

Marco Serafini

[Homepage](#)

COMPSCI 677 - Spring 20

[Course homepage](#)

Project 2

This project is a group project and it is due on **Apr 6th, 11.59 pm**

Project goals

The goal of this project is to teach you:

- To teach you distributed web programming using web frameworks
 - To teach you programming using REST APIs
 - To teach concepts of multi-tier web design and micro-services.
 - To teach you basics of distributed systems design and experimentation.
-

Project description

You have been tasked to design Pygmy.com - the World's smallest book store. Pygmy.com carries only four books for sale:

1. How to get a good grade in 677 in 20 minutes a day.
2. RPCs for Dummies.
3. Xen and the Art of Surviving Graduate School.
4. Cooking for the Impatient Graduate Student.

Since Pygmy.com hopes to one day become an Amazon, they would like to use sound design principles to design their **online store** in order to allow for future growth.

The store will employ a **two-tier web design - a front-end and a back-end** - and use microservices at each tier. The front-end tier will accept user requests and perform initial processing. The backend consists of two components: **a catalog server and an order server**. The catalog server maintains the catalog (which currently consists of the above four entries). For each entry, it maintains the number of items in stock, cost of the book and the topic of the book. **Currently all books belong to one of two topics: distributed systems (first two books) and graduate school (the last two books).** The order server maintains a list of all orders received for

the books. The front-end is implemented a single microservice, while the catalog and order services in the backend are implemented as two separate microservices. **In this case, a micro-service can be viewed as a separate process that accepts requests**. There will be *three separate components/processes* in total: the front-end server, the catalog server, and the order server.

The front end server supports three operations:

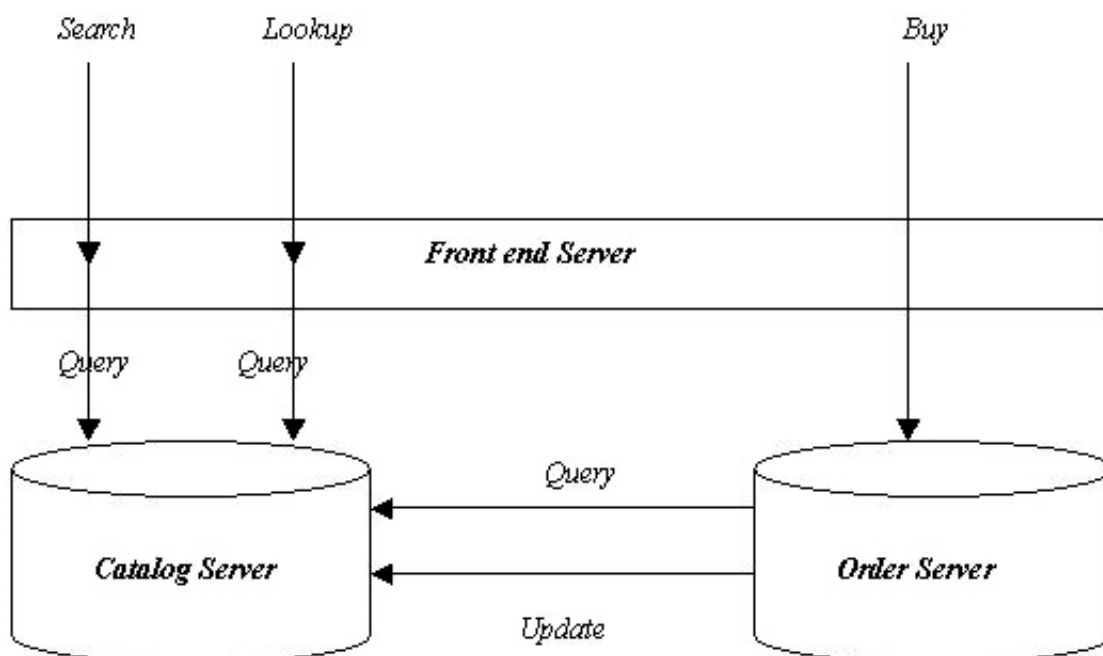
1. *search(topic)* - which allows the user to specify a topic and returns all entries belonging to that category (a title and an item number are displayed for each match).
2. *lookup(item_number)* - which allows an item number to be specified and returns details such as number of items in stock and cost
3. *buy(item_number)* - which specifies an item number for purchase.

The first two operations trigger queries on the catalog server. The buy operations triggers a request to the order server. **The catalog server supports two operations: query and update. Two types of queries are supported: query-by-subject and query-by-item. In the first case, a topic is specified and the server returns all matching entries. In the second case, an item is specified and all relevant details are returned. The update operation allows the cost of an item to be updated or the number of items in stock to be increased or decreased.**

The order server supports a single operation: *buy(item_number)*. Upon receiving a buy request, the order server must first verify that the item is in stock by querying the catalog server and then decrement the number of items in stock by one. The buy request can fail if the item is out of stock.

Assume that new stock arrives periodically and the catalog is updated accordingly.

A pictorial representation of the system is as shown in the figure below.



How to tackle the problem

Unlike lab 1, which used low-level RPC or socket programming, lab 2 will involve programming at a higher level of abstraction. You will use a web framework to write the code for lab 2. We strongly recommend using either Java or Python for this lab and we also strongly recommend use of a lightweight micro web framework such as Flask for Python or **Spark / Ninja for Java**. These are very lightweight web frameworks that are easier to learn than full-fledged web frameworks such as Django / Struts etc. We would strongly discourage you from using these heavyweight frameworks since one of the goals of this lab is to write small/lightweight micro-services. Like before, you still have creative freedom to use a different language or a web framework but you will need to discuss your design choice beforehand with us. A second requirement of the lab is to implement the above interfaces for each component as a HTTP REST interface (the above web frameworks have this functionality built into it). For example, **rather than implementing *search(item)* using an RPC interface, you should use a REST call of the form**

```
SERVER_IP:80/search/itemName
```

or simply

```
SERVER_IP:80/search
```

In the former case, the argument item name is part of the request URL itself, while in the second case, the argument can be passed as a json object as part of the request. In either case, the response should be returned as a JSON object. All interface calls, namely search, lookup, buy, query, update should be exposed as HTTP REST calls by each component.

REST Interface

As noted above, your system should use a REST client/server architecture. This means creating several endpoints that correspond to the interfaces provided above.

A sample endpoint for the search(item) interface may be of the following form:

```
serverIP:80/search/itemName
```

Clients can make this api call to the server and will receive back a json object of the following form:

```
{
  "items": {
    "RPC for Dummies": 345,
    "Cooking for the Impatient Graduate Student": 359
  }
}
```

Observe that you code is no longer using RPCs to communicate and is instead using HTTP requests and responses for communications.

Like before, your code needs to support concurrent requests. However **modern web frameworks have built-in support for accepting multiple concurrent requests and use threads or async processing, and thus you get**

concurrency for “free” without writing low level threads code.

The order and catalog requests need to maintain data in a persistent manner (i.e., on disk). Normally a web framework will maintain persistent data in a database. Your code should minimally use a simple text file (e.g., CSV file) to maintain the catalog and order log, but it is also fine to use a very simple database such as sqlite. Please refrain from using heavyweight databases such as mysql or postgresql / mongodb etc for this lab. Simple text files or sqlite database should suffice.

Your multi-tier application should have the ability to run the three components on different machines in a distributed fashion. You can achieve this by implementing each component (e.g., front-tier, order server, catalog server) as a separate flash micro-service that communicate with each other using HTTP REST calls.

Like in Lab 1, you will use github for a source code control repository. Please make sure you use multiple commits and provide detailed commit messages. You will be evaluated on your use of effective commits. Like before, one of the goals of the lab is to teach you properly test distributed code. For the purposes of the lab, you should write at least two tests of your choice either using a testing framework or using your own scripts/inputs to test the code. The tests and test output should be submitted along with your code. Please be sure to test your code in a distributed (multi-machine) setting.

We do not expect elaborate use of github or testing frameworks - rather we want you to become familiar with these tools and start using them for distributed programming (or your own work.)

No GUIs are required. Simple command line interfaces are fine.

Evaluation and Measurements

Unlike the P2P architecture of lab 1, this lab uses a client-server model, and the server itself uses a multi-tier and microservices design.

1. Deploy your system on three machines, with each of the three component on a different edlab machine. BE SURE NOT TO USE PORT 80 for your code since it may conflict with processes run by other (also port 80 is reserved and typically not allowed for user processes). Run a client on a separate 4th machine and show that your code works properly by making different types of requests and printing appropriate log messages at the client and the components.
2. Compute the average response time per client search request by measuring the end-to-end response time seen by a client for , say, 1000 sequential requests. Also, measure the response times when multiple clients are concurrently making requests to the system, for instance, you can vary the number of clients and observe how the average response time changes.
3. Following the idea shown above, break down the end-to-end response time into component-specific response times by computing the per-tier response time for query and buy requests

Make necessary plots to support your conclusions.

What You Will Submit

When you have finished implementing the complete assignment as described above, you will submit your solution to github. We expect you would have used github throughout for source code development for this lab; please use github to turn in all of the following (in addition to your code)

1. Source code with inline comments/documentation.
2. A copy of the output generated by running your program. **When it receives a book, have your program print a message “bought book book_name “. When a client issues a query (lookup/search), having your program print the returned results in a nicely formatted manner.**
3. A separate document of approximately two to three pages describing the **overall program design**, a description of “how it works”, and design tradeoffs considered and made. Also describe possible improvements and extensions to your program (and **sketch how they might be made**). You also need to describe clearly **how we can run** your program - if we can’t run it, we can’t verify that it works. **Please submit the design document and the output in the docs directory in your repository.**
4. A **separate description of the tests** you ran on your program to convince yourself that it is indeed correct. Also describe any cases for which your program is known not to work correctly. Please submit this document along with the above design doc in the docs directory in your repo. **The tests themselves should be checked into a tests sub-directory.**
5. Include **one single script** to automatically compiles your code, deploys it on **multiple EdLab machines**, and **runs all your tests and experiments**.
6. **Performance results** of your measurements/experiments should be included in the docs directory. Provide the results/data as simple graphs or tables with brief explanation.

Grading policy for all programming assignments

- Program Listing
 - works correctly *by running a single script*: 40%
 - in-line documentation: 10%
- Design Document
 - quality of your system design and creativity: 15%
 - quality of your design document: 10%
- Performance experiments: 10%
- Development style
 - Use of github with checkin comments: 5%
 - Thoroughness of test cases: 10%

Published with [GitHub Pages](#)