Assignment 2

Yiwen Zhang

1. RepoURL: https://github.com/ywenzhang/Distributed-System
2. Server Design:

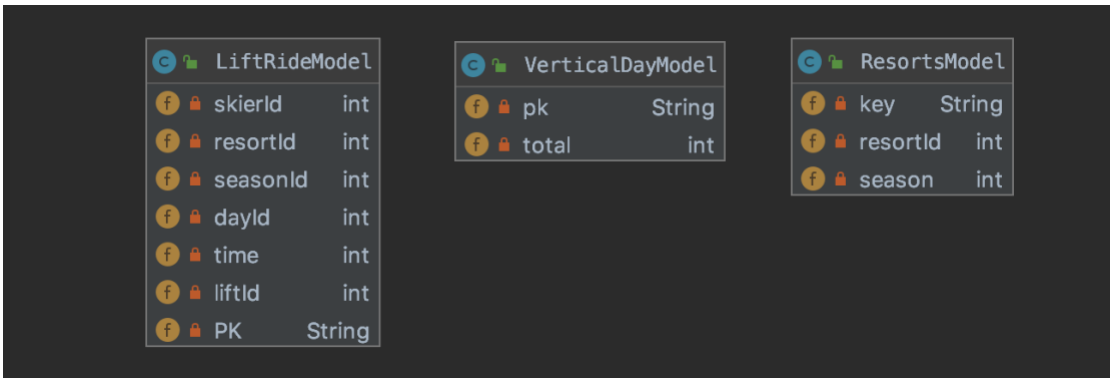
   I used JDBC and MySQL for this assignment.

   There are four packages in my project:

   - ConnectionManager - to set parameters of the servelets and connection to the database management system.
   - Dao – to connect the servlets with the database management system.
   - Model –to create instances to transfer data between Daos and Servlets.
   - Servlet –to get the data from http request and post data to the database management system

   (1) Models

   I have three tables to realize the API methods: LiftRide, Resorts and Vertical.



   LiftRide is the table to record each LiftRide. PK in LiftRide is a string which is the primary key of the table generated by concatenating skierId, seasonId, dayId and time.

```
String PK = skierId + "/"  +seasonId + "/" + dayId + "/" + time;
```

   VerticalDay is the table to record the total vertical for a skier at a resort in a season on a day. Pk is also the primary key generated by concatenating skierID, resortID, seasonID and dayID

```
this.pk = skierID + "/" + resortID + "/" + seasonID + "/" + dayID;
```

   Resorts is the table is to record the season at a resort. The key is the primary key generated by concatenating resortId and season.

```
this.key = resortId + "/" + season;
```

   (2) Servlets

   1. SkierServlet

doGet:	doPost:

if the url is valid:
  if the url.length == 8:
  VerticalDayDao.getVerticalDay
  else:
  VerticalDayDao.getVerticalTotal
	if the url is valid:
	  LiftRideDao.createLiftRide
	  VerticalDay.createOrUpdateVerticalDay

2. ResortServlet

doGet:	doPost:

if the url is valid:
  if the url.length == 1:
  VerticalDayDao.getResortList
  else:
  VerticalDayDao.getResortSeason
	if the url is valid:
	  LiftRideDao.createResort

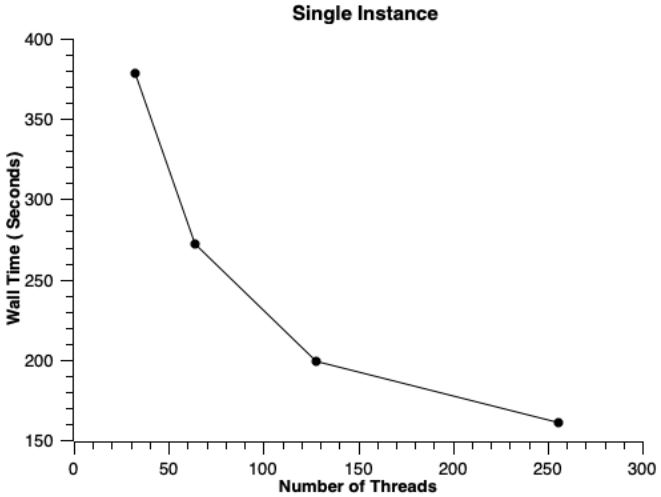
3. Single Server Test: I deployed both the client and server on the cloud to avoid the networking latency.

```
[ec2-user@ip-172-31-33-166 ~]$ java -jar client2v2.jar 256
256
total time spent: 161 s
number of threads: 256
number of successful requests: 440000
number of unsuccessful requests: 0
mean response time: 73 ms
median response time: 79 ms
throughput: 2722.435342160624 /s
p99: 192 ms
max response time: 440 ms
```

```
[ec2-user@ip-172-31-33-166 ~]$ java -jar client2v2.jar 128
128
total time spent: 199 s
number of threads: 128
number of successful requests: 440000
number of unsuccessful requests: 0
mean response time: 41 ms
median response time: 42 ms
throughput: 2204.7954300603815 /s
p99: 111 ms
max response time: 454 ms
```

```
[ec2-user@ip-172-31-33-166 ~]$ java -jar client2v2.jar 64
64
total time spent: 272 s
number of threads: 64
number of successful requests: 440000
number of unsuccessful requests: 0
mean response time: 26 ms
median response time: 26 ms
throughput: 1616.5475687491964 /s
p99: 84 ms
max response time: 264 ms
```

```
[ec2-user@ip-172-31-33-166 ~]$ java -jar client2v2.jar 32
32
total time spent: 378 s
number of threads: 32
number of successful requests: 440000
number of unsuccessful requests: 0
mean response time: 23 ms
median response time: 24 ms
throughput: 1162.5940644288494 /s
p99: 47 ms
max response time: 295 ms
```



For the single instance case, the wall time decreases with increase of number of threads. The wall time is sensible as the lower bound of the mean response time is around 20 ms. When number of threads is higher, the requests were sent concurrently and it reduced the wall time. However, the mean response time also increases with the increase of threads. This is because the server is overloaded, and the number of threads has exceeded the total number of threads that a single server can handle. One evidence is that the CPU occupancy of the single server has reached 70%. The other evidence is that the maximum number of requests that a single server can handle is around 60. Therefore, we can expect that the mean response time for four instances with a load balancer with different number of threads will have around the same response time which is around 20 ms.

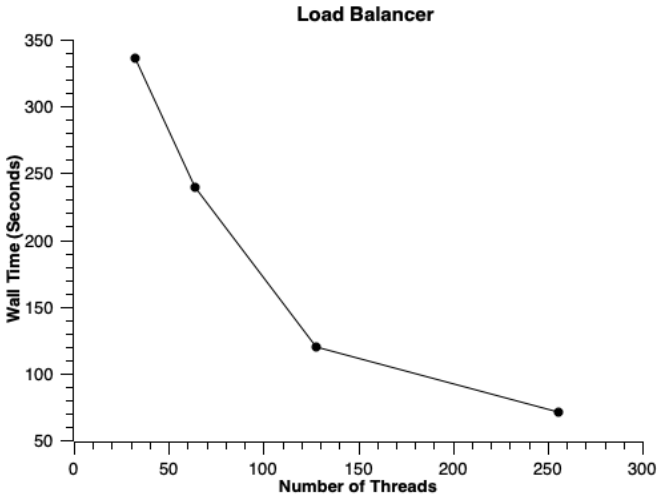
4. Load Balancer Test:

```
^C[ec2-user@ip-172-31-33-166 ~]$ java -jar client2v2.jar 32
32
total time spent: 336 s
number of threads: 32
number of successful requests: 440000
number of unsuccessful requests: 0
mean response time: 20 ms
median response time: 21 ms
throughput: 1308.9550373944653 /s
p99: 39 ms
max response time: 267 ms
```

```
^C[ec2-user@ip-172-31-33-166 ~]$ java -jar client2v2.jar 64
64
total time spent: 239 s
number of threads: 64
number of successful requests: 440000
number of unsuccessful requests: 0
mean response time: 22 ms
median response time: 22 ms
throughput: 1834.510477556432 /s
p99: 60 ms
max response time: 341 ms
```

```
[ec2-user@ip-172-31-33-166 ~]$ java -jar client2v2.jar 128
128
total time spent: 120 s
number of threads: 128
number of successful requests: 440000
number of unsuccessful requests: 0
mean response time: 22 ms
median response time: 23 ms
throughput: 3644.9790413705123 /s
p99: 47 ms
max response time: 299 ms
```

```
[ec2-user@ip-172-31-33-166 ~]$ java -jar client2v2.jar 256
256
total time spent: 71 s
number of threads: 256
number of successful requests: 440000
number of unsuccessful requests: 0
mean response time: 27 ms
median response time: 28 ms
throughput: 6189.946963408973 /s
p99: 74 ms
max response time: 547 ms
```



As we can see, with load balancer we have three other instances to handle the requests. The server are evenly loaded with requests, therefore no sever was overloaded. The mean response time since dramatically reduced. I got the wall time for 256 threads better than running on my local machine.

5. Runtime Statistics Collection:

I used Servlet Context to record three values: average latency, max response time and number of request in a hashmap. And I calculated the average using running average. Each time I call a get method or a post method, I will check whether I have recorded the hashmap for the specific endpoint. Then, I will create or update the hashmap:

Create:

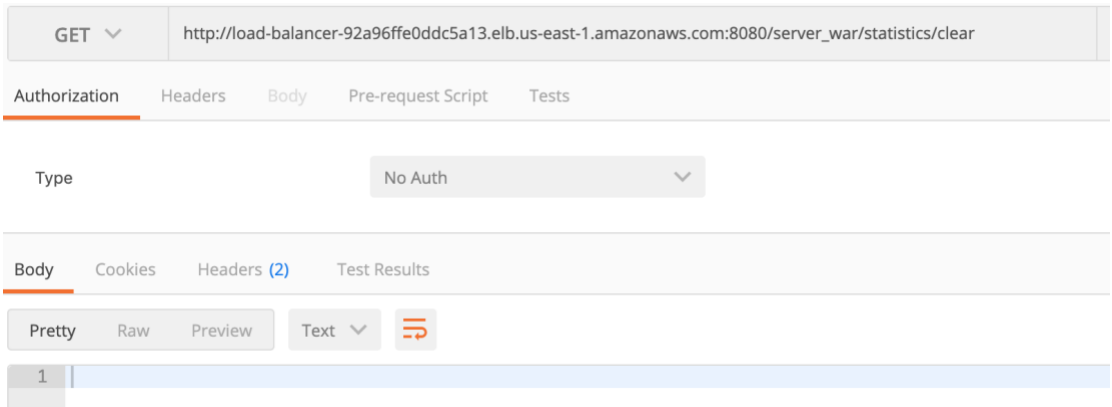
```
if (sc.getAttribute(API) == null)

{

   HashMap<String,Long> hashMap = new HashMap<>();

   hashMap.put("count",Long.valueOf(1));
```

```
    hashMap.put("max",responseTime);

    hashMap.put("average",responseTime);

    hashMap.put("method",Long.valueOf(1));

    sc.setAttribute(API,hashMap);

}
```

Update:

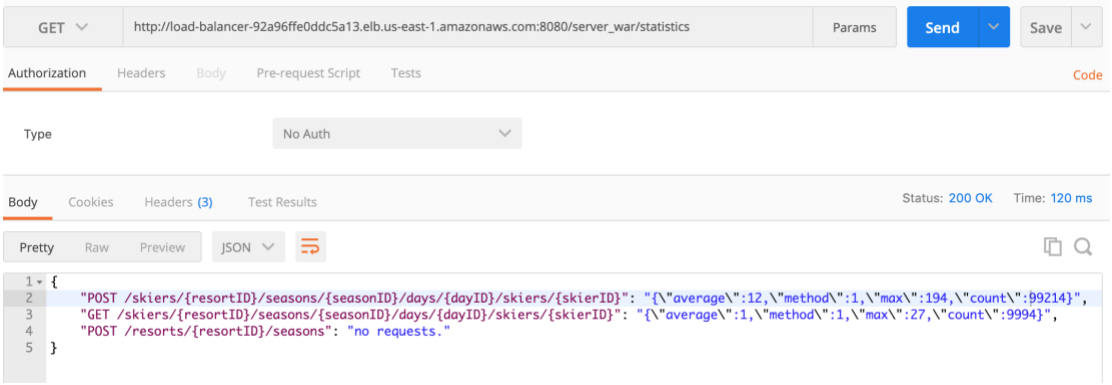
```
if (sc.getAttribute(API) != null){

HashMap<String,Long> hashMap = (HashMap<String, Long>) sc.getAttribute(API);

long count = hashMap.get("count");

long averageToUpdate = hashMap.get("average");

long maxToUpdate = hashMap.get("max");

if(maxToUpdate<responseTime){

    hashMap.put("max",responseTime);

}

hashMap.put("average", (averageToUpdate*count+responseTime)/(count+1));

hashMap.put("count", count+1);}
```

Before doing the test, we simply clear the statistics by sending this request in postman.



After the program terminated, I immediately send the request to the load balancer. I got the following result:

We can see that the load balancer works pretty well since one server received nearly 100000 post request and 10000 get requests and the total number requests is 44000. I used four tires. How smart the AWS people are! The average is 13 ms to write. Therefore, we spent 7 ms for each request on the data exchange between the server and the client. The get average at the very beginning shocked as I couldn't believe it is 1 ms. But, I reminded what I learnt in database. I understood. MySQL uses LRU cache!!!



I love this assignment as I learnt a lot by messing up a lot of things. I am also very grateful for the extension that you gave to me. I will feel more confident in the interviews after taking this course! Thanks a lot.