ASSIGNMENT 2

**(CS 6650 - Building Scalable Distributed Systems)**

**Rahul Pandey**

**Github:**

[**https://github.com/rahulpandeycs/bsds6650-Course-fall2020/tree/master/Assignment%203**](https://github.com/rahulpandeycs/bsds6650-Course-fall2020/tree/master/Assignment 3)

**Results comparison (10 points) -**

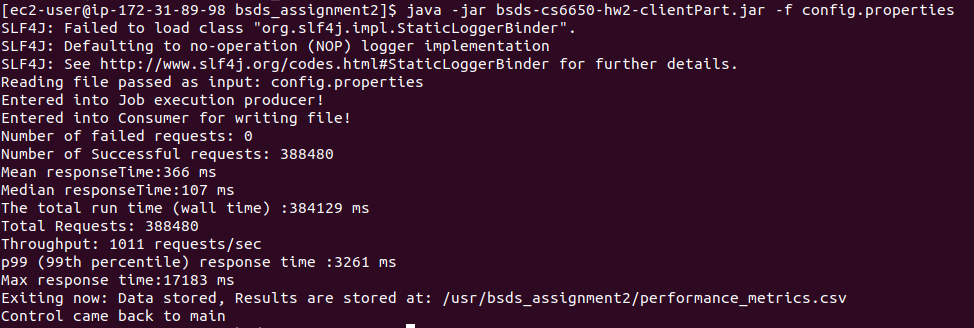
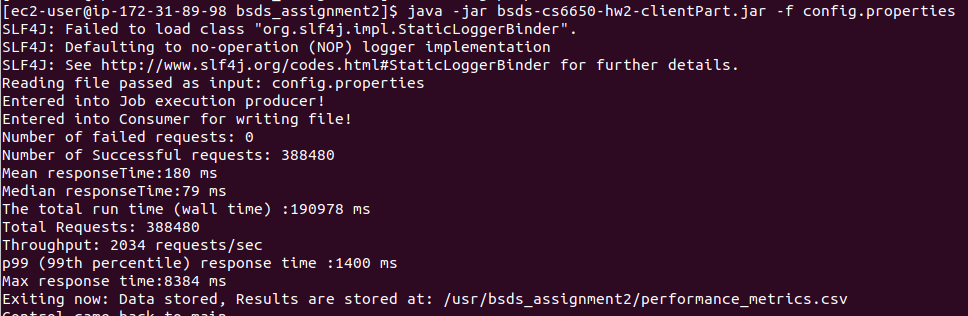
**Throughput:**

Compare and discuss through put values for run on 256 threads.

**Persistent/non-persistent queues:**

**256 Threads :**

**Persisted Queue:**

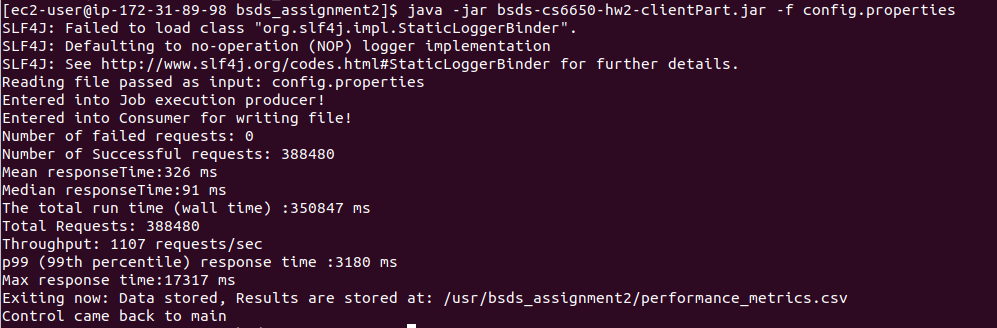
**Non persisted queue: (Run again)**

**Different instances : (Single/Load balanced)**

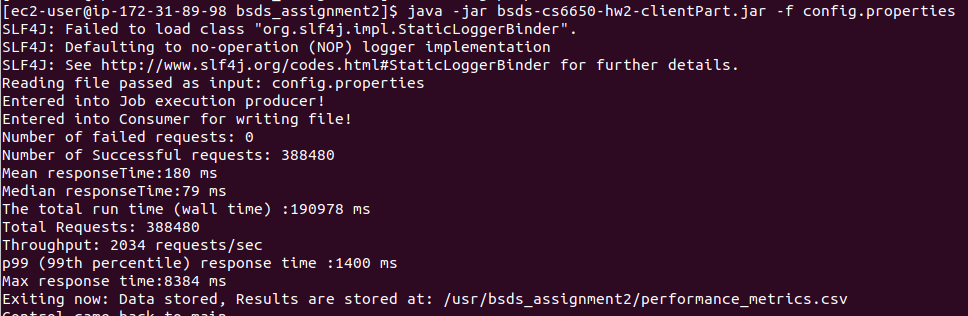
### A comparison of the results for 256 client threads between this assignment 3 and assignment 2, both single server and load balanced:

### Assignment 2, Single server vs Assignment 3 Single server (256 Threads)

**Assignment 2:**

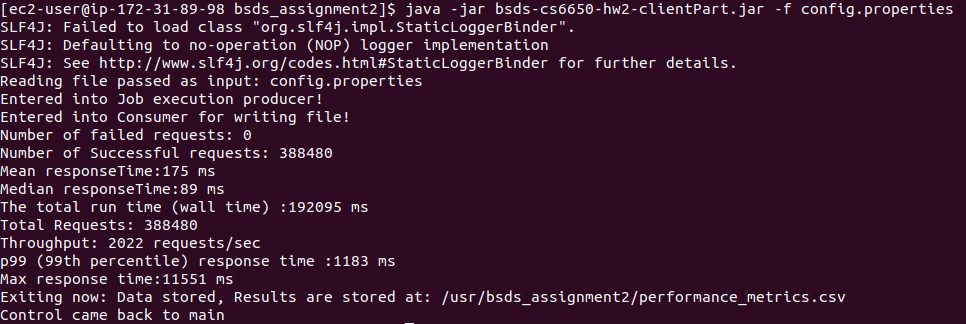
****

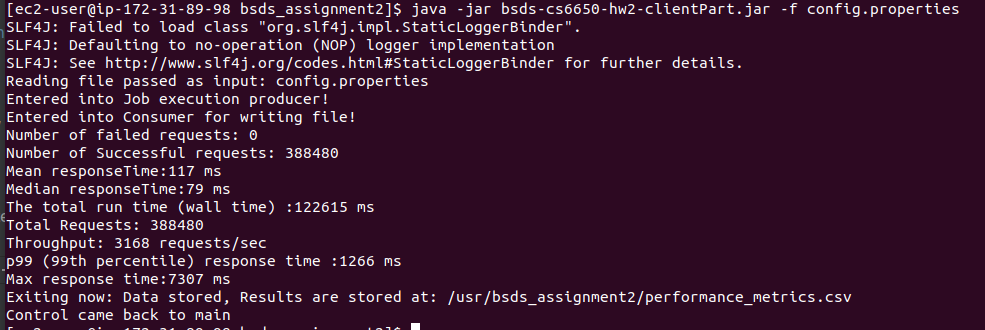
**Assignment 3:**

****

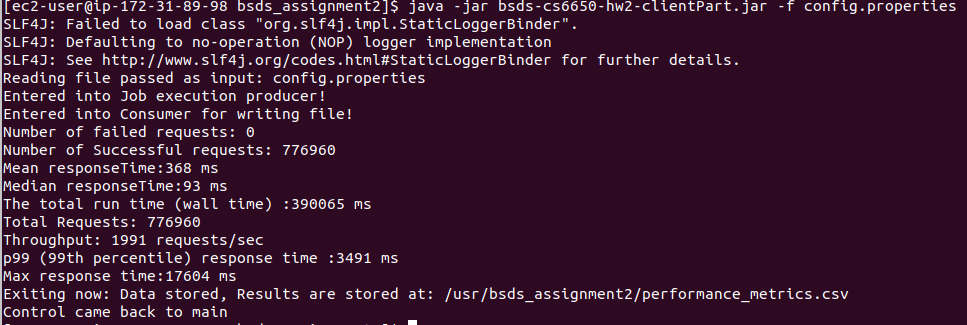
### Assignment 2, Load balanced vs Assignment 3 Load balanced (256 Threads) (Free 4 Ec2 Tier, Single RDS Instance with Free tier plan)

**Assignment 2 :**



**Assignment 3:**

**Results for a test with 512 client threads :**

****

## **Load Testing**

1. Do I need load balancing? Or can my system work with 1 free-tier (or slightly upgraded) server:

For my server i was able to run

2. How many consumers nodes do I need?

I used 20 Consumer to achieve eventual consistency

3. GETs still need to access the database? Can a cache be used to make GETs faster? How would the cache be updated?

I leveraged Redis server for caching my read results and invalidating them during a read.

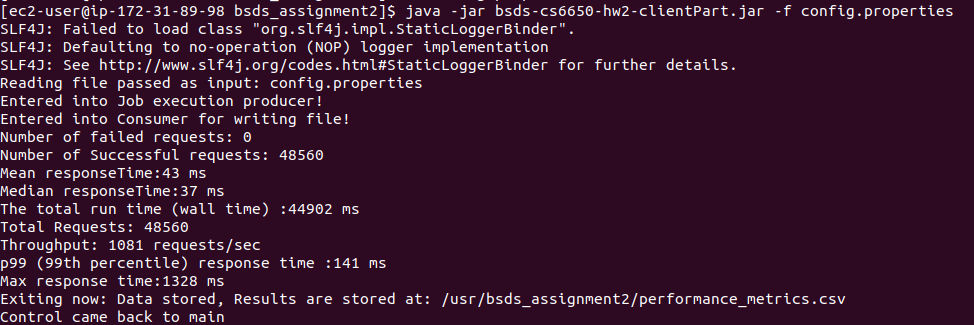
**Running logs:**

[https://github.com/rahulpandeycs/bsds6650-Course-fall2020/tree/master/Assignment%202/RunLogs](https://github.com/rahulpandeycs/bsds6650-Course-fall2020/tree/master/Assignment 2/RunLogs)

**Single server tests:**(Free 1 Ec2 Tier, Single RDS Instance with Free tier plan)

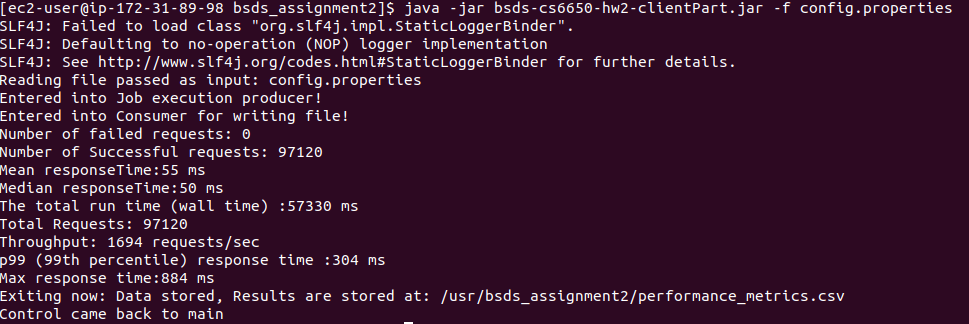
32 Threads

ScreenShot:



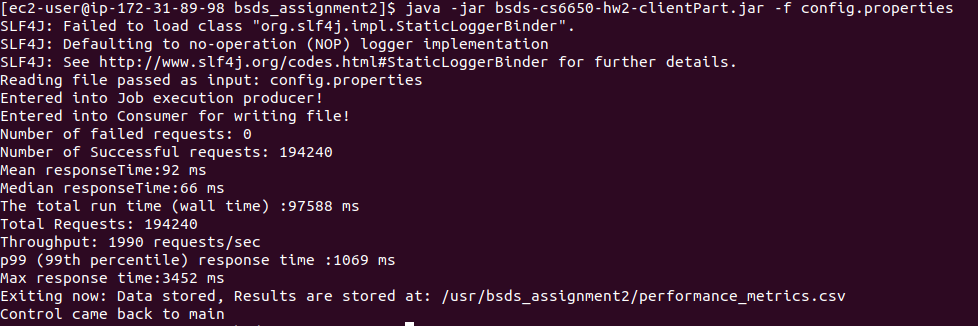
64 Threads

ScreenShot:



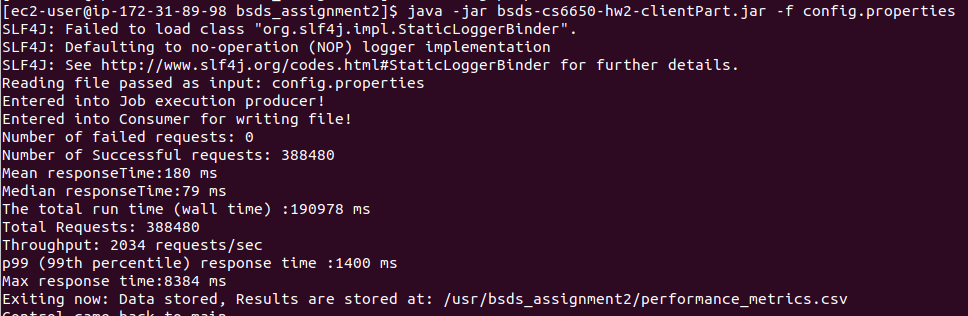
128 Threads

ScreenShot:



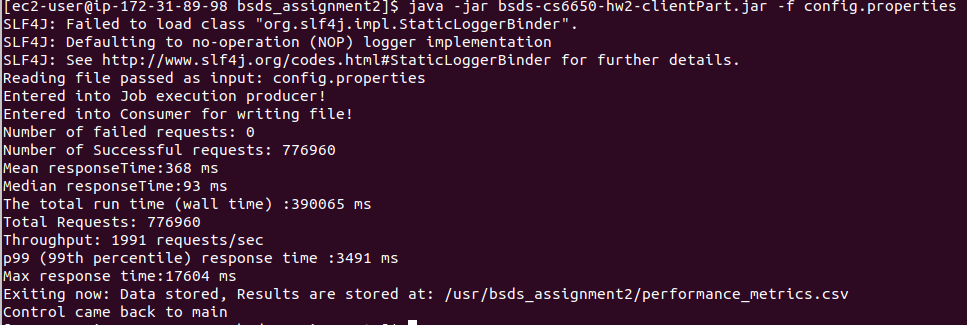
256 Threads

ScreenShot:



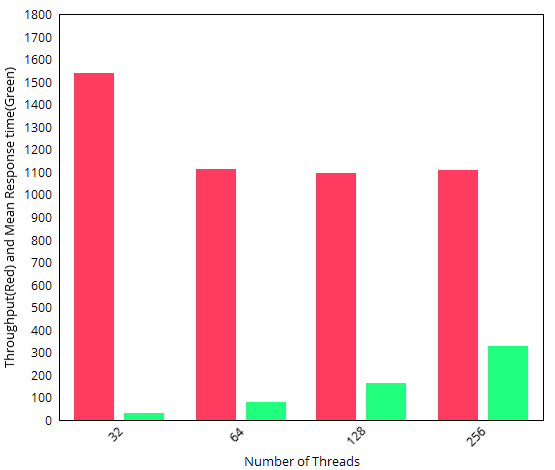
512 Threads

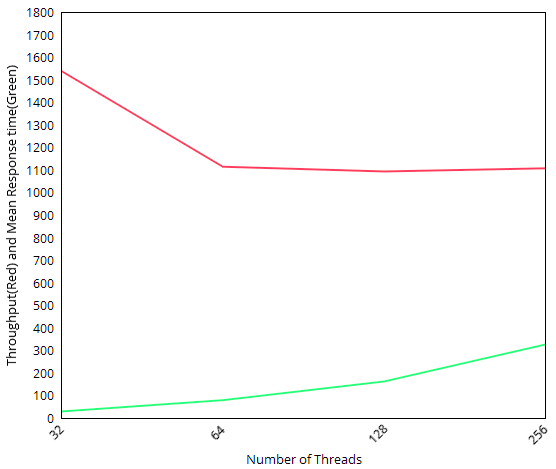
ScreenShot:



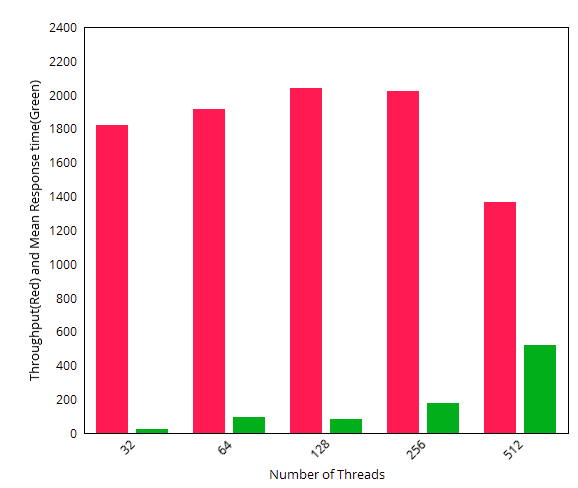
**Throughput and mean response by the number of threads**:

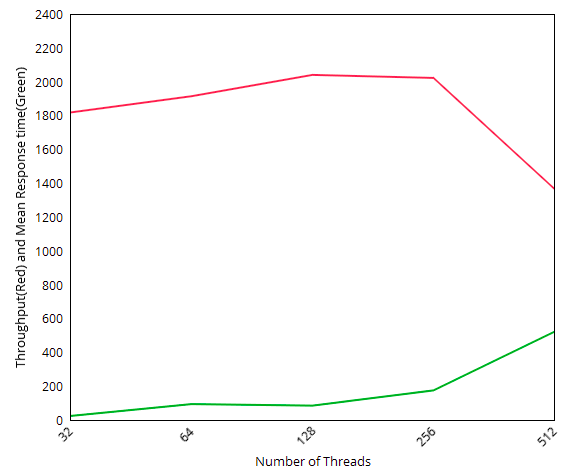
Single Instances tests:





Load Balanced tests:





**UML diagrams:** (Also included with detailed section below)

Server:

[https://github.com/rahulpandeycs/bsds6650-Course-fall2020/tree/master/Assignment%202/CS6650-hw2-Server/UML](https://github.com/rahulpandeycs/bsds6650-Course-fall2020/tree/master/Assignment 2/CS6650-hw2-Server/UML)

Client:

[https://github.com/rahulpandeycs/bsds6650-Course-fall2020/tree/master/Assignment%202/bsds-cs6650-hw2-clientPart/UML](https://github.com/rahulpandeycs/bsds6650-Course-fall2020/tree/master/Assignment 2/bsds-cs6650-hw2-clientPart/UML)

**Executable Jars:**

[https://github.com/rahulpandeycs/bsds6650-Course-fall2020/tree/master/Assignment%202/ExeculatebleJars](https://github.com/rahulpandeycs/bsds6650-Course-fall2020/tree/master/Assignment 2/ExeculatebleJars)

**Running the application:**

The application is divided into 3 Parts:

* The Server
* The Client Part
* The load balancer

The server needs to be hosted and kept running either on Cloud (e.g AWS) or Localhost. The client will then need to modify the **resources/config.properties** to point to its address and execute calls.

A sample view of contents of config.properties looks like:

1. maximum number of threads to run (maxThreads - max 256)
2. number of skier to generate lift rides for (numSkiers - default 50000), This is effectively the skier’s ID (skierID)
3. number of ski lifts (numLifts - range 5-60, default 40)
4. the ski day number - default to 1
5. the resort name which is the resortID - default to “SilverMt”
6. IP/port address of the server

Config.properties

***cmd.maxThreads=32***

***cmd.numSkiers=20000***

***cmd.numLifts=60***

***cmd.skiDay=1***

***cmd.resortId=SilverMt***

***#Local***

***cmd.addressPort=***[***http://bsdscs6650-1923097914.us-east-1.elb.amazonaws.com:8080/CS6650-hw2-Server-deploy***](http://bsdscs6650-1923097914.us-east-1.elb.amazonaws.com:8080/CS6650-hw2-Server-deploy)

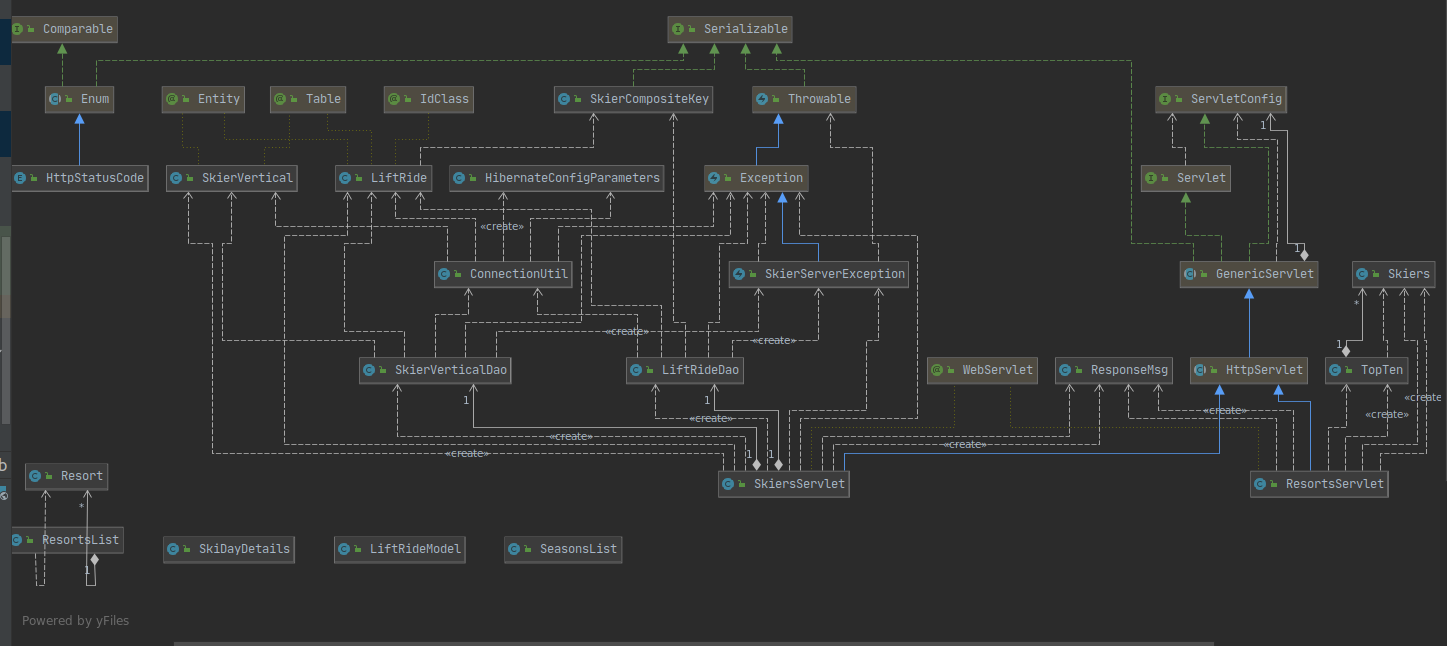
To run the application, the client needs to be packaged as .jar with configured config.properties. Then run as below:

**Client part: (Jar included in folder executable\_jar)**

java -jar bsds-cs6650-hw2-clientPart.jar -f config.properties

***Note: If no config.properties is provided it reads default config.properties***

## **The Server design description**



**The server exposes below API using Java Servlets:**

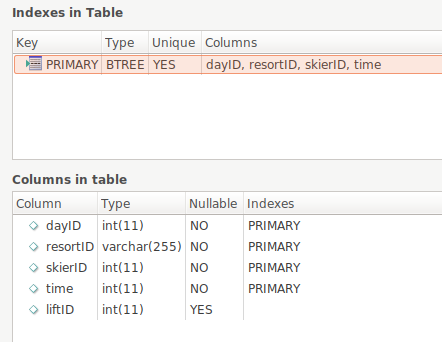
Skiers

**POST/skiers/liftrides**

**GET/skiers/{resortID}/days/{dayID}/skiers/{skierID}**

**GET/skiers/{skierID}/vertical**

**The Database Design**



The database is designed to keep in mind the responsiveness of the read and write API’s. Several refactorings were performed to choose the primary keys so that the database doesn’t incur locks during high load. Also it has been made sure that queries run faster so as to get higher throughput.

**Validations on server:**

The input to the doGet and doPost method is first validated to make sure they are in format:

*POST/skiers/liftrides*

*GET/skiers/{resortID}/days/{dayID}/skiers/{skierID}*

*GET/skiers/{skierID}/vertical*

The data inside the url i.e *{resortID}, {dayID}, {skierID}* is also validated to make sure they are of proper types. The data input to the skiers post method is validated to make sure it is of type liftRide. After validation and proper check the data is processed using Dao layer and response is returned in application/json format.

The Exception class SkierServerException is created to provide custom exceptions.

**JDBC Connection and pooling:**

The application has support for both **dbcp** and **hibernate** for connection to database. The current implementation is defaulted to **hibernate-c3p0.** The connection also uses the pooling feature and makes sure the max number of connections to the database is **15** at a time.

The application class ConnectionUtil provides the session via getSessionFactory() method. This session is used by the dao layer classes to establish connections for each query. The ConnectionUtil class ensures singleton creation of sessions.

**The Dao Layer**:

The database access layer (Dao) provides different methods to perform CRUD operations on respective databases. It uses ConnectionUtil class to establish the connection and then perform various operations like Read, update and create on the database.

Example:

*try (Session session = ConnectionUtil.getSessionFactory(LiftRide.class).openSession()) {*

*transaction = session.beginTransaction();*

*liftRide = session.get(LiftRide.class, ride);*

*transaction.commit();*

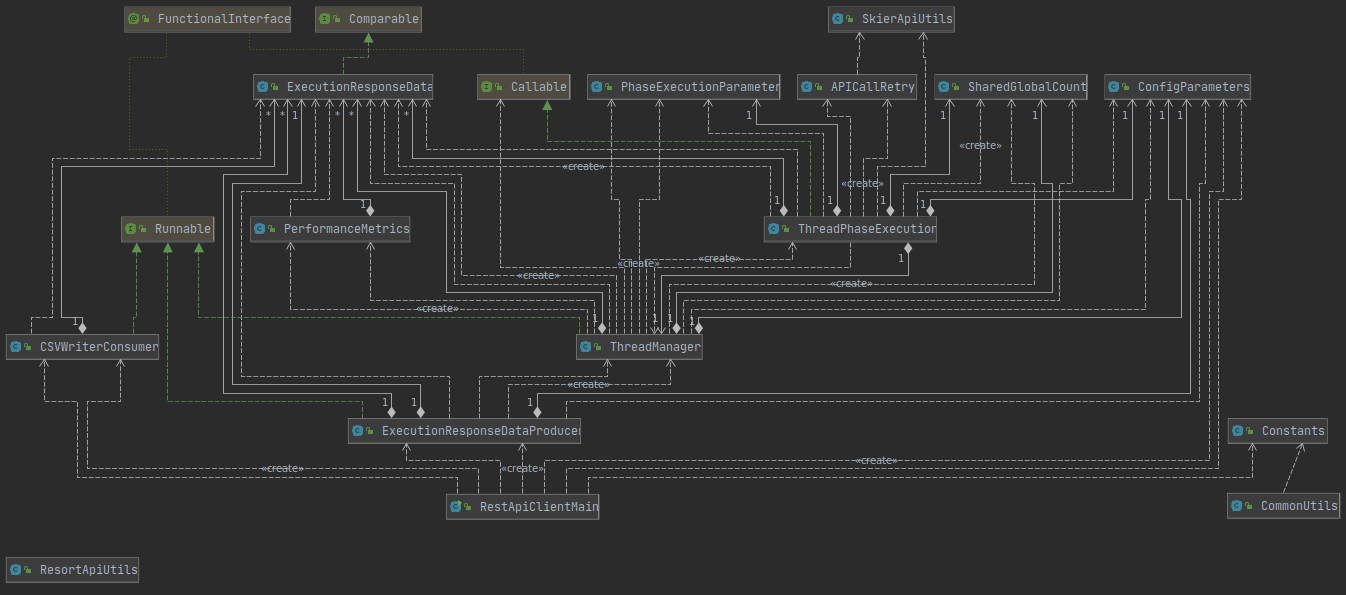
*}*

The server is connected to Single RDS instance with 1 Gb memory.

**Unit tests**:

## The junit tests are written to ensure that the server works properly and delivers desired results. The units test mock the servlets and use mockito to perform testing and various assertions.

## The Client Part



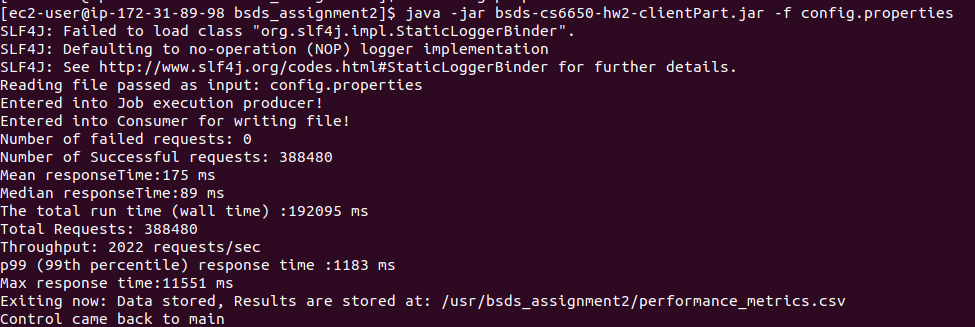
**The Startup phase**

The Client has been designed as a multi threaded client. The client execution starts with execution of the jar file and clients main class *RestApiClientMain* is called. The main class parses the input config.properties into a properties file and initializes *ConfigParameters* class.

The control is then passed to *ThreadManager* class which is executed in a separate thread then main. The threadManager takes care of threads phase execution by calling *ThreadPhaseExectution* class with their respective configuration as initialized in *PhaseExecutionParameter* Class.

The respective phases are then executed and after 10% of each phase is elapsed the next phase is started,this is taken care of by using CountDownLatch and initializing it to 10% of the total number of threads for a respective previous phase. As results need to be returned for each executed thread, Callable is used instead of runnable as in client part 1. The results are stored in list of *ExecutionResponseData* class, this list is then passed to *PerformanceMetrics* class which takes care of generating mean, median and various other performance metrics.

The execution then terminates with results printed to the console and CSV is generated from run results as stored in List of *ExecutionResponseData* Class. A sample output will look like:



**The Executor Framework:**

The executor framework has been used to create a fixed thread pool as:

*ExecutorService WORKER\_THREAD\_POOL = Executors.newFixedThreadPool(parameters.getMaxThreads() / 4 + parameters.getMaxThreads() + parameters.getMaxThreads() / 4);*

Phase threads are then passed to the thread pool for execution using method **submitToThreadPhaseExecution**(ExecutorService threadPool, PhaseExecutionParameter phaseExecutionParameter, double countDownThreshold**)** and submitted to thread pool as shown below.

//Creating a thread execution callable based on input to **submitToThreadPhaseExecution**

Callable<List<ExecutionResponseData>> phaseThread = new ThreadPhaseExecution(parameters, phaseExecutionParameter, this, latch);

//Submitting the phase execution thread to threadpool

Future<List<ExecutionResponseData>> futureExecutionResponseData = threadPool.submit(phaseThread);

//Adding results to blocking queue to write to the file later

blockingQueue.add(futureExecutionResponseData);

Each of these future calls are then later retrieved in the thread manager to build up the performance metrics and write data to CSV using LinkedBlocking queue.

**The API retry on request failure:**

The application has been designed to be fault tolerant and make sure it retries enough number of times before marking a request as fail.

It is expected that the call to the server will fail with unknown reasons considering several factors. Hence, each api call has been configured to retry at least 5 number of times before repeating the actual failure. The class APICallRetry has been created to cater to the API call failures and the calling class needs to create its class and call its respective methods to handle retry.

**The blocking queue to prevent memory overhead:**

It was observed that the memory could become a bottleneck as the number of API calls grow and size of the data structure being used to calculate metrics will exceed the available memory, hence keeping it in mind the data structure to store api execution results is changed to **blocking queue**. The LinkedBlockingDeque implementation is used to input api execution data at one end (Producer) and another thread pool consumer “CSVWriterConsumer” is created that will keep pulling out data from the other end of the queue and keep appending it to the local CSV file.

**The Load Balancer:**

It is observed the application throughput and mean response time was not enough to meet the end user needs. Hence an application level load balancer is introduced to route the traffic to 4 instances of the server. More than 100% improvement in the throughput and mean response time was observed. The single RDS instance is used and AWS Load balancer service is used to implement load balancer.