COP5570 Term Project Report
Title: **Online Marketplace**

Group Members:
Mahidhar Reddy Narala
Sai Jyothi Attuluri
Bharath Seshavarapu

Type of Project

The type of project chosen is a software-development oriented one. We are choosing this to develop a new software system from scratch to understand, implement and evaluate concurrent, parallel, and distributed programming concepts.

Objectives

The objective of our project is to develop a software called ‘Online Marketplace’ an e-commerce software where people can buy/sell products, much like Amazon, but on a much simpler and smaller scale. We are only interested in developing the backend part of the software as developing a frontend UI or user interface is out of scope for this course. The aim is to design the software as a microservice inorder to implement concurrent parallel and distributed concepts that were learnt as part of this course.

The software should have the following components at the least:

1. ***User** module*: The user management module will allow users to create an account, and manage their profile.
2. ***Product** Module*: The product management module will allow sellers/vendors to create, edit, and view products. The products will be stored in a database and will include product information such as name, category, description and price.
3. *Product **Search***: The product search module will allow users to search for products based on their name or category. The search functionality will be implemented in a very basic manner as going deep into search and implementing things like NLP is out of project scope.

4. ***Order** Module*: The order management module will allow users to create, edit, and view orders. The orders will be stored in a database and will include order information such as product name, quantity, price etc.

In addition, to test the scalability of the software, a tool is to be developed which will test the performance of the software. So, a manner of load/stress tests must be performed and metrics are to be gathered for the software with respect to *response time*. This is to know the limits of the software so that it can be prepared to support real world user traffic. The test metrics must be best in class.

Research - System design

Several e-commerce websites like Amazon, Walmart, and Flipkart have been researched to understand how these companies design their backend softwares. A common theme for all their softwares is that they all use some sort of microservice framework. This is to maintain separation of concern for each of their services and to scale their services independently with user traffic. So, we have chosen a popular microservice framework ***Spring Boot***. Spring Boot provides out of the box libraries to create REST APIs and host them in an embedded server (tomcat). This allows us to rapidly create APIs for our purpose without hassle. Moreover, spring boot provides finer control on the number of threads the software can utilize to serve concurrent Http requests and utilize database connections.

The programming language chosen to implement all of this is ***Java8***. Java is one of the best choices for programming languages used by millions of microservices across the world. To manage the libraries required for the software, we used ***Maven*** as a library management framework.

For the database, we have chosen to use ***MySQL*** database as it is a popular RDBMS where intrinsic checks can be kept at tables itself like unique usernames, and FK relations between users and orders etc. In addition, it has better indexing capabilities to provide faster search query responses.

To test the software performance, instead of using any existing tool/framework out there, we decided to develop our own tool called *API Tester* as this tool also incorporates the concurrent and parallel concepts we learnt as part of the course. So, we chose to create a **Python** script to test the scalability of our software in terms of response time.

Software

Online Marketplace (OMP)

The code for the OMP software has been created as a maven java project.
The database used as mentioned is MySQL.

Database

For the OMP software, we basically require 3 tables which are defined as follows:

- **User**

id	**username**	**email**	**address**

id is the primary key, *username* and *email* have unique constraints

- **Product**

id	**category**	**name**	**price**	**quantity**	**address**

id is the primary key

- **Orders**

id	**user_id**	**product_id**	**quantity**	**address**	**status**

id is the primary key, *user_id* and *product_id* are the foreign keys to USER table, PRODUCT table respectively.

To prepare the database, the following database script can be executed:

```
CREATE DATABASE omp;

USE omp;

CREATE TABLE `User` (
`id` int NOT NULL AUTO_INCREMENT,
`username` varchar(100) NOT NULL,
`email` varchar(100) DEFAULT NULL,
`address` varchar(100) DEFAULT NULL,
PRIMARY KEY (`id`),
UNIQUE KEY `username_UNIQUE` (`username`)
) ENGINE=InnoDB AUTO_INCREMENT=17 DEFAULT CHARSET=utf8mb3;

CREATE TABLE `Product` (
`id` int NOT NULL AUTO_INCREMENT,
`category` varchar(45) NOT NULL,
`name` varchar(100) DEFAULT NULL,
`price` varchar(45) DEFAULT NULL,
`quantity` int DEFAULT NULL,
`address` varchar(100) DEFAULT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=19 DEFAULT CHARSET=utf8mb3;

CREATE TABLE `Orders` (
`id` int NOT NULL AUTO_INCREMENT,
`user_id` int NOT NULL,
`product_id` int NOT NULL,
`quantity` int NOT NULL,
`address` varchar(100) NOT NULL,
`status` varchar(45) NOT NULL,
PRIMARY KEY (`id`),
KEY `user_id_idx` (`user_id`),
KEY `product_id_idx` (`product_id`),
CONSTRAINT `product_id` FOREIGN KEY (`product_id`) REFERENCES `Product`
(`id`),
CONSTRAINT `user_id` FOREIGN KEY (`user_id`) REFERENCES `User` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8mb3;
```

API

The APIs available for each model are described as below. These *curl*s can be executed directly from the command line to hit the API and get responses.

User module

- Add user

```
curl --location 'http://localhost:8080/user/add' \
--header 'Content-Type: application/json' \
--data-raw '{
"username": "user9",
"email": "user@email.com",
"address": "address and pincode"
}’
```

- Get all users
```
curl --location 'http://localhost:8080/user/get_all'
```

- Get user by user id or username
```
curl --location 'http://localhost:8080/user/get?id=0&username=user2'
```

- Update user
```
curl --location --request PUT 'http://localhost:8080/user/update' \
--header 'Content-Type: application/json' \
--data-raw '{
"id": 16,
"username": "user99",
"email": "user@email.com",
"address": "address and pincode"
}'
```

- Delete user by user id or username
```
curl --location --request DELETE 'http://localhost:8080/user/delete?id=0&username=user3'
```

- Delete all users
```
curl --location --request DELETE 'http://localhost:8080/user/delete_all'
```

Product module

- Add product
```
curl --location 'http://localhost:8080/product/add' \
--header 'Content-Type: application/json' \
--data '{
"category": "books",
"name": "House of Dragons",
"price": 19.99,
"quantity": 100
}'
```

- Get all products
```
curl --location 'http://localhost:8080/product/get_all'
```

- Get product by id, name or category
```
curl --location 'http://localhost:8080/product/get?id=1&name=&category=books'
```

- Update product
```
curl --location --request PUT 'http://localhost:8080/product/update' \
--header 'Content-Type: application/json' \
--data '{
```

```
"id": 7,
"category": "books",
"name": "Harry Potter",
"price": 19.99,
"quantity": 100
}'
```

- Delete product by id or name
  curl --location --request DELETE 'http://localhost:8080/product/delete?id=1&name=House%20of%20Dragons'

- Delete all products
  curl --location --request DELETE 'http://localhost:8080/product/delete_all'

Search Module

curl --location 'http://localhost:8080/search?query=books'

Orders Module

- Place order
  ```
  curl --location 'http://localhost:8080/order/place' \
  --header 'Content-Type: application/json' \
  --data '{
  "userId": 8,
  "productId": 7,
  "quantity": 10,
  "address": "Student Union, FSU"
  }'
  ```

- Get all orders
  curl --location 'http://localhost:8080/order/get_all'

- Get order by order id or user id or product id
  curl --location 'http://localhost:8080/order/get?id=0&userId=8&productId=7'

- Update order
  ```
  curl --location --request PUT 'http://localhost:8080/order/update' \
  --header 'Content-Type: application/json' \
  --data '{
  "id": 1,
  "userId": 8,
  "productId": 7,
  "quantity": 100,
  "status": "SHIPPED",
  "address": "Student Union, FSU"
  ```

```
}'
```

- Delete order by id or user id
  curl --location --request DELETE 'http://localhost:8080/order/delete?id=1&userId=8'

- Delete all orders
  curl --location --request DELETE 'http://localhost:8080/order/delete_all'

Build

To build the software and create executable jar, enter the following:

```
$ mvn clean install -DskipTests
```

Run

To run the software, execute the following:

```
$ java -jar -Dserver.port=8080 -Dserver.tomcat.threads.max=500 ./target/omp-1.jar
```

NOTE: All the spring, thread, server, and database config are present in *application.properties* file. These config can be passed as command line arguments as well at runtime if required.
e.g:-

```
-Dspring.datasource.url=jdbc:mysql://localhost:3306/omp
-Dspring.datasource.username=root
-Dspring.datasource.password=
-Dlogging.level.edu.fsu.*=DEBUG
```

API Tester

This is the tool we've developed on our own to test the performance of our OMP software in terms of scalability. This is essentially a python script which is written in a configurable way to test any REST API with respect to response time. The tool will send concurrent API requests to multiple processes in a round robin fashion to ensure fairness. The APIs themselves will be chosen randomly. It then checks all the responses and outputs the success rate, average response time and certain percentiles of response time as well.

The tool is written in python version 3.9. It uses several libraries as part of the script. All the libraries are included in the *requirements.txt* file. It contains a config file which the tool reads to get the test config to test with and the API config to test. The details on how to set the config file are as follows:

num_requests: Total number of requests to send as part of test.
rps: Number of requests per second.

max_threads: Maximum number of threads to assign for tests.
processes: The url of the processes (just hostname:port).
apis: A list of config names of the API to be included in the test.
[...]: The API config name which has the API endpoint details
request_type, api_url, payload, headers

Run

To run the tool, make sure you have set the api_config properly and execute the following:

```
$ python3 api_tester.py
```

Metrics & Conclusion

To test the scalability of the OMP software and know its limits, we have run the software in multiple configurations with different numbers of threads for concurrent and parallel execution and with multiple instances running for distributed execution.

We have used the API tester tool that was developed to test the performance of our APIs with respect to response time. Here are the results for 10,000 requests each:

Server Config	**Success rate**	**Average response time**	**p99**	**p90**	**p75**	**p50**	**p25**
1 process, 10 threads	100%	11.69	16.28	13.96	12.67	11.46	10.60
1 process, 100 threads	100%	10.54	16.13	13.21	11.87	10.55	9.39
1 process, 200 threads	100%	12.74	24.60	17.69	14.47	12.82	11.22
2 processes, 200 threads each	100%	1.56	6.08	3.11	1.98	1.45	0.44
3 processes, 200 threads each	100%	*0.95*	3.19	1.74	1.19	0.84	0.49
4 processes, 200 threads each	100%	3.78	9.70	6.97	4.78	3.06	2.43
5 processes, 200 threads each	*99.96%*	6.88	49.82	27.60	4.13	2.04	0.8

From the load/stress tests, we can see that we were able to scale our software up to less than 1 sec response time with optimal config of 3 servers running simultaneously with 200 threads each. This is with around 500 requests sent at time to all servers.

As we can see, with more servers, the response time is getting worse. After debugging this, we found out that the bottleneck was actually the database itself. The only thing stopping from more horizontal scaling is the database. Since we are using a single database for all servers, due to the limited number of connections of the database, the server threads were kept waiting for connections to get freed up and this is increasing the response time.

Future improvements

Based on the challenges seen with the database, we could resolve this issue with a distributed database structure. We can utilize a master slave architecture of distributed databases to maintain synchronization so that each instance can have its own database. This will allow us to horizontally scale our system to handle user traffic.

We can even split up some parts of the database which don't require relations with other tables like Users etc to NoSQL databases like MongoDb etc for faster access. This in conjunction with caching at process level for products search etc will make our response times even quicker.

The search functionality implemented is fairly simple and we can make it more easier for users by implementing Natural Language Processing etc. With all these improvements, we are confident that our OMP software can serve as a best in class e-commerce platform.