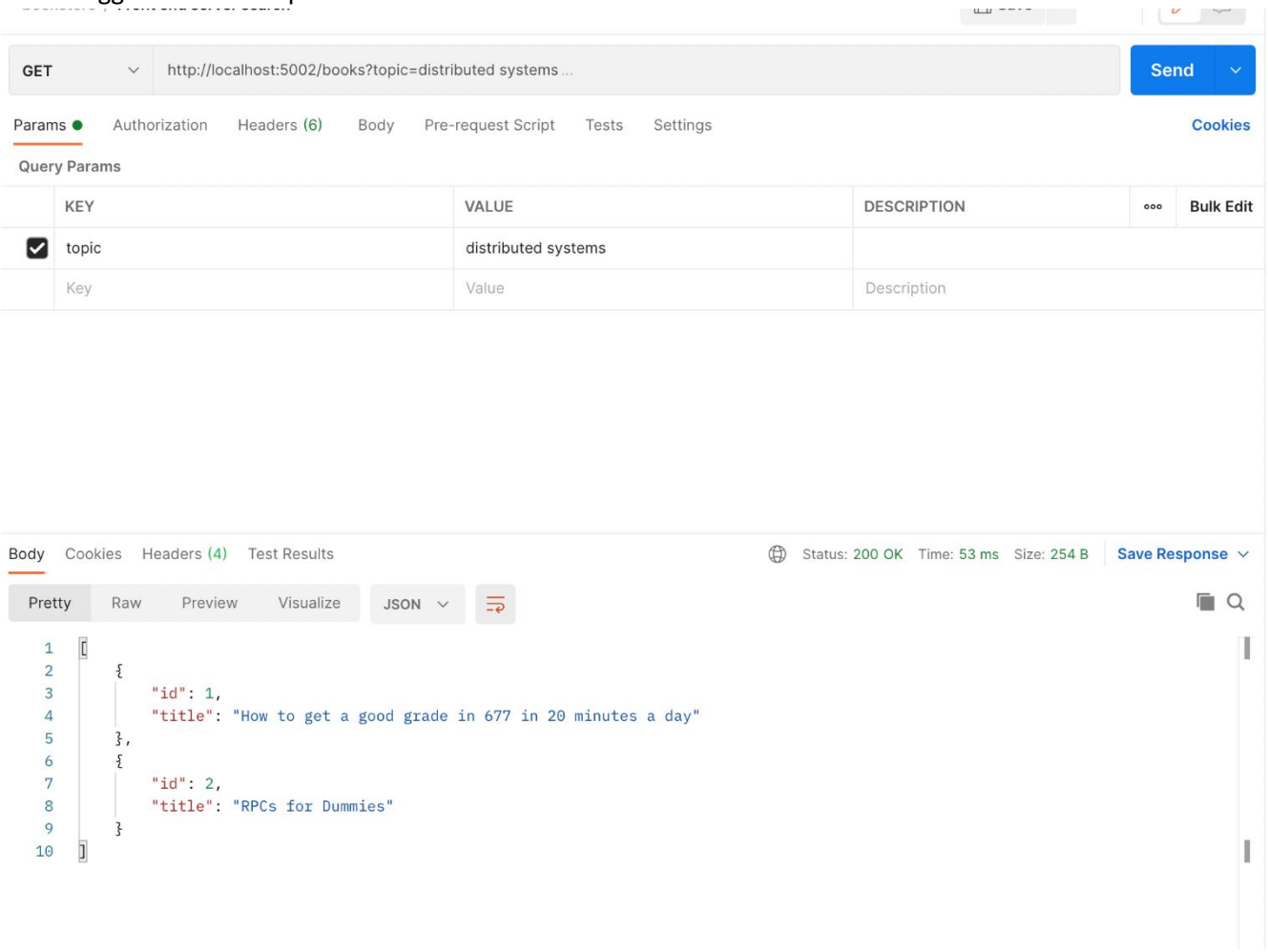
1.3 Check the frontend.log and note that the requests are evenly distributed among catalog servers (5000 and 5003).



1.4 Stop the first catalog server (5000).

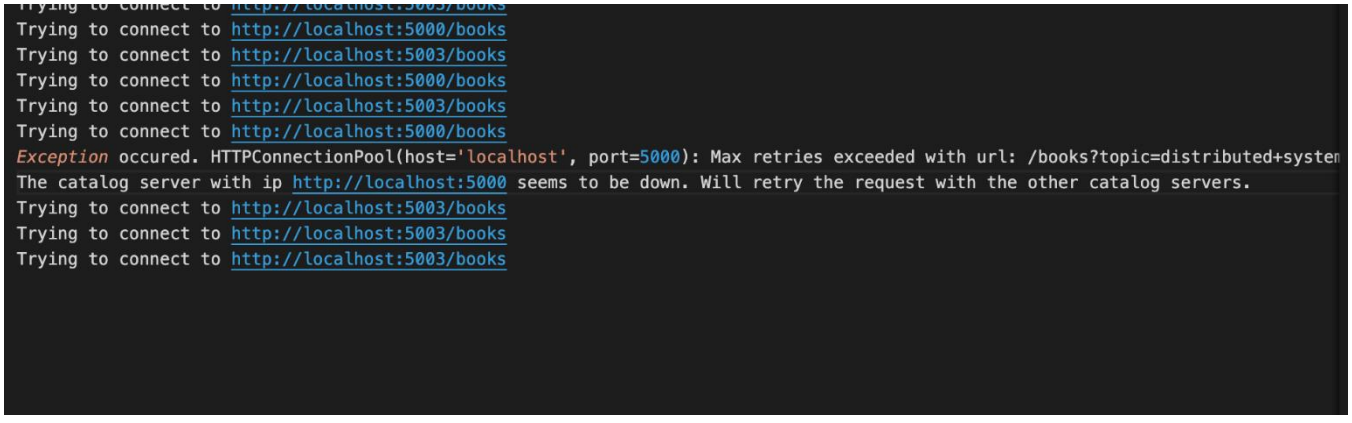

Lab-3-Pygmy-The-book-store — -zsh — 80×18
warnings.warn(FSADeprecationWarning(
* Serving Flask app "models" (lazy loading)
* Environment: production
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Debug mode: on
/Users/vigneshr/umass/677/Lab-2-Pygmy-The-book-store/.venv/lib/python3.9/site-packages/flask_marshmallow/__init__.py:26: UserWarning: Flask-SQLAlchemy integration requires marshmallow-sqlalchemy to be installed.
warnings.warn(
/Users/vigneshr/umass/677/Lab-2-Pygmy-The-book-store/.venv/lib/python3.9/site-packages/flask_sqlalchemy/__init__.py:872: FSADeprecationWarning: SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and will be disabled by default in the future. Set it to True or False to suppress this warning.
warnings.warn(FSADeprecationWarning(
^C%
(.venv) (base) vigneshr@Vigneshs-MacBook-Pro Lab-3-Pygmy-The-book-store %

1.5 Trigger 4 search requests to the frontend server.



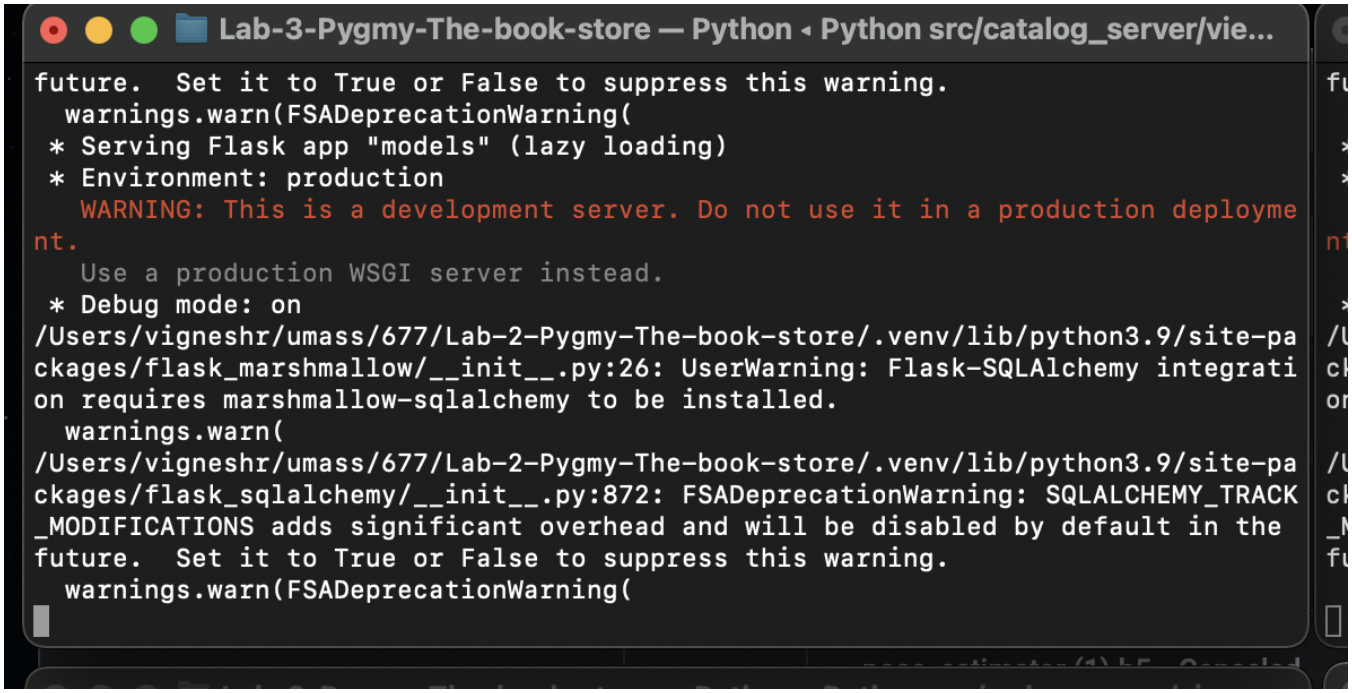
1.6 Check the frontend.log and note an exception log saying the first catalog server is down.

Also note that the subsequent 4 requests are all sent to the other catalog server (5003).

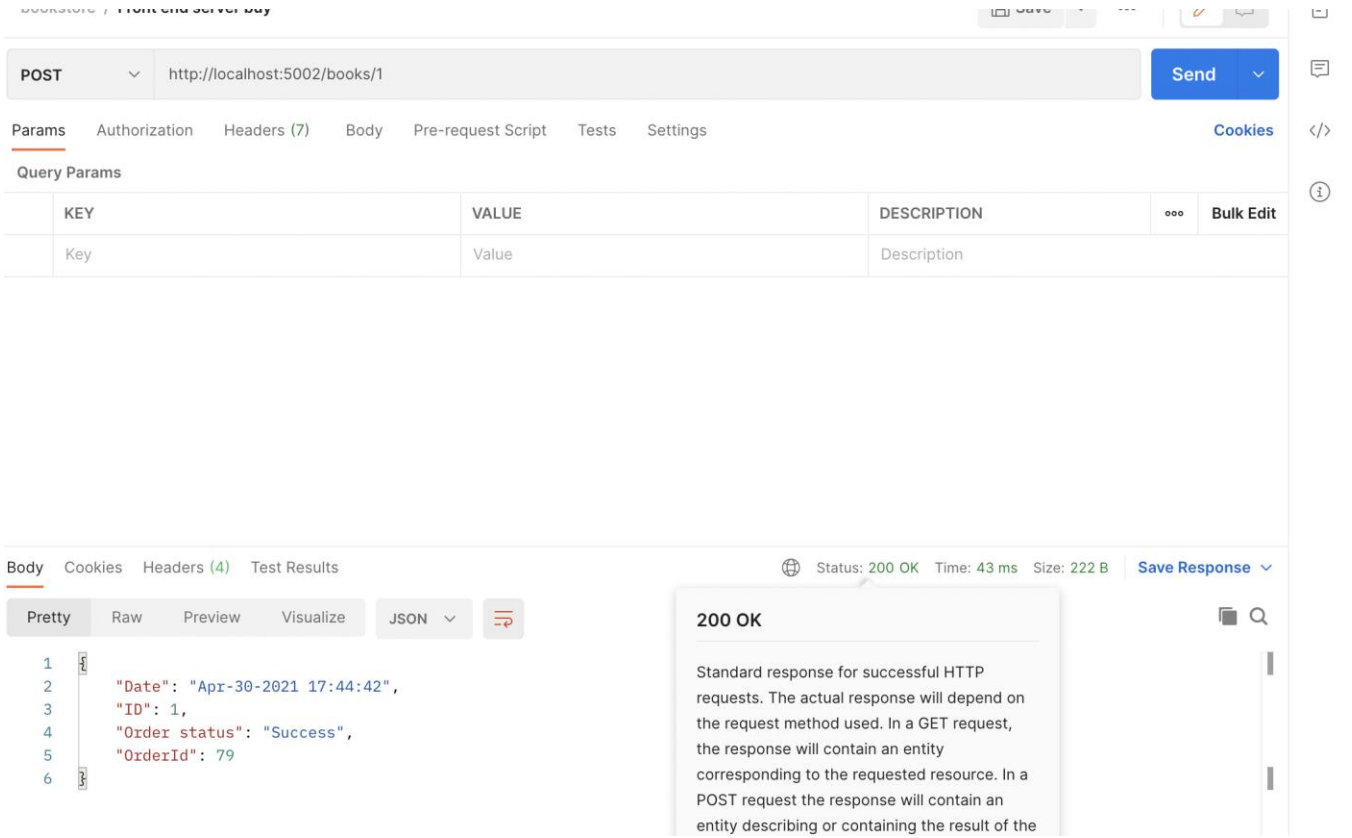


Sequence of operations for 2:

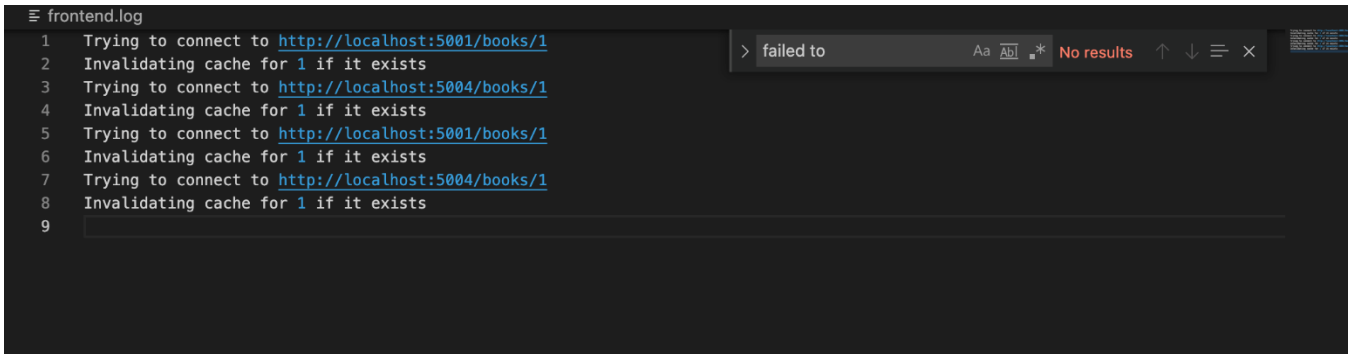
2.1 Ensure all the servers are running.



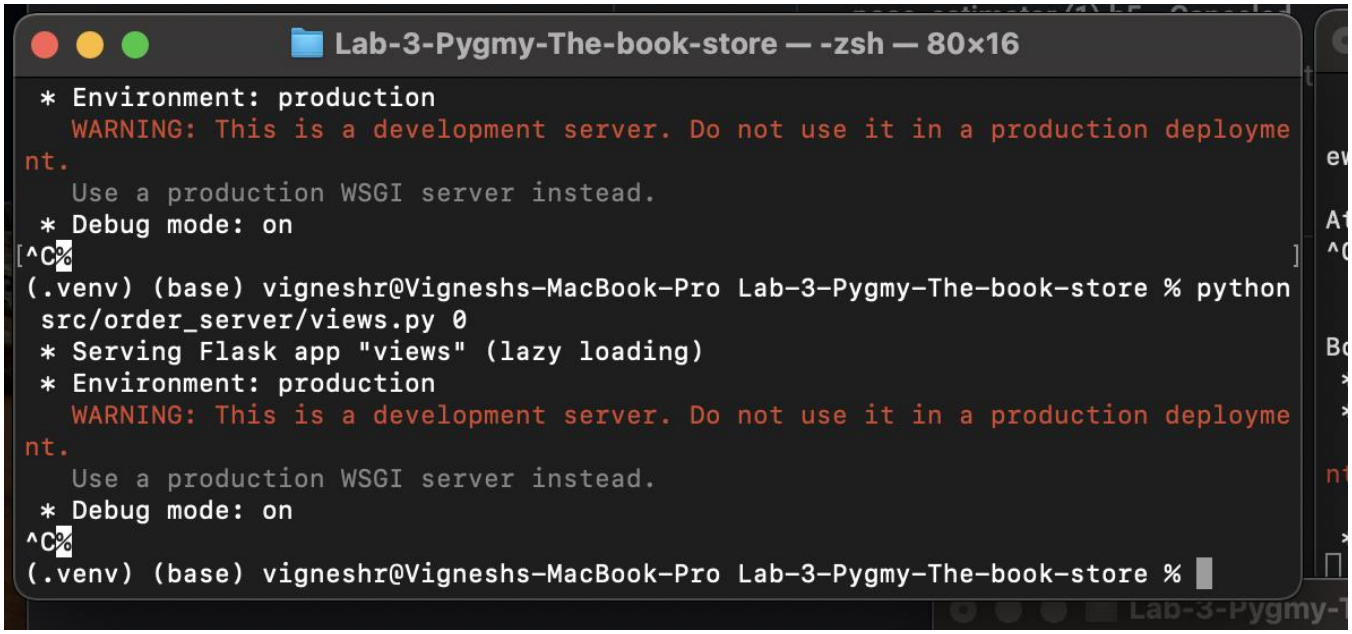
2.2 Trigger 4 buy requests to the frontend server.



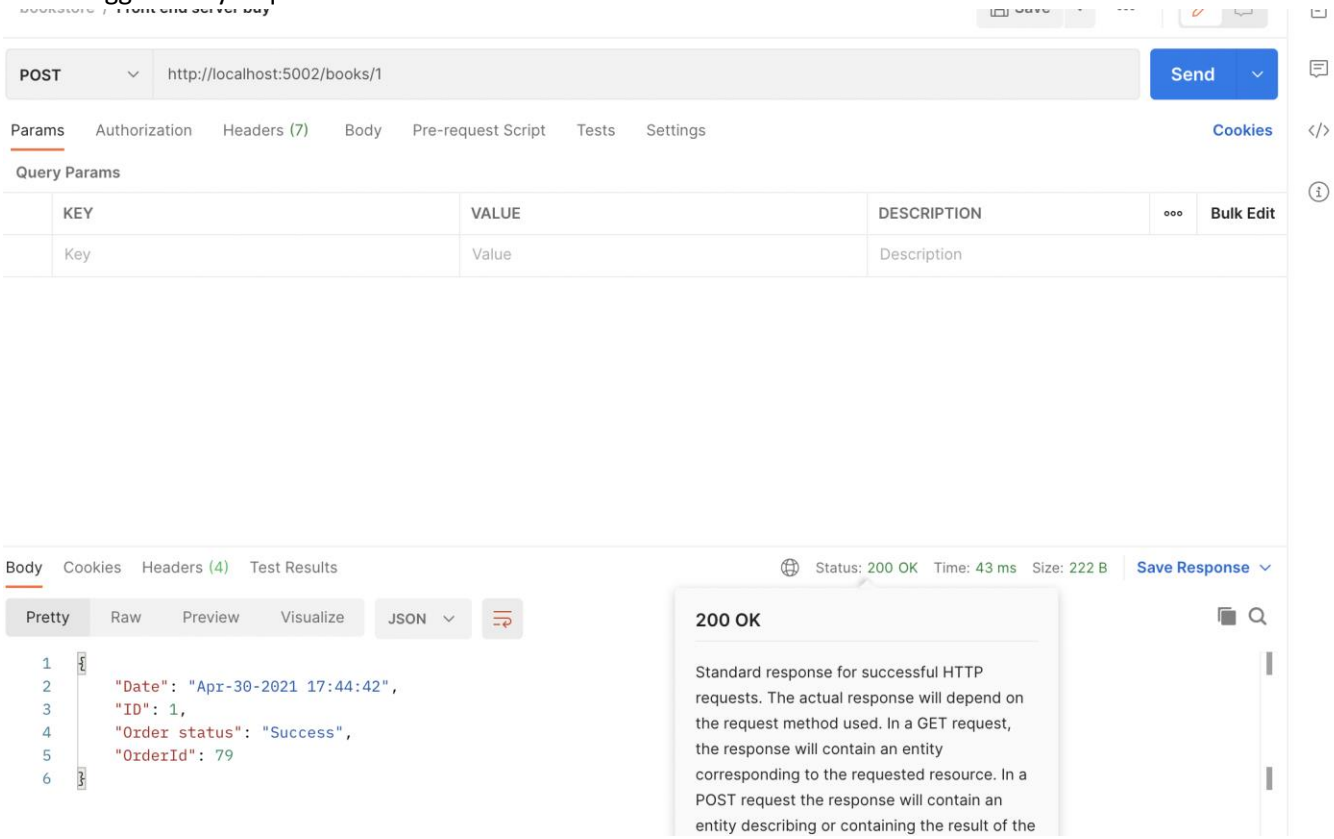
2.3 Check the frontend.log and note that the requests are evenly distributed among order servers (5001 and 5004).



2.4 Stop the first order server (5001)

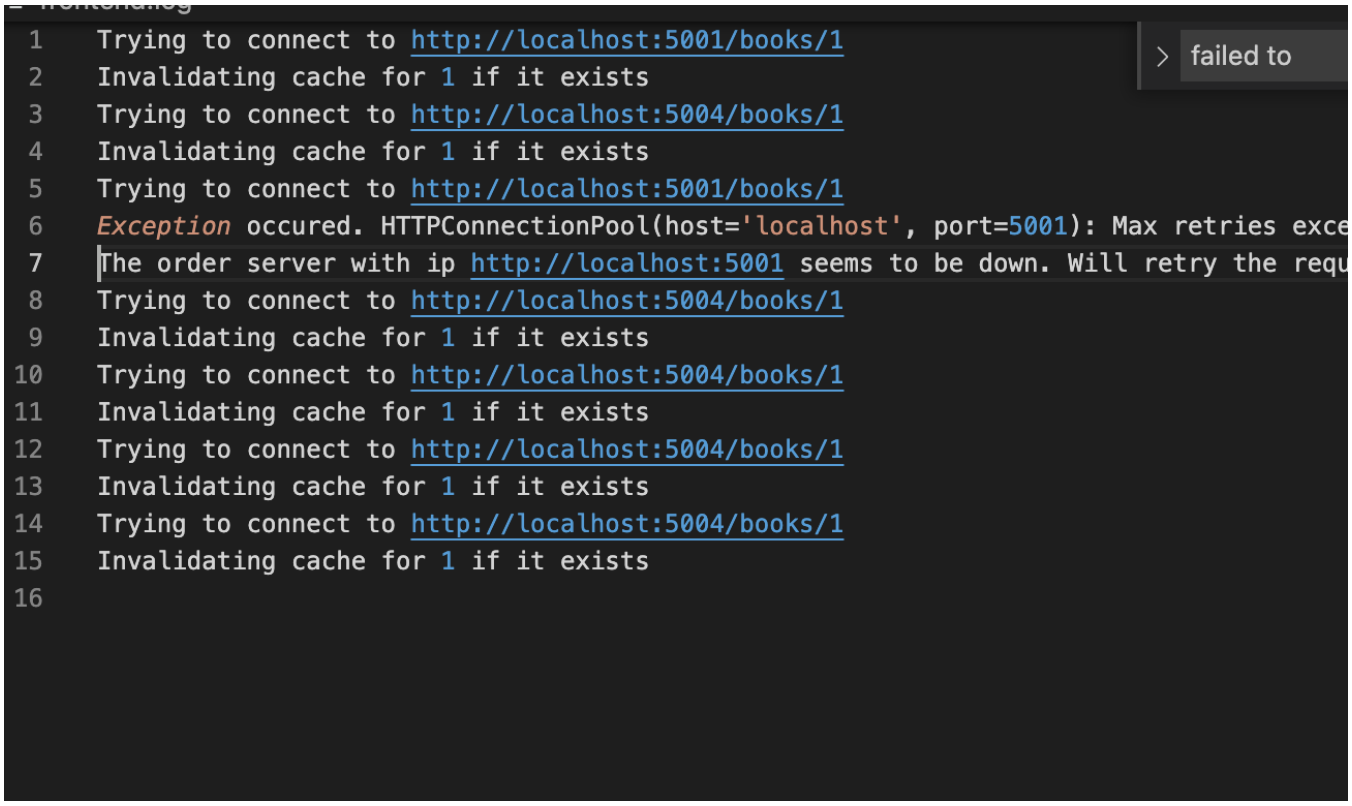


2.5 Trigger 4 buy requests to the frontend server.



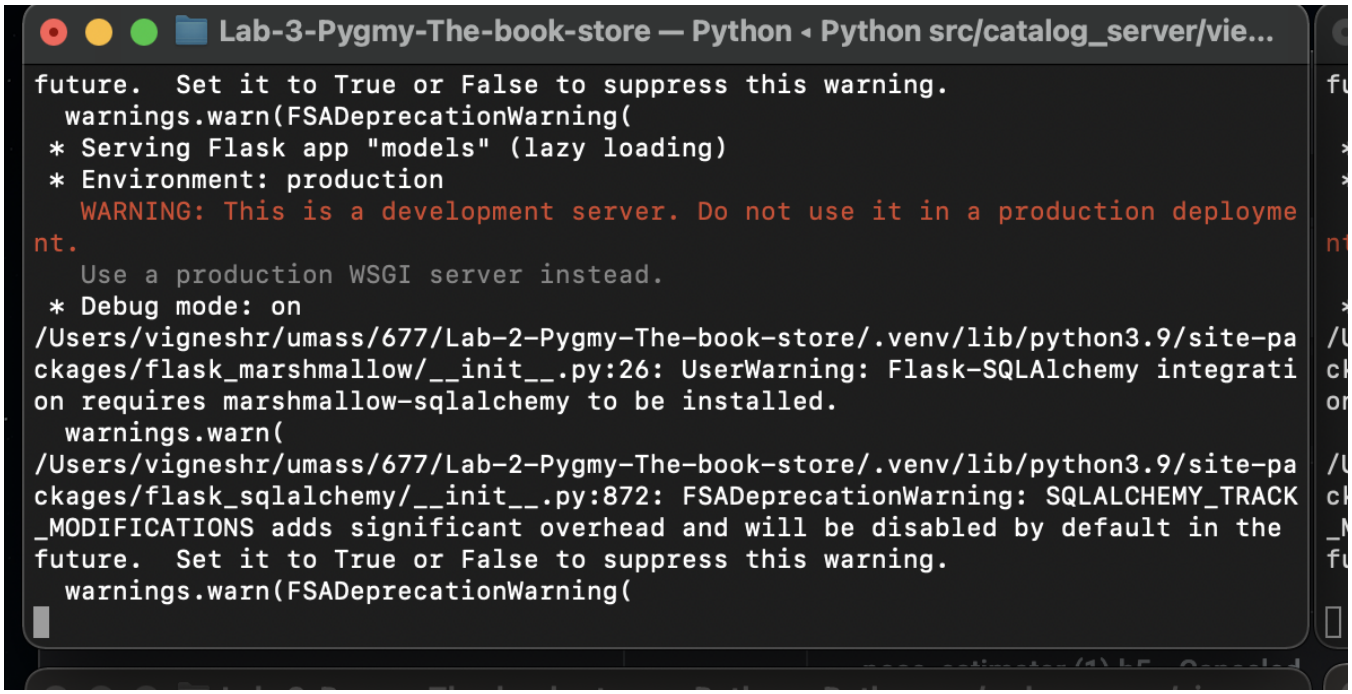
2.6 Check the frontend.log and note an exception log saying the first order server is down.

Also note that the subsequent 4 requests are all sent to the other order server (5003).

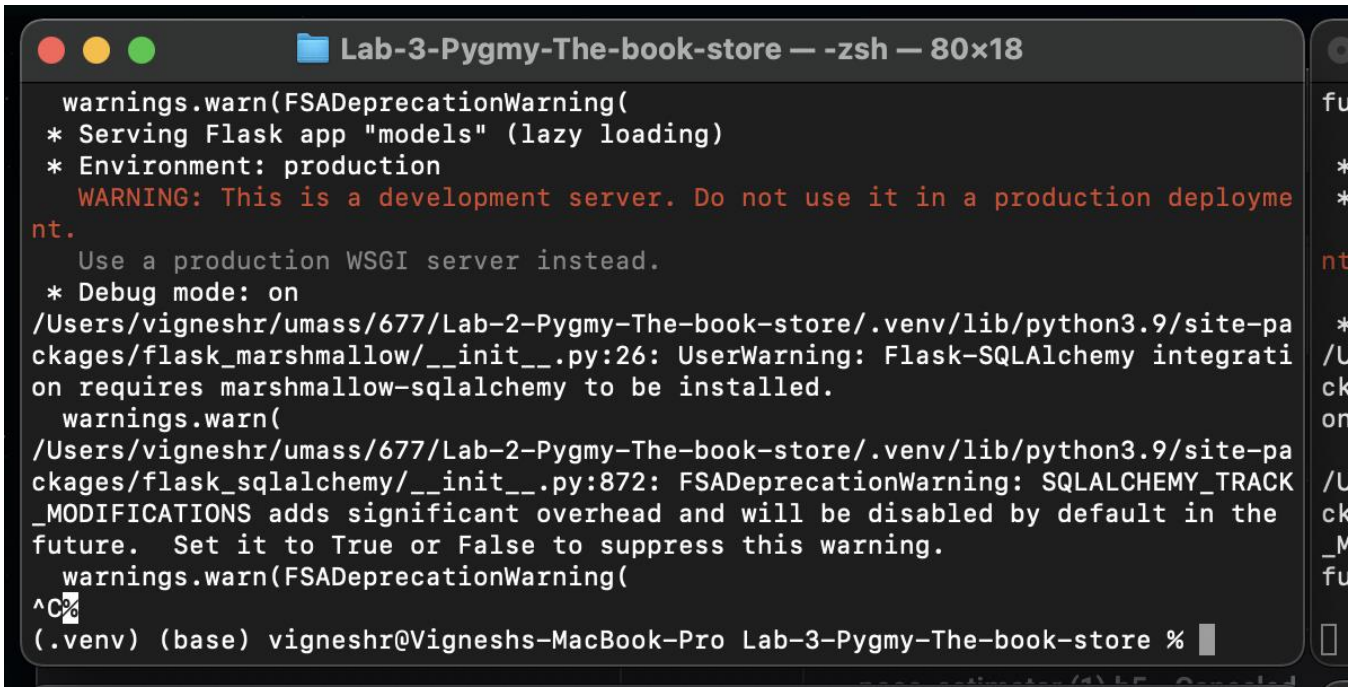


Sequence of operations for 3:

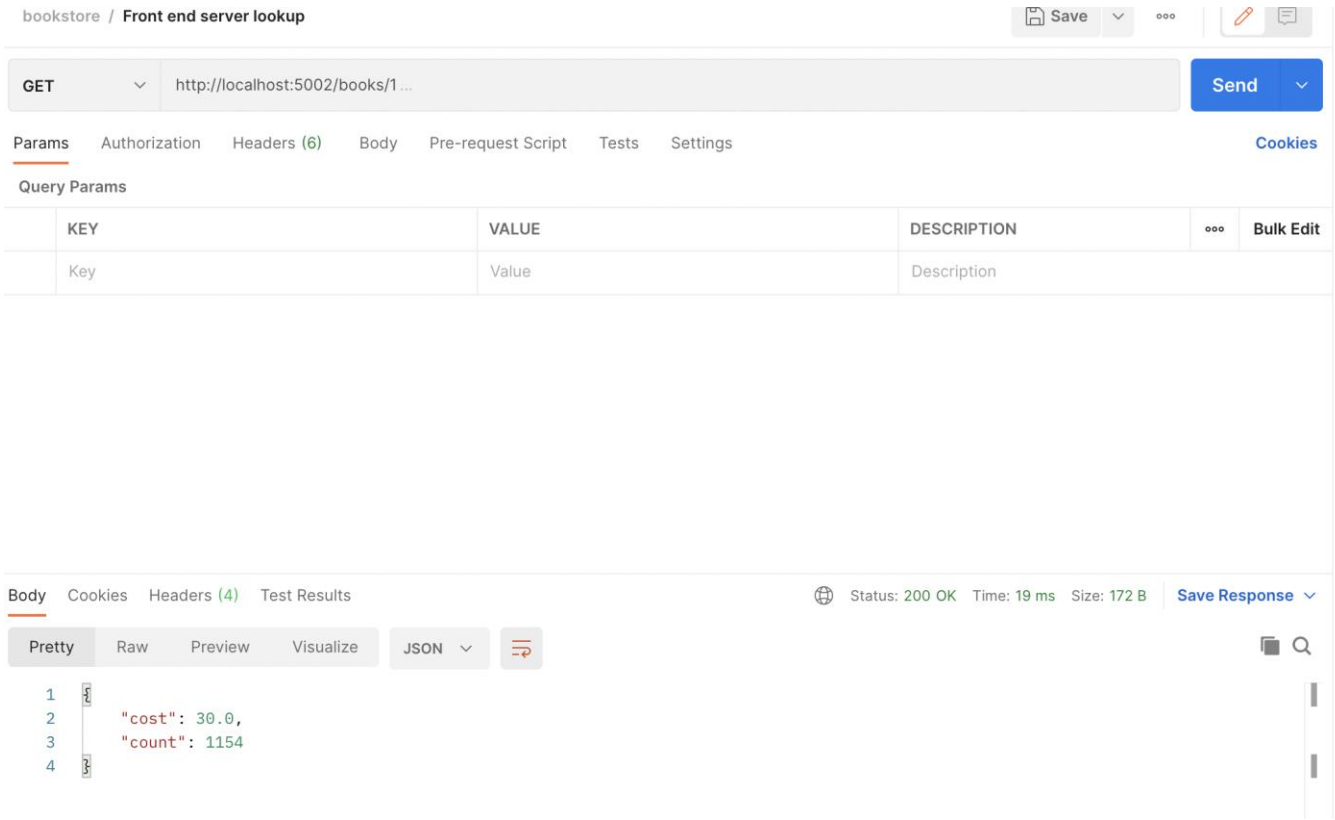
3.1 Ensure all the servers are running.



3.2. Stop the first catalog server (5000)



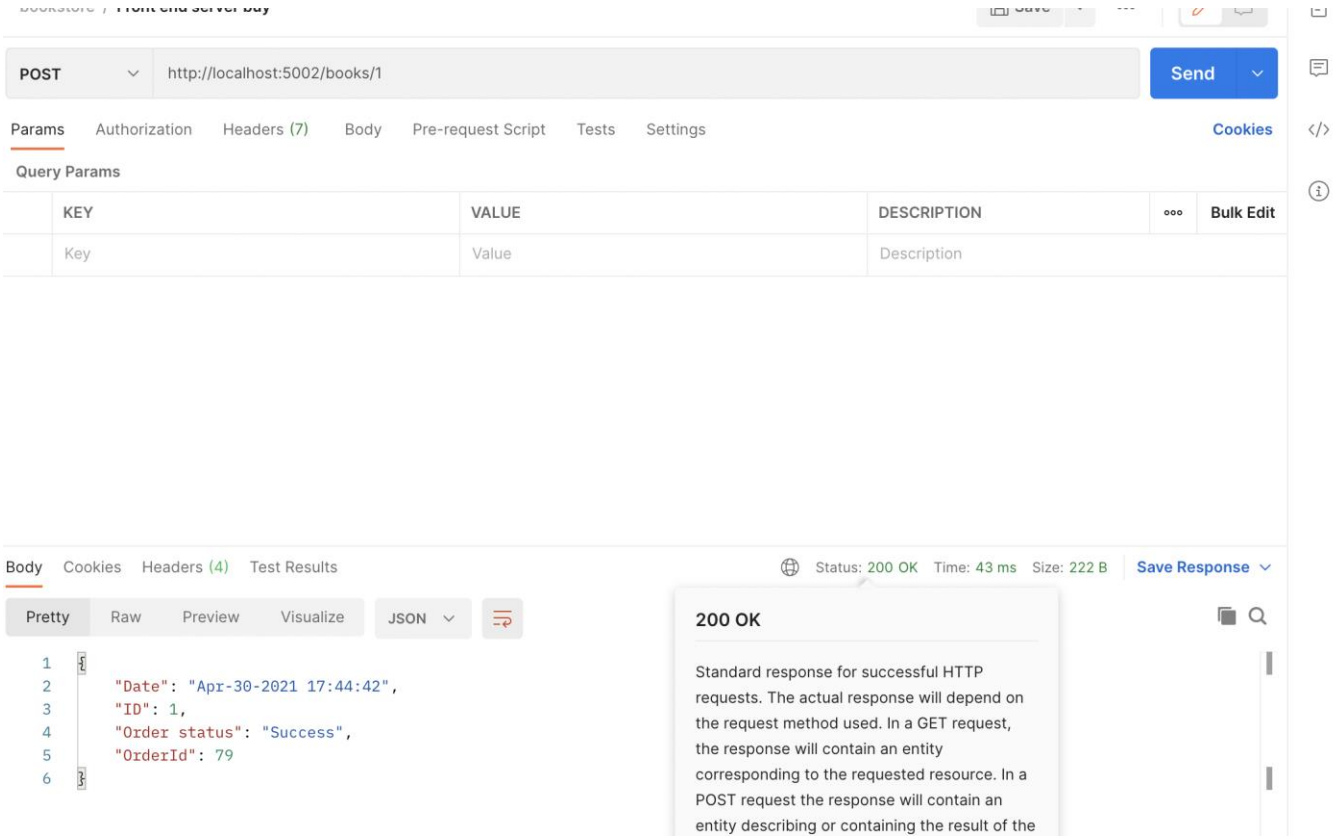
3.3 Trigger a lookup request to the frontend server and note the count of the book (1154).



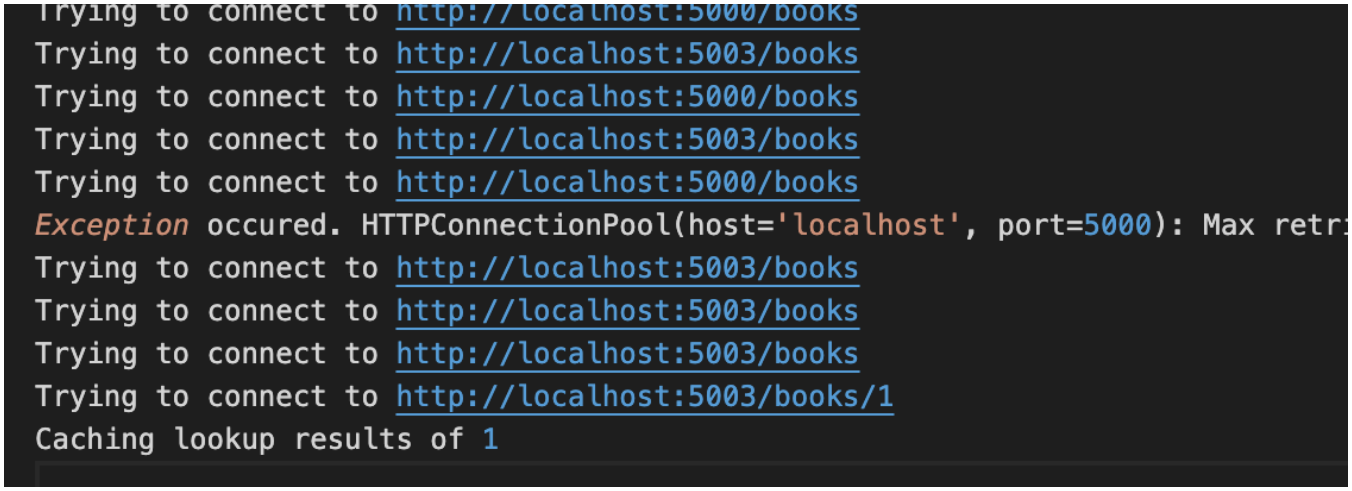
3.4 Note in the frontend.log file that the request was sent to the other catalog server (5003).

```
Trying to connect to http://localhost:5000/books
Trying to connect to http://localhost:5003/books
Trying to connect to http://localhost:5000/books
Trying to connect to http://localhost:5003/books
Trying to connect to http://localhost:5000/books
Exception occured. HTTPConnectionPool(host='localhost', port=5000): Max retr
Trying to connect to http://localhost:5003/books
Trying to connect to http://localhost:5003/books
Trying to connect to http://localhost:5003/books
Trying to connect to http://localhost:5003/books/1
Caching lookup results of 1
```

3.5 Trigger a buy request to the frontend server. The expected count is now 1153.



3.6 Note that the request is again sent to the only catalog server (5003).

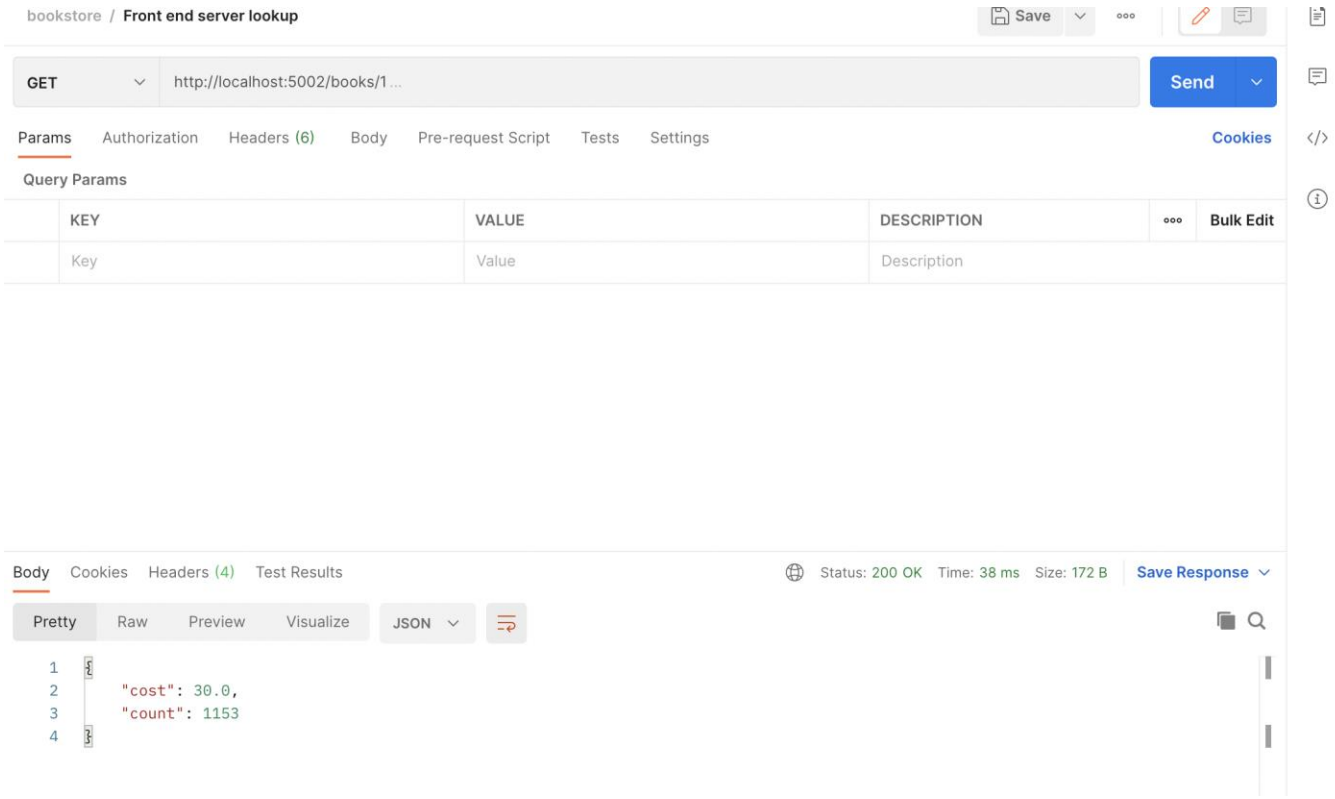


3.7 Start the first catalog server.

3.8 Check the catalog.log to verify that the catalog server syncs itself by talking to the other catalog server (5003)

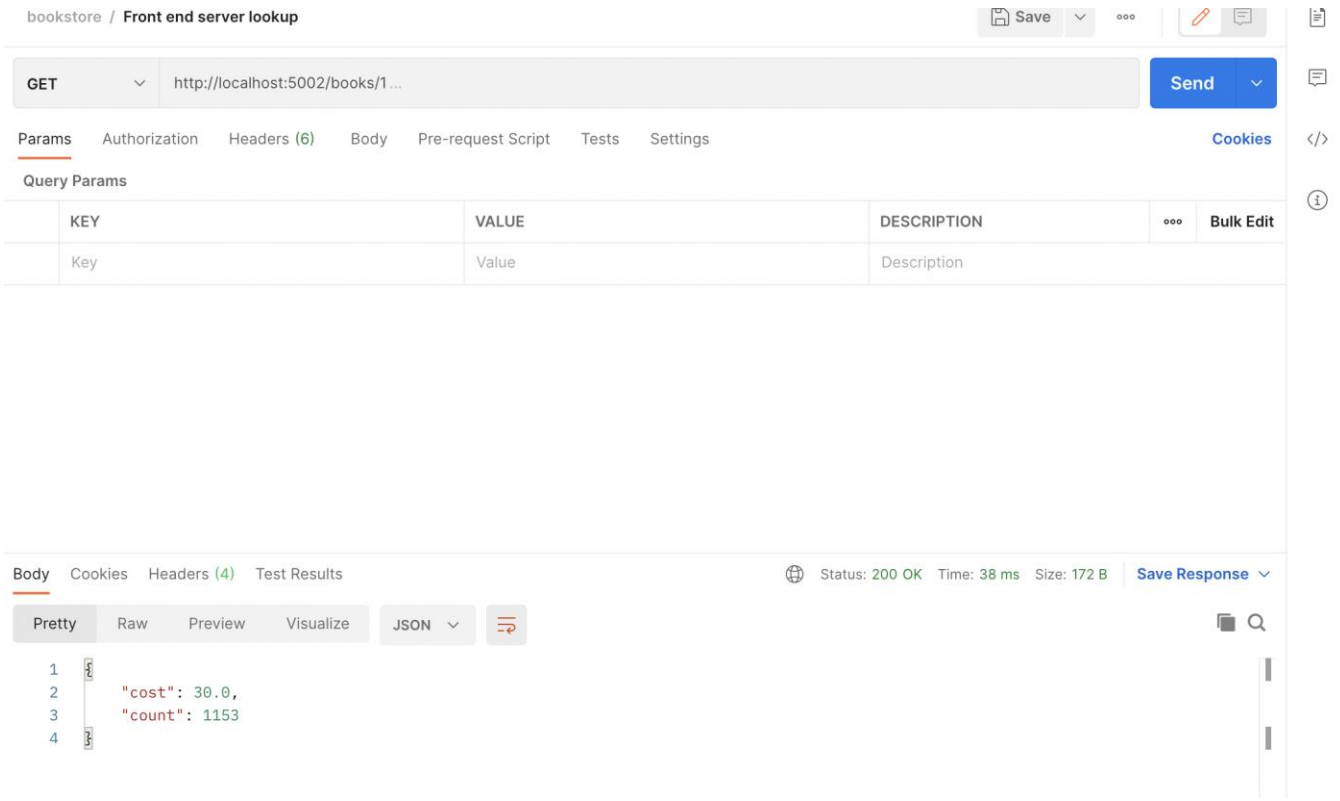
```
INFO:werkzeug: * Restarting with stat
INFO:root:Trying to sync with replica with node num 0 - http://localhost:5003
DEBUG:urllib3.connectionpool:Starting new HTTP connection (1): localhost:5003
INFO:werkzeug:127.0.0.1 - - [30/Apr/2021 15:17:51] " [37mGET /table HTTP/1.1 [0m" 200 -
DEBUG:urllib3.connectionpool:http://localhost:5003 "GET /table HTTP/1.1" 200 1036
INFO:root:Replica 0 - http://localhost:5003 is in RUNNING state. We will sync our DB with that of the replica
```

3.9 Trigger a lookup request to the frontend server.



3.10 frontend.log shows that the request was now sent to the recovered server since it is back.

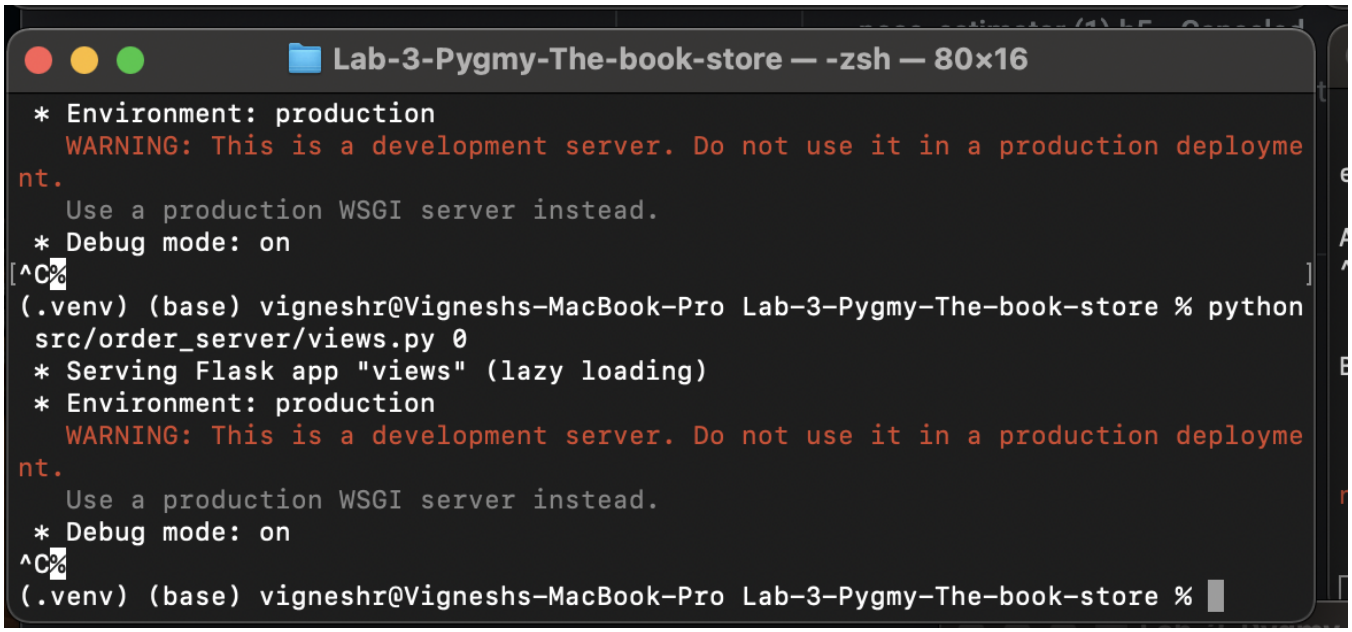
3.11 Check the output of the request and verify that the count is as expected (1153) in 3.4.



Sequence of operations for 4:

4.1 Ensure all the servers are running.

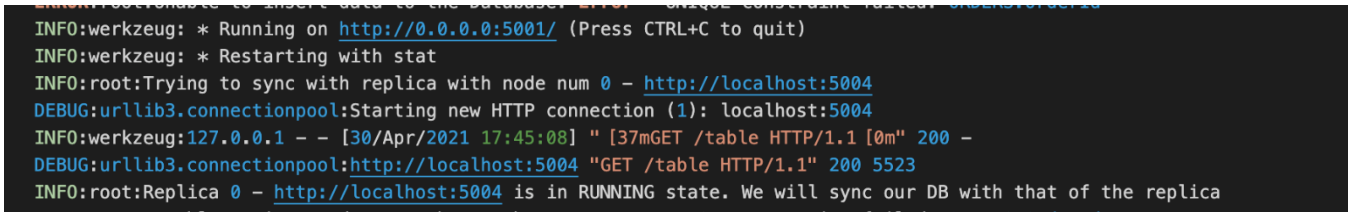
4.2. Stop the first order server (5001).



4.3 Trigger a buy request to the frontend server.

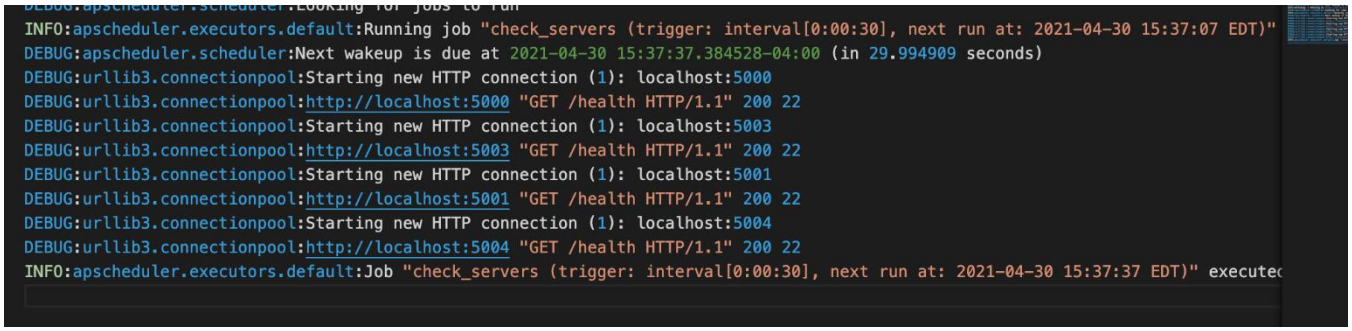
4.4 Start the first order server (5001).

4.5 Check the orders.log to verify that the order server (5001) syncs itself by talking to the other order server (5004)



Sequence of operations for 5:

5.1 Check the frontend.log for /health endpoint hits and the 200 return code



5. Areas of improvement

1) **Consistency:** We could have better synchronization methods to ensure that the data across all replicas is consistent at all times. The current implementation only ensures eventual consistency.

2) **Backup and recovery:** Backups of the databases must be taken regularly to avoid syncing entire database with a non-faulty replica when there is a system failure. The current implementation requires that the entire database is recreated by syncing with a functional replica.

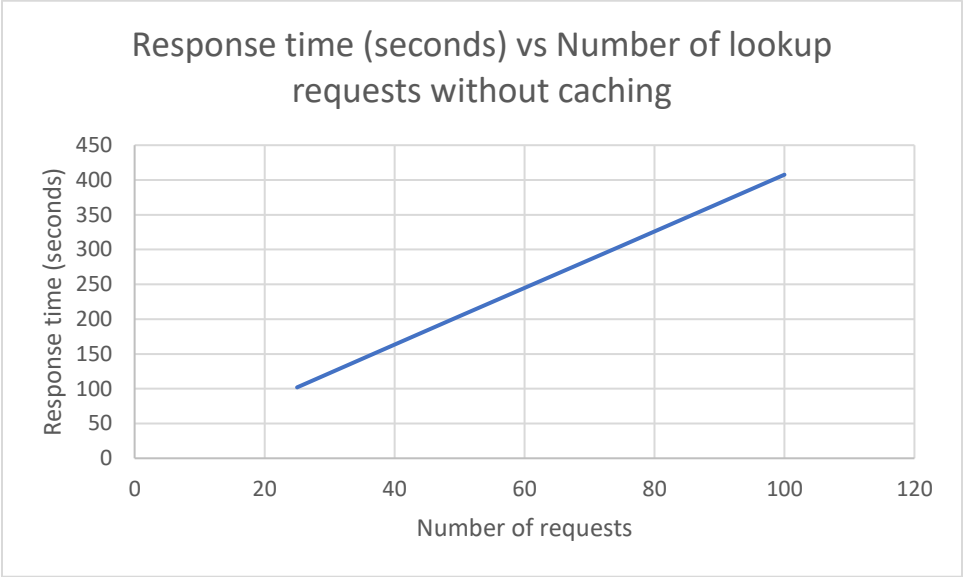
6. Design Trade-offs

1) We use Flask as it is lightweight. But Flask is not well known for handling concurrent requests and so as number of clients would increase, the system will become extremely slow resulting in a bad user experience.

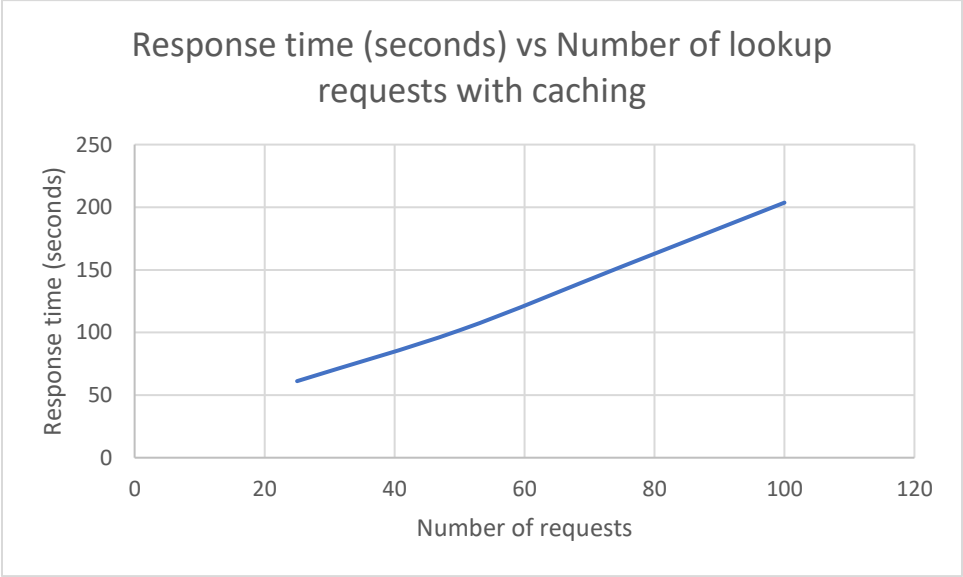
7. Evaluation and Performance

7.1 Lookups – With and without caching.

Below is a plot showing number of lookup requests on the x axis and time taken to get a response (in seconds) on the y axis. The front-end server does not have any caching mechanism in this case. The average response time is about 4 seconds/request.



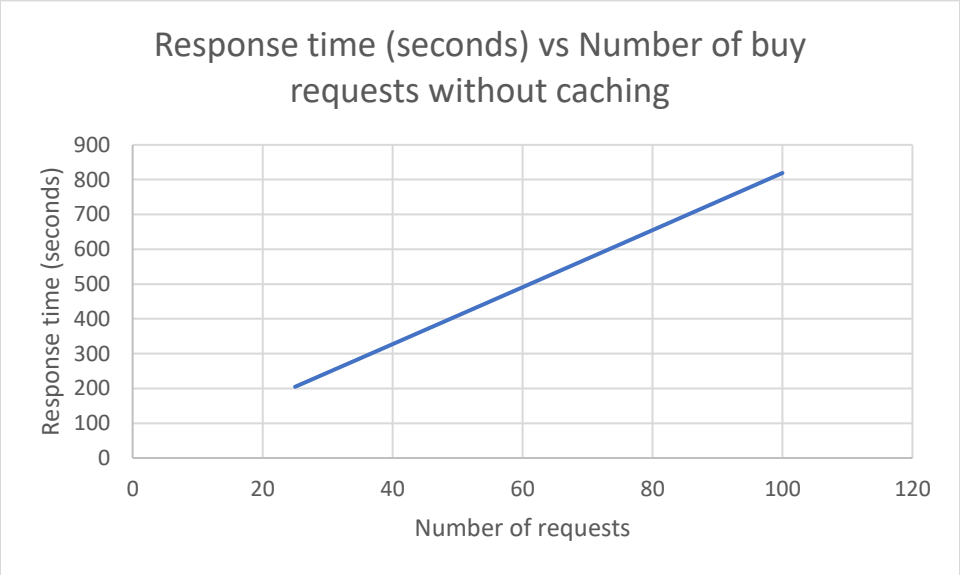
Below is a plot showing number of lookup requests on the x axis and time taken to get a response (in seconds) on the y axis. The front-end server has in-memory caching in this case. The average response time is about 2 seconds/request.



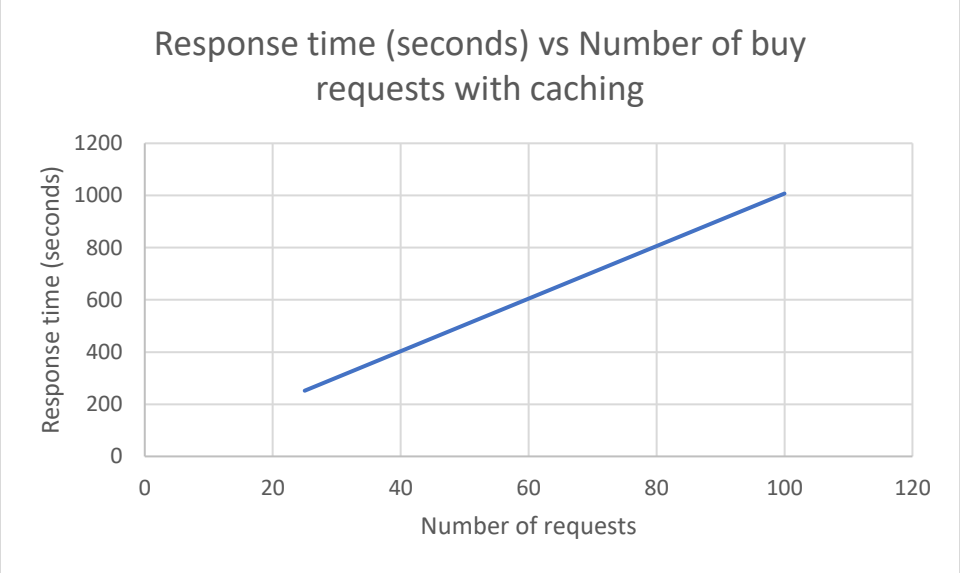
It can be clearly seen from the above two plots that caching significantly reduces response times (nearly reduced by one half).

7.2 Buy – With and without caching

Below is a plot showing number of buy requests on the x axis and time taken to get a response (in seconds) on the y axis. The front-end server does not have any caching mechanism in this case. The average response time is about 8.1 seconds/request.

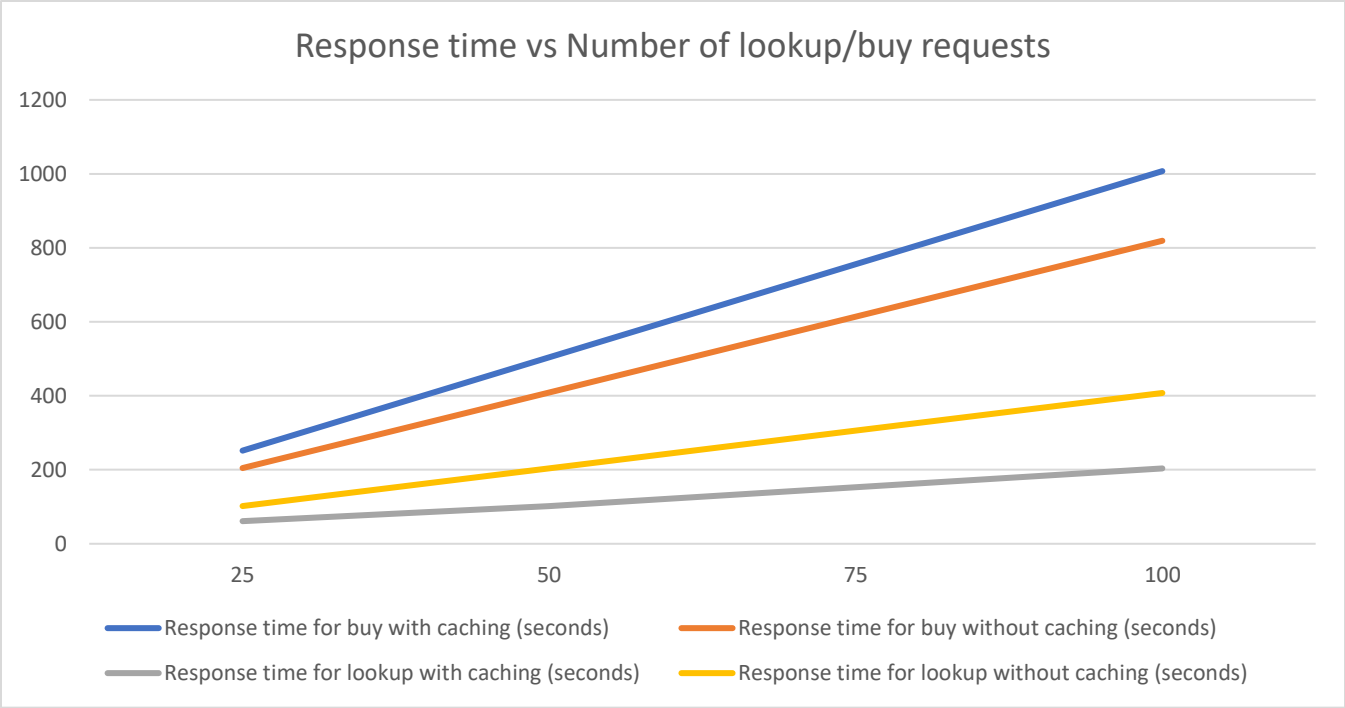


Below is a plot showing number of buy requests on the x axis and time taken to get a response (in seconds) on the y axis. The front-end server has in-memory caching in this case. The average response time is about 10.3 seconds/request.



It can be seen that the cache consistency operations increases the response times slightly (by 2.2 seconds/request on an average) as the buy requests results in an update request on the catalog server which in turn invalidates the cache in the front-end server. However, this latency due to cache operations is not very significant. This maybe since the cache resides in memory and hence the cache invalidate operation does not have any IO which makes is quite fast.

Below is a plot showing trends in both lookup and buy response times with and without caching for comparison.



From the above experiments, we see that the overhead of cache consistency operations is about 1.8 seconds/request.

The latency of a subsequent request if it sees a cache miss is about 4 seconds.