Distrubted Operating Systems

Lab #2 - Replication and caching

Bazar.com: A Multi-tier Online Book Store

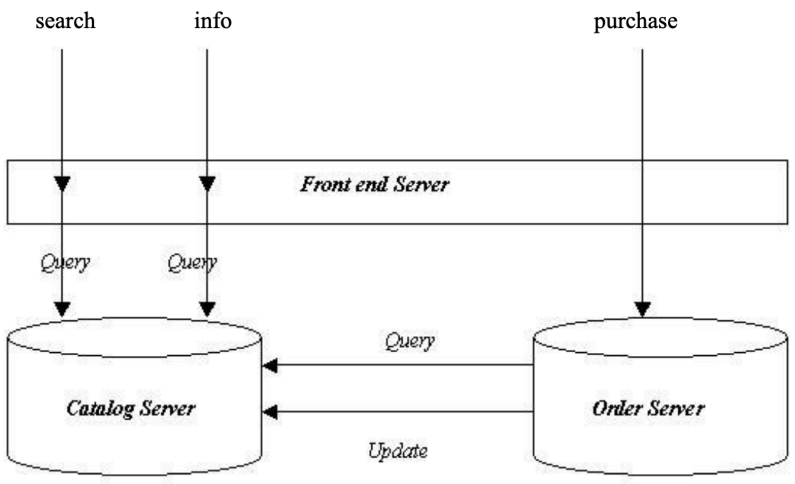
Yousef Hanbali
11923575

GitHub Repository:
`https://github.com/yousefhanbali/dos-lab1-v2/`

January 9, 2024

1 Introduction

This multi-tier online book store are based on 3 different services: a service for frontend, a service for purchase(order) and a service for the catalog in which we store, get and update the data of the books from.
We will use Lumen to implement the catalog service, Spark to implement Frontend and order service, and each service will be deployed on a seperate Docker instance based on Linux. It will be done with this architecture:



2 Catalog Service

2.1 Technology

In this service we used Lumen, which is based on PHP and a mini framework from Laravel.

2.2 Interfaces

This service exposes 3 interfaces

2.2.1 Get Book By Id

In this service we could perform a GET request on the server and get a certain book by it's Id.
I utilize the URI: `/book/1` in order to get the book with ID = 1 for example.

2.2.2 Get Books By Subject

In this service we could also perform a GET request on the server, to get books that have a certain subject.
We can utilize the URI: `/search/distrubtion` in order to get the book that is has distrubtion as it's subject.

2.2.3 Update the book quantity/price

In this service we could perform a PUT request in order to update the quantity and price of a certain product
So for example, in order to update book that has Id = 1, with quantity of 30 and price of 10, we send to the URI: `/book/1` a put request that has body

```
{
    "quantity":30,
    "price":10
}
```

And then it notifies all other replicas, as we will explain later how.

2.2.4 Purchase book quantity/price

In this service we could perform a PUT request in order to purchase of a certain product
So for example, in order to purchase book that has Id = 1, we send to the URI: `/purchase/1`

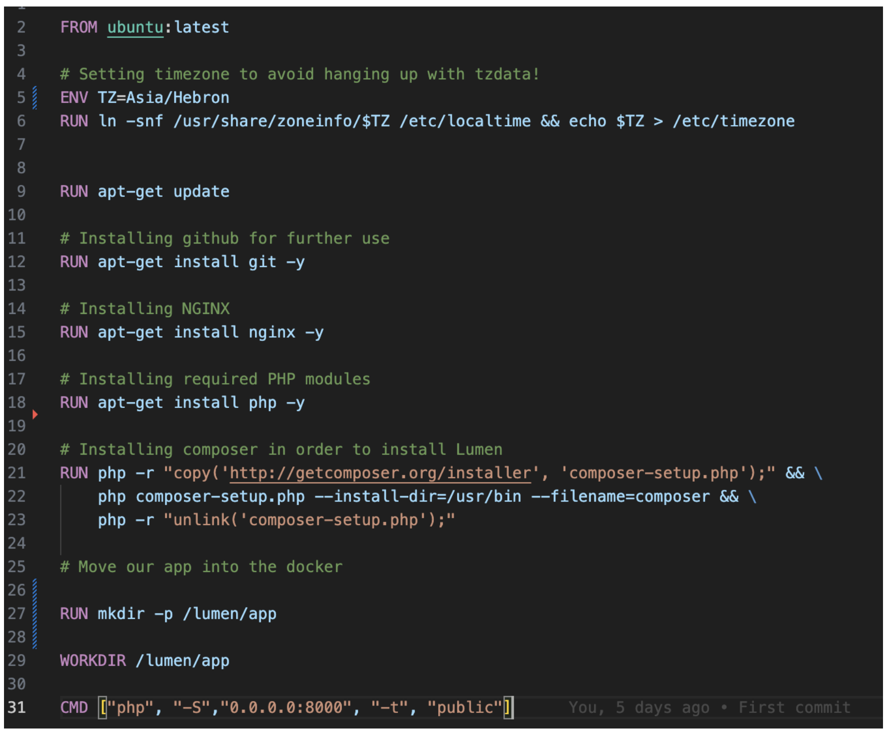
a put request, and then it notifies other replicas.

2.2.5 Notify node

This interface was particularly added in order to implement strong consistency between replicas of the catalog service.
When a node receives a PUT request, it updates its content, and then notifies all other replicas in order to hold consistency between data.
So it sends a POST request to `/notify/1` to notify on change for book 1 with the new data in the body of the request.

2.3 Dockerfile

The docker makefile for this image will be based on Ubuntu as we mentioned earlier.



```
FROM ubuntu:latest

# Setting timezone to avoid hanging up with tzdata!
ENV TZ=Asia/Hebron
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone


RUN apt-get update

# Installing github for further use
RUN apt-get install git -y

# Installing NGINX
RUN apt-get install nginx -y

# Installing required PHP modules
RUN apt-get install php -y

# Installing composer in order to install Lumen
RUN php -r "copy('http://getcomposer.org/installer', 'composer-setup.php');" && \
    php composer-setup.php --install-dir=/usr/bin --filename=composer && \
    php -r "unlink('composer-setup.php');"

# Move our app into the docker

RUN mkdir -p /lumen/app

WORKDIR /lumen/app

CMD ["php", "-S","0.0.0.0:8000", "-t", "public"]        You, 5 days ago • First commit
```

First, we set up the timezone by setting a enviroment variable TZ=Asia/Hebron, because when we install PHP, it requires timezone, in which if we don't add it, it will stall up the image build and won't continue.
Then we do a apt-get to get latest updates of packages, then we install php and nginx which is required by PHP also. And for the latest, we install composer, in which we can use to update the files of the framework later on if we wanted to. At last we make a directory for our app which we will mount it in the docker compose file later, and then the run command in CMD, which we used this exact command for because this only runs when you run the image.

3 Purchase(Order) Service

3.1 Technology

In this service, I use Spark miniframework, which is based on JAVA.

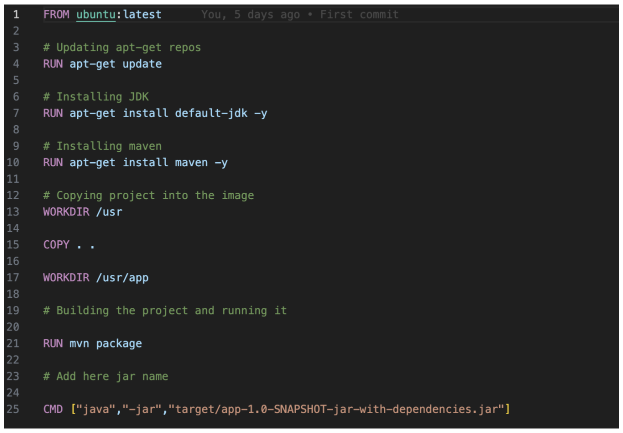
3.2 Interfaces

This service exposes 1 interface only.

3.2.1 Purchase a book

We could perform a PUT request on the interface, which tell you if the product is out of stock, or the purchase operation was successful.
The URI for the interface is `/purchase/1` for a purchase request for book with id = 1.

3.2.2 Dockerfile



```
FROM ubuntu:latest      You, 5 days ago • First commit

# Updating apt-get repos
RUN apt-get update

# Installing JDK
RUN apt-get install default-jdk -y

# Installing maven
RUN apt-get install maven -y

# Copying project into the image
WORKDIR /usr

COPY . .

WORKDIR /usr/app

# Building the project and running it

RUN mvn package

# Add here jar name

CMD ["java","-jar","target/app-1.0-SNAPSHOT-jar-with-dependencies.jar"]
```

For this, we only install maven and jdk, and copy files. We didn't mount a volume because the file wouldn't have mounted when the package building command (MVN package) was run during the building, and it's not efficient to build every time we turn on the image.

4 Frontend Service

4.1 Technology

This will use Spark as the purchase service.

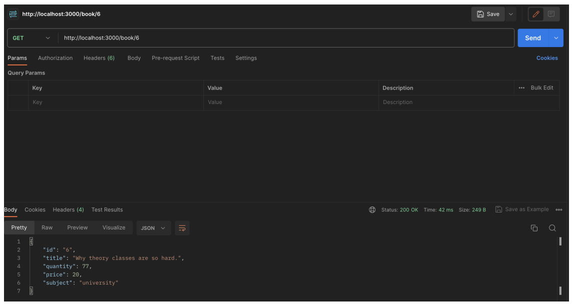
4.2 Interfaces

This service will act as a proxy server (because there is not a real frontend). In order to make this work, we see that GET requests that arrive at the front end are all forwarded to

the category service, and the put requests are forwarded to the purchase service.
So In order to make this work, we make the front end server forward all GET requests to the category service, and the PUT to the purchase service, and when the response comes back it gets forwarded back to the client.

4.2.1 Get a book by id

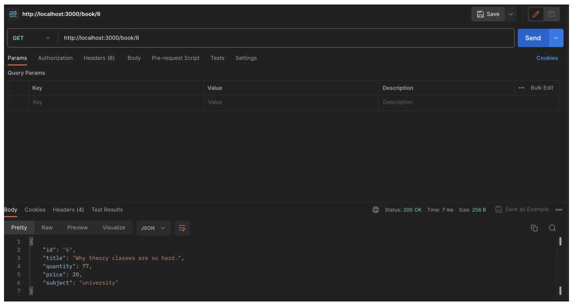
The result:


GET http://localhost:3000/book/6

Status: 200 OK Time: 42 ms Size: 249 B

```
{
    "id": "6",
    "title": "Why theory classes are so hard.",
    "quantity": 77,
    "price": 20,
    "subject": "university"
}
```

(a) Cache miss


GET http://localhost:3000/book/6

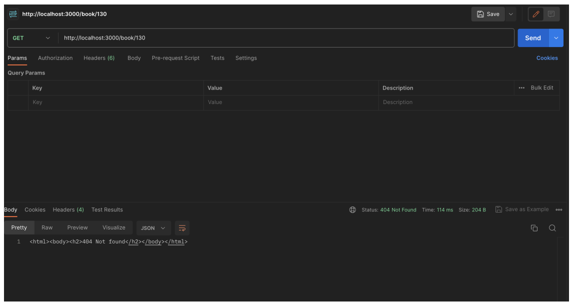
Status: 200 OK Time: 7 ms Size: 256 B

```
{
    "id": "6",
    "title": "Why theory classes are so hard.",
    "quantity": 77,
    "price": 20,
    "subject": "university"
}
```

(b) Cache hit for id=6

Figure 1: Getting Book with Id=6

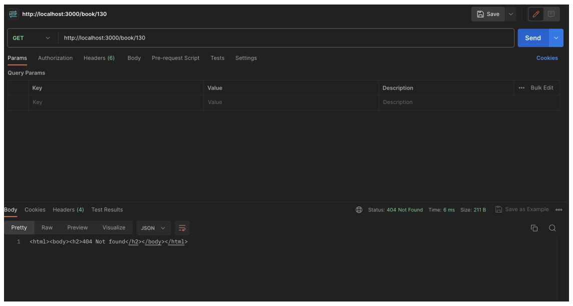
Testing a book that does not exist:


GET http://localhost:3000/book/130

Status: 404 Not Found Time: 114 ms Size: 204 B

```
<html><body><h2>404 Not found</h2></body></html>
```

(a) Cache miss


GET http://localhost:3000/book/130

Status: 404 Not Found Time: 6 ms Size: 211 B

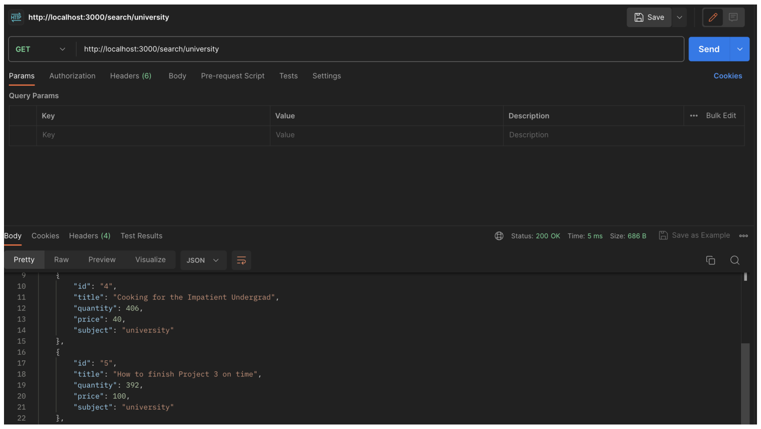
```
<html><body><h2>404 Not found</h2></body></html>
```

(b) Cache hit for id=130 (Does not exist)

Figure 2: Getting Book with Id=6

4.2.2 Get books by subject

The result:



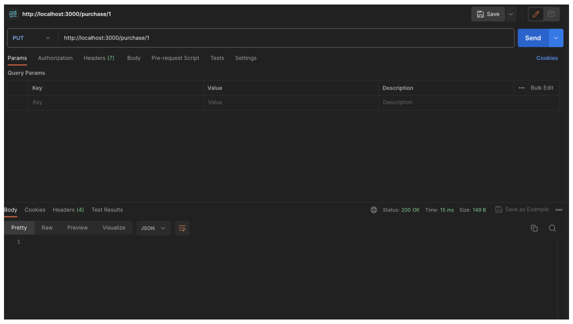
GET http://localhost:3000/search/university

Status: 200 OK Time: 5 ms Size: 686 B

```
    {
        "id": "4",
        "title": "Cooking for the Impatient Undergrad",
        "quantity": 406,
        "price": 40,
        "subject": "university"
    },
    {
        "id": "5",
        "title": "How to finish Project 3 on time",
        "quantity": 392,
        "price": 100,
        "subject": "university"
    },
```

4.2.3 Purchasing a book

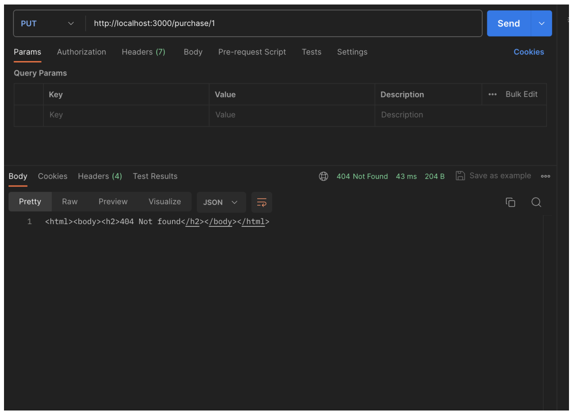
The result:



PUT http://localhost:3000/purchase/1

Status: 200 OK Time: 15 ms Size: 149 B

(a) Purchasing a book



PUT http://localhost:3000/purchase/1

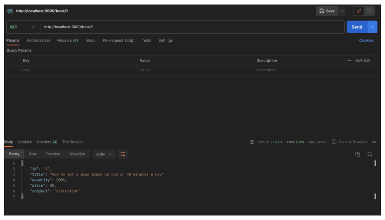
404 Not Found 43 ms 204 B

```
<html><body><h2>404 Not found</h2></body></html>
```

(b) Out of stock books

4.2.4 Invalidating Cache

This interface will be used by catalog service in order to invalidate cache entries, which we will talk about in caching section. Purchasing a book will result in invaldating cache entry.



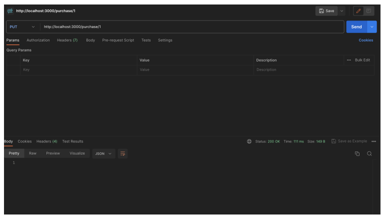
http://localhost:3000/book/1

GET http://localhost:3000/book/1

Status: 200 OK Time: 6 ms Size: 277 B

```
{
    "id": "1",
    "title": "How to get a good grade in DOS in 40 minutes a day",
    "quantity": 2091,
    "price": 30,
    "subject": "distubtion"
}
```

(a) Cached book

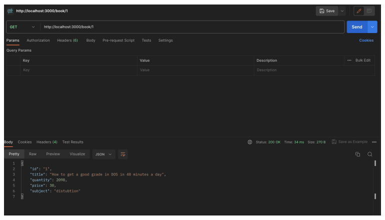


http://localhost:3000/purchase/1

PUT http://localhost:3000/purchase/1

Status: 200 OK Time: 111 ms Size: 149 B

(b) Purchasing a book to cause invalidate entry



http://localhost:3000/book/1

GET http://localhost:3000/book/1

Status: 200 OK Time: 34 ms Size: 270 B

```
{
    "id": "1",
    "title": "How to get a good grade in DOS in 40 minutes a day",
    "quantity": 2090,
    "price": 30,
    "subject": "distubtion"
}
```

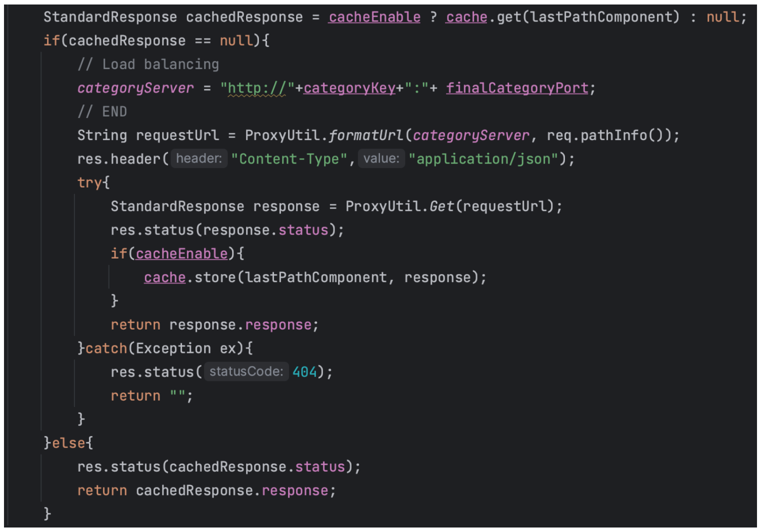
(c) Cache miss for the book

4.3 Dockerfile

Same dockerfile as purchase service.

4.4 Caching

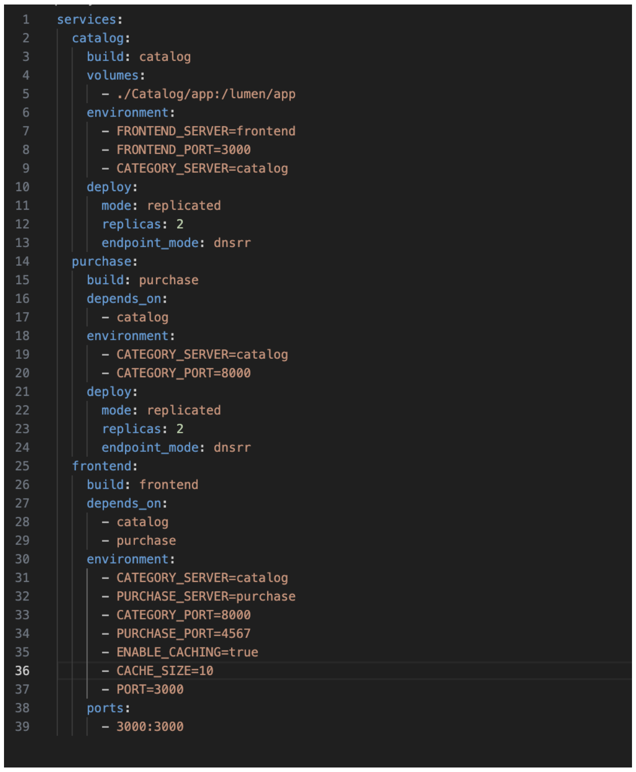
In this node we added a cache, which caches the GET requests that comes into the server. It has a Least Recently Used replacement policy, and has a capacity of 10 by default, which was the optimal size for performance.
We cache the response by the whole url, and when an invalidate request for the cache arrive, it is sent by the catalog service, so then we delete the entry and also delete all search entries from the cache from the cache in order to preserve consistency.
Cache was implemented by a hash map to guarantee an access time of O(1) when searching for a cache hit.



```
StandardResponse cachedResponse = cacheEnable ? cache.get(lastPathComponent) : null;
if(cachedResponse == null){
    // Load balancing
    categoryServer = "http://"+categoryKey+":"+ finalCategoryPort;
    // END
    String requestUrl = ProxyUtil.formatUrl(categoryServer, req.pathInfo());
    res.header( header: "Content-Type", value: "application/json");
    try{
        StandardResponse response = ProxyUtil.Get(requestUrl);
        res.status(response.status);
        if(cacheEnable){
            cache.store(lastPathComponent, response);
        }
        return response.response;
    }catch(Exception ex){
        res.status( statusCode: 404);
        return "";
    }
}else{
    res.status(cachedResponse.status);
    return cachedResponse.response;
}
```

5 Docker compose

In this part, we will build the compose.yaml file, of which docker compose will use to build up all images together.



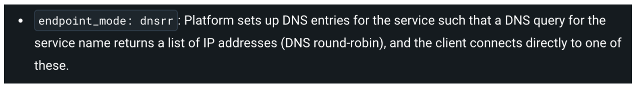
```yaml
services:
  catalog:
    build: catalog
    volumes:
      - ./Catalog/app:/lumen/app
    environment:
      - FRONTEND_SERVER=frontend
      - FRONTEND_PORT=3000
      - CATEGORY_SERVER=catalog
    deploy:
      mode: replicated
      replicas: 2
      endpoint_mode: dnsrr
  purchase:
    build: purchase
    depends_on:
      - catalog
    environment:
      - CATEGORY_SERVER=catalog
      - CATEGORY_PORT=8000
    deploy:
      mode: replicated
      replicas: 2
      endpoint_mode: dnsrr
  frontend:
    build: frontend
    depends_on:
      - catalog
      - purchase
    environment:
      - CATEGORY_SERVER=catalog
      - PURCHASE_SERVER=purchase
      - CATEGORY_PORT=8000
      - PURCHASE_PORT=4567
      - ENABLE_CACHING=true
      - CACHE_SIZE=10
      - PORT=3000
    ports:
      - 3000:3000
```

For the catalog service, we need to make 2 replicas for this service. So can use replicas in the deploy section of the catalog service which will result in 2 replicas. And in order to use them, we use DNSRR mode, which is a DNS with a round robin, which will do the load balancing on its own and allow us to change less code in each service, and which will allow the replicas to see each other, which will be used to provide strong consistency. And the mode setting in deploy has two options: global and replicated, in which we will use replicated so that it can assign IPs to each container.
For the purchase service, because it depends on the category, we need to add catalog to depends_on section, so that it starts up after it. And we add the catalog server to the environment variable, so that if we want to change port of the category service, we can without changing our code. And also it needs replication so we set it up same as the catalog server.
For the frontend service, it depends on both catalog, and purchase. And we add both the

category and purchase server to the environment variables, as well as the port. And we add some other environment variables for settings of caching.

5.1 Load Balancing

We will be using DNSRR mode, which will be mentioned in the Compose file section, of which according to docker documentation here gives us a DNS with a round robin, which gives us the load balancing we want.



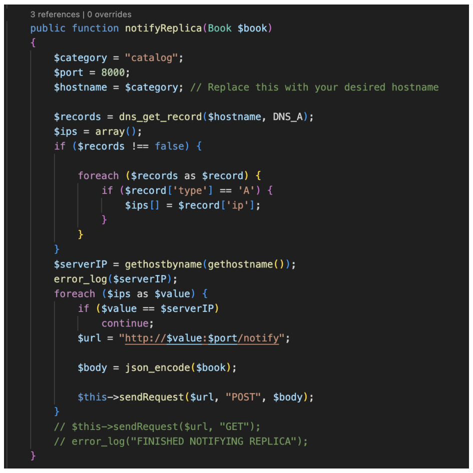
- `endpoint_mode: dnsrr`: Platform sets up DNS entries for the service such that a DNS query for the service name returns a list of IP addresses (DNS round-robin), and the client connects directly to one of these.

Figure 5: Docker documentation

5.2 Replication

In order for replication to work, as we mentioned earlier, we exposed an interface to notify other replicas of change in catalog service. But replicas don't directly see each other. So in order to exactly send requests to other replicas, we need to know the IP the docker assigned to the container, and we can do this by resolving IPs behind the DNS named catalog (name of the service) which will give us IPs of all alive replicas, then when an PUT request arrives, we can send requests to replcias to notify them on this change.

And for accessing every replica from the frontend node, we can access them normally, by using the service name with the port, and the DNS round robin will switch up between the nodes.



```
3 references | 0 overrides
public function notifyReplica(Book $book)
{
    $category = "catalog";
    $port = 8000;
    $hostname = $category; // Replace this with your desired hostname

    $records = dns_get_record($hostname, DNS_A);
    $ips = array();
    if ($records !== false) {

        foreach ($records as $record) {
            if ($record['type'] == 'A') {
                $ips[] = $record['ip'];
            }
        }
    }
    $serverIP = gethostbyname(gethostname());
    error_log($serverIP);
    foreach ($ips as $value) {
        if ($value == $serverIP)
            continue;
        $url = "http://$value:$port/notify";

        $body = json_encode($book);

        $this->sendRequest($url, "POST", $body);
    }
    // $this->sendRequest($url, "GET");
    // error_log("FINISHED NOTIFYING REPLICA");
}
```

Figure 6: Catalog code for resolving DNS

6 How to run

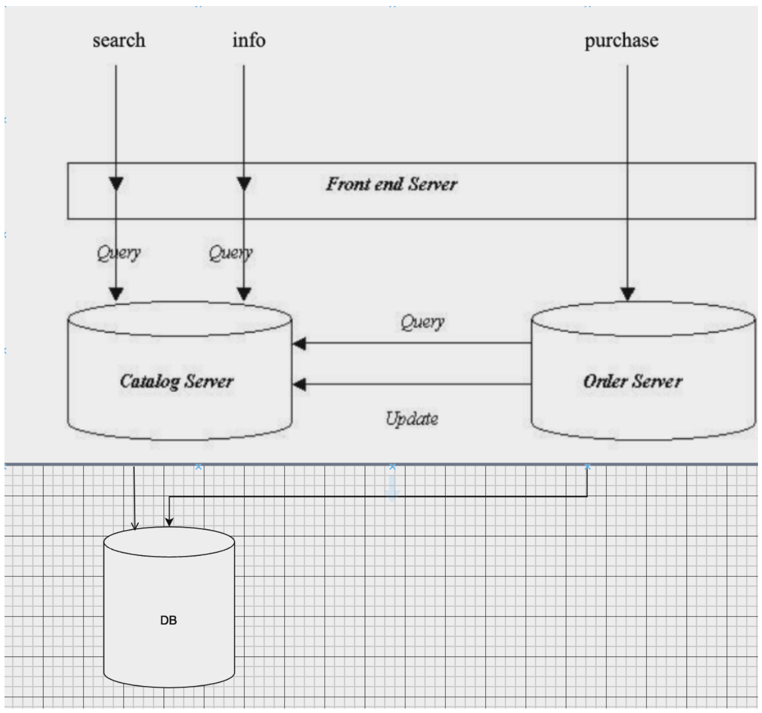
List of commands to run:

```
$ docker swarm init
$ docker compose build
$ docker compose up
```

And you can access the frontend node from `http://localhost:3000/`

7 How to expand the project

On an obvious hindsight, we can make a seperate service for database instance, and make it unrelated to the catalog server.
This will allow us to scale our database without the need of scaling the catalog service.



Another upgrade we can do is to make a custom load balancer depending on what makes the best performance between round robin, least response time, etc.. which will allow us to have a better performance.