COMPSCI 677 Spring 2022

Lab 2: Tiered Microservices-Based Toy Store

Team Members: Maoqin Zhu, Yixiang Zhang

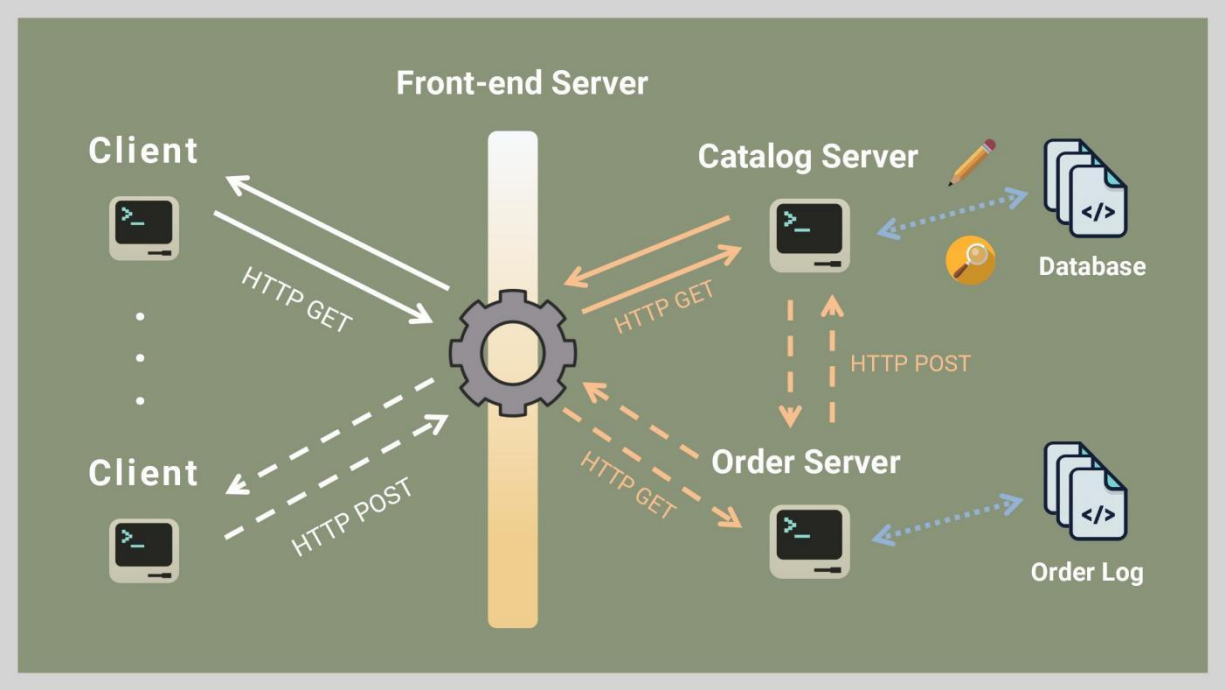
Part 1 - Implement Your Multi-Tiered Toy Store as Microservices

1. Architecture

In this section, we will introduce the architecture of our microservices-based application.

1.1 Overview

In this lab assignment, we aim to design distributed server applications using a multi-tier architecture and microservices.



As you can see, there are three servers in our application. A service-oriented architecture exposes components as services. Each component provides a service. Micro-services are one modern implementation.

1.2 Components

Now we summarize the responsibilities of each components.

client.py

Mode 1: Query and Buy randomly

It randomly queries an item, if the returned quantity is greater than 0, with probability "p"(environment variable initialized in terminal) it will send an order request.

Mode 2: Initiate a serials of Query

You can specify the toy name and query times as you want.

Mode 3: Initiate a serials of Buy

You can specify the toy name, quantity and number of requests as you want.

front_end.py

Note that when implementing the front-end service, we did NOT use existing web frameworks such as Django, Flask, Spark, etc. Rather we implement our own functionalities.

Function 1: Interact with clients

The clients should be able to communicate with us using HTTP-based REST APIs. .

Function 2: Dispatch clients requests properly

Front-end does NOT implement the "Query" or "Buy". Rather it will forward the request to the other servers depending on request type (GET or POST).

Function 3: Filter invalid bad requests

Considering there are only few types of toys, we simulate an invalid requests filter in this assignment, which will validate the "toyName" parameter.

catalog_server.py

Note that we implemented the catalog server as an in memory but persistent on disk database.
In other words, in our application, it is sort of like the Redis.

Function 1: Maintain data in memory & Persist data on disk

It can be used as a cache to store data in memory for a speed up. In this case, all the "Query" can be faster without I/O operations. In terms of the persistent strategy, we write the data into disk every time we modify the data in memory.

Function 2: Process the "Buy" request from order server

The order server nteracts with catalog server to complete the order. Specifically, a buy order should succeed only if the item is in stock, and the stock should be decremented in this case. These operations should be done in the catalog server.

Concurrency & Synchronization

Consider that our application is multi-threaded which supports concurrency, so whatever you call **read()** or **write()**, you should try to acquire the **lock** at first.

order_server.py

Note that the order server will process all the "Buy" (POST) requests from the front-end server. It should communicate with catalog server during the process.

Function 1: Communicate with catalog server

First, it should forward the request from front end to the catalog server, so the parameter delivery should be implemented. Second, according to the response from catalog server, it should send a reply to the front end (successful or not).

Function 2: Generate correct order ID

We wrote a *Counter class* to help generate order ID. Consider that our application is multi-threaded, the increment at counter should be conducted with **lock** to guarantee the ID is unique, incremental and starting from 0.

Function 3: Write order log

In our design, after processing a request, order server will maintain the order log (including order number, product name, and quantity) in a persistent manner.

Note: if a "Buy" request is unsuccessful, we write the order ID as -1 in our log.

At the end, here we described the dependency relationships between services as follows. An arrow points from A to B indicates B depends on A. This is important when we start each server in order in this part.

$$catalog\ server \leftarrow order\ server \leftarrow frontend\ server$$

The details of how to pass IP address as environment variable when starting servers can be found in Section 3.2.

2. Communication

In this section, we provide an explanation about how our application interacts with clients and how servers communicate with each other.

2.1 HTTP Methods

In this lab assignment, we mainly employed the HTTP GET and HTTP POST. For each HTTP communication(between client and server / between micro services), we write an automatic unit testing process in **“test_func.py”**.

HTTP GET

When to call? How to call? And what we receive?

Scenario1: Client → Front-end server (send “Query” requests)

Interface: http://<IP: Port>/products/<product_name>

Example URI: HTTP GET http://127.0.0.1: 6060/products/Tux

Scenario2: Front-end server → Catalog server

Interface: http://<IP: Port>/products/<product_name>

Example URI: HTTP GET http://127.0.0.1: 10086/products/Tux

JSON Response(payload) for scenario 1 and 2:

Successful:

```
{
    "data": {
        "name": "Tux",
        "price": 25.99,
        "quantity": 100
    }
}
```

Unsuccessful:

```
{
    "error": {
        "code": 404,
        "message": "product not found"
    }
}
```

Scenario3: Front-end server → Order server

Interface: http://<IP: Port>/products/<product_name>/<quantity>

Example URI: HTTP GET http://127.0.0.1: 10010/products/Tux/100

JSON Response(payload):

Successful:

```
{
    "data": {
        "order_number": "1"
    }
}
```

Unsuccessful 1:

```
{
    "order_number": "-1",
    "message": "out of stock"
}
```

Unsuccessful 2:

```
{
    "order_number": "-1",
    "message": "product not found"
}
```

HTTP POST

When to call? How to call? And what we receive?

Scenario1: Client → Front-end server (send “Buy” requests)

Interface**:** http://<IP: Port>/<product_name>/<quantity>

Example URI: HTTP POST http://127.0.0.1: 6060/Tux/100

JSON Response(payload):

Successful:

```
{
    "data": {
        "order_number": "1"
    }
}
```

Unsuccessful 1:

```
{
    "error": {
        "code": 404,
        "message": "out of stock"
    }
}
```

Unsuccessful 2:

```
{
    "error": {
        "code": 404,
        "message": "product not found"
    }
}
```

Scenario2: Order server → Catalog server

Interface: http://<IP: Port>/<product_name>/<quantity>

Example URI: HTTP POST http://127.0.0.1: 10086/Tux/100

JSON Response(payload):

Successful:

```
{ "message": "order has been placed" }
```

Unsuccessful 1:

```
{ "message": "out of stock" }
```

Unsuccessful 2:

```
{ "message": "product not found" }
```

2.2 Python Modules

Here we mention three important modules in our source code.

import socket

We actually apply a regular socket connection in this project.

- s.connect(): actively initiates TCP server connection

- s.bind(): binds address (hostname, port) to socket

- s.listen(): sets up and start TCP listener

import requests

Python requests module which allows us to send HTTP requests using Python. And this HTTP request returns a response in a specific format, including status line, headers and body.

- s.get(url, params, args): sends a GET request to the specified url

- s.post(url, data, json, args): sends a POST request to the specified url

import json

Python has a built-in package called JSON, which can work with JSON data.

- json.dumps(obj): convert Python object into a JSON string

3. Configurations

In this section, we mention several parameters you should know.

3.1 Static Parameters List

1) Initial products info in "database.txt"
- *(toyName, price, quantity)*

2) Socket ports in "front_end.py", "catalog_server.py", "order_server.py"
- *port number*

3) Socket port in "client.py"

- *port number*

4) Client Terminal

- *Type in: which mode, toy name, quantity, number of requests*

(check out Section1.2 for details of each mode)

3.2 Environment Variables

As discussed in Section 2.1, we are passing the server IP as an environment variable because in Part 2 the server IP address might change. And also, since there exist dependency relationships between servers, we should do it in specific order in this part. Here is an example of how we type the Linux command.

Example Command at Server Terminal

Step1: $ python3 catalog_server.py

Step2: $ CATALOG=128.119.243.175 python3 order_server.py

Step3: $ CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 front_end.py

Example Command at Client Terminal

FRONT=128.119.243.175 p=0.5 python3 client.py

Note: Be careful when you do local test. When starting a server, we will print IP address on your screen. Type that IP address as environment variable rather than “127.0.0.1”.

4. Reference

[1] REST API Tutorial

(https://restfulapi.net/http-methods/#get)

[2] W3Schools Python JSON

(https://www.w3schools.com/python/python_json.asp)

[3] W3Schools Module Reference

(https://www.w3schools.com/python/module_requests.asp)

[4] Mozilla HTTP Messages

(https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages)

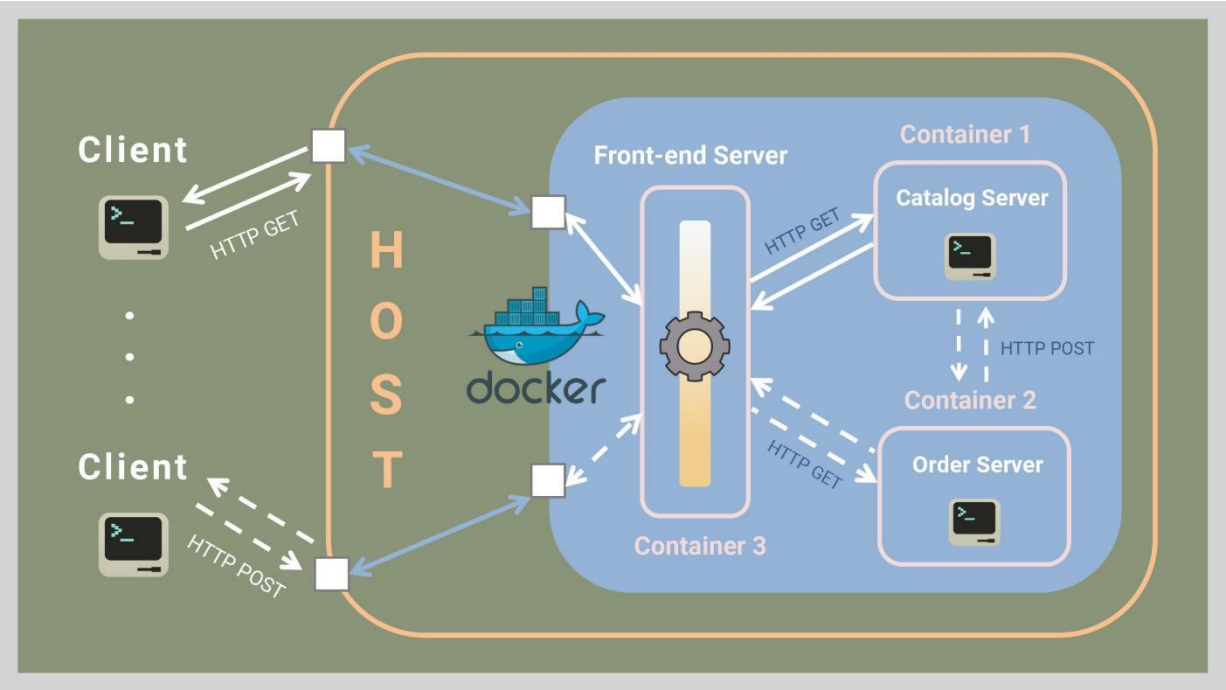
Part 2 - Containerize Your Application

1. Architecture

In this section, we will learn how to use Docker to containerize our micro-service, and learn to manage an application consisting of multiple containers using Docker Compose.

1.1 Overview

The architecture is similar to that in part 1, but what's different is that we containerize our application code and then learn to deploy all components as a distributed application using Docker.



Rather than directly connect to the font-end server, clients actually visit IP address & port number of the host machine. The blue-colored area indicates a **Docker environment**, where containers communicates with each other using a special virtual IP and port (different from the host IP exposed to clients).

If we wish to communicate with a Docker container from our host machine, then we need to have a port mapping. Port mapping is used to access the services running inside a Docker container. We open a host port to give us access to a corresponding open port inside the Docker container. Then, all the requests that are made to the host port can be redirected into the Docker container.

In the next section, we will introduce how we use Docker and how to configure them with a few examples.

2. Docker

Docker provides the ability to package and run an application in a loosely isolated environment, that is a container. The isolation and security allows us to run many containers simultaneously on a given host.

2.1 DockerFile

A Dockerfile is a text file that defines a Docker image. We will use a Dockerfile to create our own custom Docker image, in other words to define our custom environment to be used in a Docker container.

We created 3 DockerFiles for each components in our application.

DockerFile List:

1) front_end_dockerfile

2) catalog_dockerfile

3) order_dockerfile

Instruction List:

- FROM python:3.8-alpine: use the official python image as the base layer.

- RUN pip install <library>: runs the command to install the dependencies.

Example: RUN pip install requests

- WORKDIR </path>: use /path as our working directory inside the docker image.

Example: WORKDIR /src

- CMD ["a", "b", "c"]: specifies the instruction that is to be executed

Example: CMD ["python", "-u", "catalog_server.py"]

2.2 Docker Compose

Docker Compose is a Docker tool which used to define and run multi-container applications. With Compose, we use a YAML file to configure our application's services. Then, with a single command, we can create and start all the services from our configuration.

We created a docker compose file for this part as follows:

docker-compose.yml

```yaml
version: '3.9'
services:
  catalog:
    build: .
    image: catalog
    volumes:
      - ../src/catalog:/src

  order:
    build: .
    image: order
    volumes:
      - ../src/order:/src
    depends_on:
      - catalog

  front_end:
    build: .
    image: front_end
    ports:
      - 6060:6060
    depends_on:
      - catalog
      - order
    volumes:
      - ../src/front_end:/src
```

As shown above, we have three services. Notice that we have described the dependency relationships between services in our yml file. We use the arrows to summarize the relationship as follows. An arrow points from A to B indicates B depends on A. This is important when we start each server in order in this part:

$$catalog\ server \leftarrow order\ server \leftarrow frontend\ server$$

Note that files we write in a Docker container are not directly accessible from the host, and they will be erased when the container is removed. So using **"volumes"** line shown above, we mount a directory on the host machine as a volume to the catalog and order services, so that files and output can be persisted after the containers are removed.

3. Configurations

In this section, we mention several parameters you should know.

3.1 Static Parameters List

1) Initial products info in “database.txt”

- (toyName, price, quantity)

2) Socket ports in “front_end.py”, “catalog_server.py”, “order_server.py”

- port number

3) Socket port in “client.py”

- port number

4) Client Terminal

- Type in: which mode, toy name, quantity, number of requests

(check out Section1.2 for details of each mode)

3.2 Docker Images

Before running the Docker containers, we should build the Docker images, and if we want to stop and remove those images we built, we can do it as follows.

Build Docker Image

$ sudo docker build -f <DockerFile> . -t <Name>

Example: $ sudo docker build -f catalog_dockerfile . -t catalog

The dot “.” means using the current folder as the working directory for building the image. “-t” means the built image should be tagged with a name.

Stop & Remove Image

$ sudo docker stop <Name> and **$ sudo docker rm <Name>**

Example: $ sudo docker stop catalog / $ sudo docker rm catalog

The “docker stop“ command will stop a running container and free the cpu/memory used by that container. Stopped containers remain on our disk: intermediate results produced will still be stored. To completely remove a container, use the “docker rm” command

3.3 Environment Variables

On the one hand, if we run dockers manually, we are passing the server IP, port mapping as environment variables. Since there exist dependency relationships between servers, we should do it in specific order.

Example Command at Server Terminal

Step1: $ sudo docker run --name catalog catalog

Step2: $ sudo docker run --name order --env CATALOG=172.17.0.2 order

Step3: $ sudo docker run --name front_end --env CATALOG=172.17.0.2 --env ORDER=172.17.0.3 -p 6060:6060 front_end

Example Command at Client Terminal

$ FRONT=172.17.0.4 p=0.8 python3 client.py

Note: Be careful when you do local test. When starting a server, we will print IP address on your screen. Type that IP address as environment variable rather than “127.0.0.1”.

On the other hand, as described in Section 2.2, we can actually run dockers using Docker compose, which is an easier way to orchestrate multiple containers.

Server Terminal

$ sudo docker-compose up

Client Terminal

$ FRONT=172.17.0.4 p=0.5 python3 client.py

4. Reference

[1] Docker Docs

(https://docs.docker.com/)

[2] Docker Tutorial: Get Going From Scratch

(https://stackify.com/docker-tutorial/)

[3] Docker for Beginners

(https://docker-curriculum.com/#introduction)

[4] Assigning a Port Mapping to a Running Docker Container

(https://www.baeldung.com/linux/assign-port-docker-container)

Part 3 - Testing and Performance Evaluation

1. Automated Testing

In this part, we are testing the functionality and performance of our applications and micro-services.

1.1 Python Unittest Module

The unittest unit framework supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

1.2 Functional Test

Our application supports "Query" and "Buy" calls from clients, and also each components will communicate with each other during processing. For different HTTP GET / HTTP POST, we created 13 test cases which correspond to 13 possible HTTP responses described in Part 1 Section 2.1.

In order to conduct an automatic unit test, we created a Python testing file called **"test_func.py"**, where our test cases go over all the types of responses by sending a series of valid/invalid requests. And also we will check if it does reply as expected using **assertEqual()** method.

In addition, we provide a shell **"test_func.sh"** for you to run multiple testing cases more conveniently. In terms of testing results we get, please check out the "Output" document for details.

Execute Shell File to Run Multiple Test Cases

$ sh test_func.sh or **$ bash test_func.sh**

In terms of each command in the shell, it follows:

$ FRONT=128.119.243.175 CATALOG=128.119.243.175 ORDER=128.119.243.175 python3 -m unittest -v <Module>.<Class>.<TestCase>

Specifically, module name is "test_func", class name is "TestFunctionality", and test case is the method name defined in "test_func.py" such as "test_app_client_query_valid".

Note: Be careful when you do local test. When starting a server, we will print IP address on your screen. Type that IP address as environment variable rather than "127.0.0.1".

Notice that our test cases are effective only when database is in initial state, because expected response is configured statically in testing codes. Of course, you can also run your own test case simply by configuring request parameters and expected responses in the method. The initial state of database should be:

```
Tux 25.99 80
Whale 34.99 94
Elephant 29.99 87
Bird 39.99 98
```

1.3 Load Test

Our application should be able to deal with concurrent requests. In order to conduct an automatic performance test, we created a Python testing file called **"test_load.py"**. We are opening multiple client terminals, and each of them will send a series of "Query" or "Buy" requests simultaneously using our testing code. After doing so, we will evaluate the latency in "evaluation" document.

Multiple Terminals

$ FRONT=128.119.243.175 python3 -m unittest -v <Module>.<Class>.<TestCase>

Specifically, module name is "test_load", class name is "TestLoadPerformance", and test case is the method name defined in "test_load.py" such as "test_load_query".

Note: Be careful when you do local test. When starting a server, we will print IP address on your screen. Type that IP address as environment variable rather than "127.0.0.1".

2. Configurations

2.1 Environment Variables

1) IP address in "test_func.sh"

- IP address of FRONT, CATALOG and ORDER

2) Unittest command at terminal

- IP address, module name, class name, method name

3. Reference

[1] Python Docs - Unittest

(https://docs.python.org/3/library/unittest.html)

[2] How To Use unittest to Write a Test Case for a Function in Python

(https://www.digitalocean.com/community/tutorials/how-to-use-unittest-to-write-a-test-case-for-a-function-in-python)