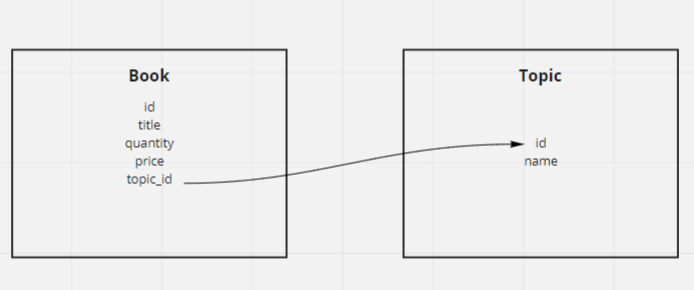
**DOS Bazar.com**

# System Design

As required in the homework specifications, the system’s design follows a microservice architecture in which the backend consists of three microservices:

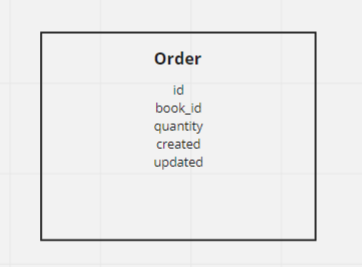
1. Catalog Microservice: This microservice is responsible for holding and manipulating data related to books catalog. The database consists of two tables as shown below:



This microservice exposes four endpoints:

* [GET] /search/{search\_phrase}: returns books with a topic that matches the search phrase
* [GET] /info/{book\_id}: returns the details of a book with the specified id
* [PUT] /update/{book\_id}: updates the book with the sent id using the values sent in the body
* [POST] /sync/{book\_id}: updates the book with the sent id using the values sent in the body (Used for syncing across replicas)

1. Order Microservice: This microservice is responsible for the orders details. It has a database with the following schema:



This microservice exposes two endpoints:

* [POST]/ purchase/{book\_id}: this endpoint checks first if the book exists by quering the catalog microservice, if it exists then it decrements the quantity value by using the update endpoint in the catalog microservice and it stores the order entry
* [POST]/sync/{book\_id}: adds an order for the specified book to the DB (used for syncing across replicas)

1. Cache Microservice: This microservice contains an in-memory cache that it uses to make API calls faster and it also implements a round-robin load balancing algorithm when calling the Catalog and Order microservices.

This microservice exposes five endpoints:

* [GET]/search/{topic}: a cached load-balanced endpoint that calls Catalog search
* [GET]/info/{book\_id}: a cached load-balanced endpoint that calls Catalog info
* [PUT]/update/{book\_id}: a load-balanced endpoint that calls Catalog update
* [POST]/purchase/{book\_id}: a load-balanced endpoint that calls Order purchase
* [DELETE]/cache/{book\_id}: an endpoint to invalidate a specific book cache

On the other hand, the frontend runs on its own separate server to and consists of three main parts that use the backend MSs:

1. search.sh: asks the user for the search phrase then queries the catalog microservice.
2. info.sh: asks the user for the book id then queries the catalog microservice.
3. purchase.sh asks the user for the book id then posts the request to the order microservice.

# Implementation & Deployment

### Backend Implementation:

Flask framework was used to implement the backend microservices because of both its ease of use and being a lightweight framework, which is suitable for our use case.

Alongside Flask, SQLAlchemy is used for object relational mapping (ORM) with the SQLite database which offers the capabilities to easily update the DB.

### Caching:

For caching on the cache server, we used an in-memory cache by using a library called Flask-Caching, we provided cache for the info and search endpoints. Cache for a book is invalidated through a [DELETE] cached endpoint on the cache server to ensure cache consistency.

### Frontend Implementation:

As we were not required to implement a GUI, I implemented the frontend as a group of bash scripts each of them calls a different API. These bash scripts use cURL to make the API calls and use python to encode inputs for URLs and for formatting the output.

### Deployment:

To deliver a unified deployment experience, we used Docker to containerize each of the microservices. Each microservice has its own Dockerfile which can be used to easily create a docker image from the files and then run a container from it.

# How to Run It

As stated before, Docker was used for the deployment of the MSs. A detailed description on how to run each part of the project is provided in the README.md file in each part repository.

Before running any part of the project, we need to make sure that Docker is installed then follow these steps:

1. Create a docker network: **docker network create --subnet=172.18.0.0/16 bazar**
2. For each MS, create a docker image using the Dockerfile by following the docker build comand in the README.md
3. Run a container using the docker run command in the README file

For frontend you can run the following commands now:

1. For search run the command **sh search.sh**
2. For info run the command **sh info.sh**
3. For purchase run the command **sh purchase.sh**

# Performance & Measurements:

We made 5 experiments without cache and the result in milliseconds were:

[12ms, 12ms, 12ms, 14ms, 12ms]

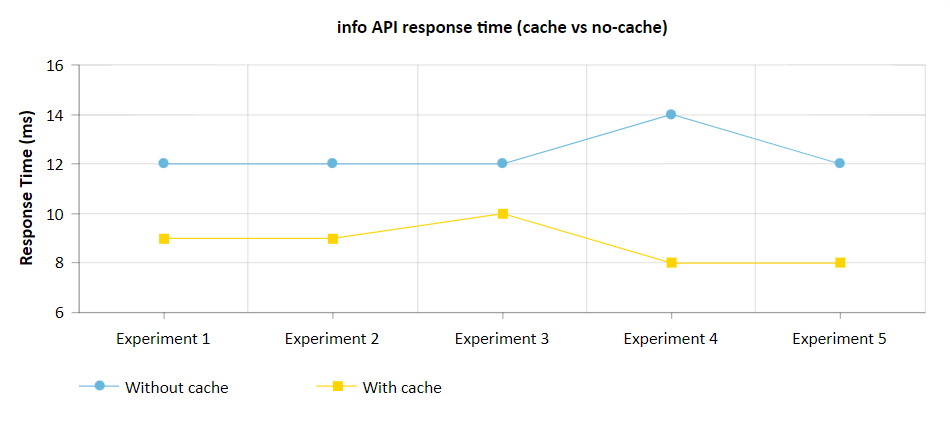
Average = 12.4ms

We made 5 experiments with cache and the result in milliseconds were:

[9ms, 9ms, 10ms, 8ms, 8ms]

Average = 8.8ms

The result of these experiments are illustrated in the graph below:



Based on the above, in average cache helps in reducing time by 12.4ms – 8.8ms = 3.6ms

The overhead due to cache miss is 3.6ms