

ASSIGNMENT 4

(CS 6650 - Building Scalable Distributed Systems)

Rahul Pandey

Github:

<https://github.com/rahulpandeycs/bsds6650-Course-fall2020/tree/master/Assignment%204>

Server implementation:

<https://github.com/rahulpandeycs/bsds6650-Course-fall2020/tree/master/Assignment%204/CS6650-hw4-Server>

Load Testing (*Test runs and results for 256 and 512 clients*)

256 Threads, Single Server, Get requests

With Read Replica

```
[ec2-user@ip-172-31-89-98 bsds_assignment2]$ java -jar bsds-cs6650-hw2-clientPart.jar -f config.properties
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Reading file passed as input: config.properties
Entered into Job execution producer!
Entered into Consumer for writing file!
Number of failed requests: 0
Number of Successful requests: 388480
Mean responseTime:192 ms
Median responseTime:81 ms
The total run time (wall time) :202808 ms
Total Requests: 388480
Throughput: 1915 requests/sec
p99 (99th percentile) response time :1498 ms
Max response time:7525 ms
Exiting now: Data stored, Results are stored at: /usr/bsds_assignment2/performance_metrics.csv
Control came back to main
```

Without Read Replica, Assignment 3 results

```
[ec2-user@ip-172-31-89-98 bsds_assignment2]$ java -jar bsds-cs6650-hw2-clientPart.jar -f config.properties
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Reading file passed as input: config.properties
Entered into Job execution producer!
Entered into Consumer for writing file!
Number of failed requests: 0
Number of Successful requests: 388480
Mean responseTime:218 ms
Median responseTime:94 ms
The total run time (wall time) :228798 ms
Total Requests: 388480
Throughput: 1697 requests/sec
p99 (99th percentile) response time :1655 ms
Max response time:10152 ms
Exiting now: Data stored, Results are stored at: /usr/bsds_assignment2/performance_metrics.csv
Control came back to main
```

512 Threads, Single Server, Get requests

With Read Replica

```
[ec2-user@ip-172-31-89-98 bsds_assignment2]$ tail Metrics.txt
Number of failed requests: 0
Number of Successful requests: 776960
Mean responseTime:351 ms
Median responseTime:85 ms
The total run time (wall time) :370474 ms
Total Requests: 776960
Throughput: 2097 requests/sec
p99 (99th percentile) response time :3472
Max response time:14379 ms
```

Without Read replica:

```
[ec2-user@ip-172-31-89-98 bsds_assignment2]$ java -jar bsds-cs6650-hw2-ClientPart.jar -f config.properties
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Reading file passed as input: config.properties
Entered into Job execution producer!
Entered into Consumer for writing file!
Number of failed requests: 0
Number of Successful requests: 776960
Mean responseTime:368 ms
Median responseTime:93 ms
The total run time (wall time) :390065 ms
Total Requests: 776960
Throughput: 1991 requests/sec
p99 (99th percentile) response time :3491 ms
Max response time:17604 ms
Exiting now: Data stored, Results are stored at: /usr/bsds_assignment2/performance_metrics.csv
Control came back to main
```

Results Analysis

Comparison of assignments 2 and 3 with the results in assignment 4 for 256 & 512 clients.

256 Threads

Assignment 2:

```
[ec2-user@ip-172-31-89-98 bsds_assignment2]$ java -jar bsds-cs6650-hw2-clientPart.jar -f config.properties
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Reading file passed as input: config.properties
Entered into Job execution producer!
Entered into Consumer for writing file!
Number of failed requests: 0
Number of Successful requests: 388480
Mean responseTime:470 ms
Median responseTime:280 ms
The total run time (wall time) :494954 ms
Total Requests: 388480
Throughput: 784 requests/sec
p99 (99th percentile) response time :3160 ms
Max response time:19984 ms
Exiting now: Data stored, Results are stored at: /usr/bsds_assignment2/performance_metrics.csv
Control came back to main
```

Assignment 3:

```
[ec2-user@ip-172-31-89-98 bsds_assignment2]$ java -jar bsds-cs6650-hw2-clientPart.jar -f config.properties
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Reading file passed as input: config.properties
Entered into Job execution producer!
Entered into Consumer for writing file!
Number of failed requests: 0
Number of Successful requests: 388480
Mean responseTime:218 ms
Median responseTime:94 ms
The total run time (wall time) :228798 ms
Total Requests: 388480
Throughput: 1697 requests/sec
p99 (99th percentile) response time :1655 ms
Max response time:10152 ms
Exiting now: Data stored, Results are stored at: /usr/bsds_assignment2/performance_metrics.csv
Control came back to main
```

Assignment 4:

```
[ec2-user@ip-172-31-89-98 bsd_assignment2]$ java -jar bsd-cs6650-hw2-clientPart.jar -f config.properties
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Reading file passed as input: config.properties
Entered into Job execution producer!
Entered into Consumer for writing file!
Number of failed requests: 0
Number of Successful requests: 388480
Mean responseTime:192 ms
Median responseTime:81 ms
The total run time (wall time) :202808 ms
Total Requests: 388480
Throughput: 1915 requests/sec
p99 (99th percentile) response time :1498 ms
Max response time:7525 ms
Exiting now: Data stored, Results are stored at: /usr/bsd_assignment2/performance_metrics.csv
Control came back to main
```

512 Threads

Assignment 2:

```
[ec2-user@ip-172-31-89-98 bsd_assignment2]$ java -jar bsd-cs6650-hw2-clientPart.jar -f config.properties
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Reading file passed as input: config.properties
Entered into Job execution producer!
Entered into Consumer for writing file!
Number of failed requests: 0
Number of Successful requests: 776960
Mean responseTime:522 ms
Median responseTime:104 ms
The total run time (wall time) :568586 ms
Total Requests: 776960
Throughput: 1366 requests/sec
p99 (99th percentile) response time :10012 ms
Max response time:54015 ms
Exiting now: Data stored, Results are stored at: /usr/bsd_assignment2/performance_metrics.csv
Control came back to main
```


Assignment 3:

```
[ec2-user@ip-172-31-89-98 bsds_assignment2]$ java -jar bsds-cs6650-hw2-clientPart.jar -f config.properties
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Reading file passed as input: config.properties
Entered into Job execution producer!
Entered into Consumer for writing file!
Number of failed requests: 0
Number of Successful requests: 776960
Mean responseTime:368 ms
Median responseTime:93 ms
The total run time (wall time) :390065 ms
Total Requests: 776960
Throughput: 1991 requests/sec
p99 (99th percentile) response time :3491 ms
Max response time:17604 ms
Exiting now: Data stored, Results are stored at: /usr/bsds_assignment2/performance_metrics.csv
Control came back to main
```

Assignment 4:

```
[ec2-user@ip-172-31-89-98 bsds_assignment2]$ tail Metrics.txt
Number of failed requests: 0
Number of Successful requests: 776960
Mean responseTime:351 ms
Median responseTime:85 ms
The total run time (wall time) :370474 ms
Total Requests: 776960
Throughput: 2097 requests/sec
p99 (99th percentile) response time :3472
Max response time:14379 ms
```

Assignment 2:

The assignment iteratively builds on the server from Assignment 2. The assignment 2 builds the server logic and persistence layer. The server logic ensures consistency across all the read and write requests. As a result of this we see a much lesser throughput and mean-median response time. To improve this the number of database connections are increased, connection pool is created and we were able to achieve the throughput in the range of 700–800 requests/sec. (For 256 Threads input)

To improve the server responsiveness and throughput we scale out the application. We opted for a load balanced server and deployed the server over multiple instances, as a result we saw a significant improvement in the response time and the application throughput. (2000–2100 requests/sec) (For 256 Threads input)

```
[ec2-user@ip-172-31-89-98 bsds_assignment2]$ java -jar bsds-cs6650-hw2-clientPart.jar -f config.properties
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Reading file passed as input: config.properties
Entered into Job execution producer!
Entered into Consumer for writing file!
Number of failed requests: 0
Number of Successful requests: 388480
Mean responseTime:175 ms
Median responseTime:89 ms
The total run time (wall time) :192095 ms
Total Requests: 388480
Throughput: 2022 requests/sec
p99 (99th percentile) response time :1183 ms
Max response time:11551 ms
Exiting now: Data stored, Results are stored at: /usr/bsds_assignment2/performance_metrics.csv
Control came back to main
```

Assignment 3:

As the immediate consistency blocks the application till we have successfully written to the database. To remove this bottleneck we advance our server logic to have eventual consistency. In Assignment 3 we build on the top of this idea and introduce RabbitMq as the intermediate queue storage. We emulated a producer-consumer scenario where the write requests are written to the queue with details, later this data is pulled out from the queue using multiple consumers who are subscribed to any updates in the queue and write this data to the persistence layer i.e RDS eventually.

As a result of this we saw a significant increase in the response rate and the application throughput. After carefully analyzing the application at runtime we realized that there are several duplicate GET requests that are being fetched from the database again and again. We changed the server design to cache such requests and next time we see such requests serve it from the cache. The redis server is leveraged to cache such requests and next time we encounter such a request it is served from redis cache. The cache is updated

on the write requests for the same data.

As a result of eventual consistency and serving requests from the cache we were able to achieve a throughput of 1600-1700 requests/sec. (For 256 Threads input)

```
[ec2-user@ip-172-31-89-98 bsds_assignment2]$ java -jar bsds-cs6650-hw2-clientPart.jar -f config.properties
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Reading file passed as input: config.properties
Entered into Job execution producer!
Entered into Consumer for writing file!
Number of failed requests: 0
Number of Successful requests: 388480
Mean responseTime:218 ms
Median responseTime:94 ms
The total run time (wall time) :228798 ms
Total Requests: 388480
Throughput: 1697 requests/sec
p99 (99th percentile) response time :1655 ms
Max response time:10152 ms
Exiting now: Data stored, Results are stored at: /usr/bsds_assignment2/performance_metrics.csv
Control came back to main
```


Assignment 4:






Till now all the requests were being served from a single database server and this is a bottleneck in the scalability of the system. Though our application is write heavy, still the throughput and response rate of the server is affected by the GET requests that are executed in parallel with the write requests. If we can partition our requests such that GET requests are served from different servers and write can be done to different servers we can get rid of this bottleneck. Based on this idea we refactored our server design to route the GET requests to the READ replica, the read replica will be in sync with any writes made to the primary database.

As a result we were able to achieve a throughput between 2000-2100 requests/sec (For 512 Threads input) and a median and mean 81 ms and 192 ms. This is equivalent to the performance metrics obtained when we had the server load balanced with 4 instances. This is indeed a great improvement.

```
[ec2-user@ip-172-31-89-98 bsds_assignment2]$ java -jar bsds-cs6650-hw2-clientPart.jar -f config.properties
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Reading file passed as input: config.properties
Entered into Job execution producer!
Entered into Consumer for writing file!
Number of failed requests: 0
Number of Successful requests: 388480
Mean responseTime:192 ms
Median responseTime:81 ms
The total run time (wall time) :202808 ms
Total Requests: 388480
Throughput: 1915 requests/sec
p99 (99th percentile) response time :1498 ms
Max response time:7525 ms
Exiting now: Data stored, Results are stored at: /usr/bsds_assignment2/performance_metrics.csv
Control came back to main
```

The Database Design

Indexes in Table			
Key	Type	Unique	Columns
 PRIMARY	BTREE	YES	dayID, resortID, skierID, time

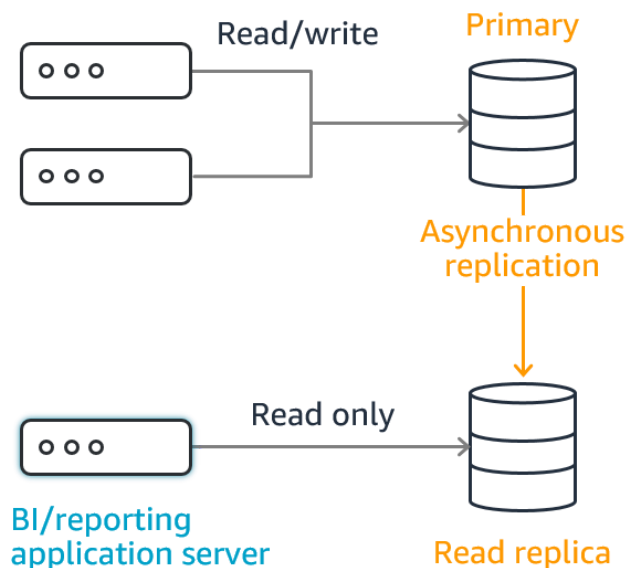
Columns in table			
Column	Type	Nullable	Indexes
 dayID	int(11)	NO	PRIMARY
 resortID	varchar(255)	NO	PRIMARY
 skierID	int(11)	NO	PRIMARY
 time	int(11)	NO	PRIMARY
 liftID	int(11)	YES	

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.

Read replica design:

It is observed that the single server instance is overloaded while addressing both read and write requests to the database. Hence there is the need to distribute the load. There are several ways to improve the reading performance. You can use this pattern to improve the overall performance through distributing the reading into multiple "Read Replicas" (that is, replicas for reading). A Read Replica follows the writing to a master, reflecting the data to itself. You can reduce the load on the master through using primarily Read Replicas for reading.

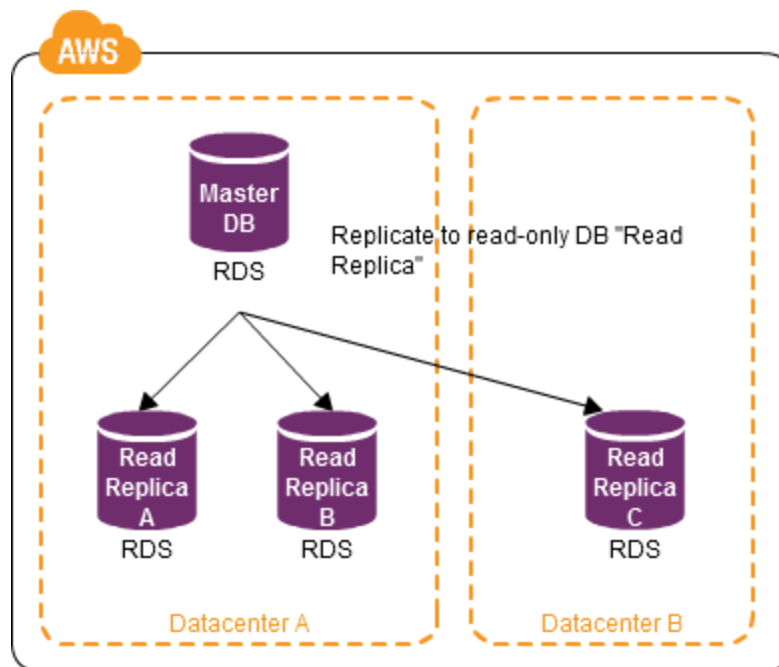
Application servers Database server



The same has been adapted for this assignment and AWS RDS has been used as the database. The AWS Relational Database Management Systems (RDBMS) service, the Amazon Relational Database Service (RDS), has a function enabling the easy creation of a read-only database known as a "Read Replica." An EC2 instance can also be used instead to create a read-only database.

We Create a read-only replica of the master database. For a database handled by RDS, we create the read-only replica using the Read Replica function.

When data is read by an application, set the access destination to the Read Replica.



Once a read replica of the RDS instance is created all the read requests are routed to the new read replica server. The AWS RDS service takes care of the replication to read replicas from master nodes where writes are done.

Amazon RDS Read Replicas provide enhanced performance and durability for RDS database (DB) instances. They make it easy to elastically scale out beyond the capacity constraints of a single DB instance for read-heavy database workloads. We can create one or more replicas of a given source DB Instance and serve high-volume application read traffic from multiple copies of your data, thereby increasing aggregate read throughput.

UML diagrams: (Also included with detailed section below)

Server:

<https://github.com/rahulpandeycs/bsds6650-Course-fall2020/tree/master/Assignment%204/CS6650-hw4-Server/UML>

Client:

<https://github.com/rahulpandeycs/bsds6650-Course-fall2020/tree/master/Assignment%204/bsds-cs6650-hw4-clientPart/UML>

Running the application:

The application design is divided into 5 Parts:

- The Server
 - Eventual Consistency to Database using RabbitMQ and Subscribed consumer
 - The server cache using Redis
- 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

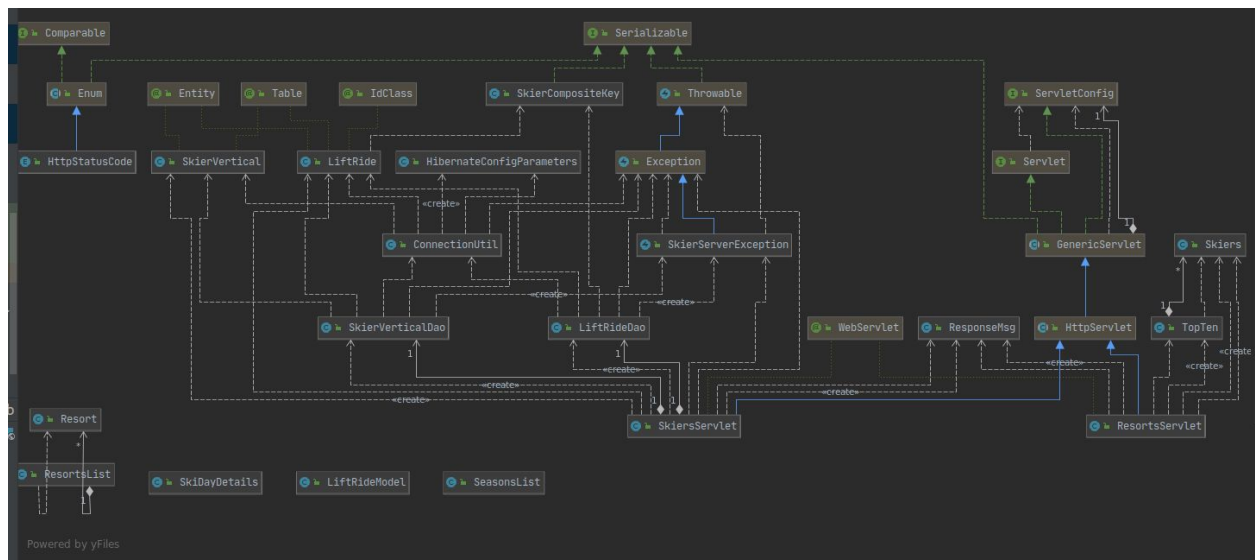
#Local

cmd.addressPort=<http://bsdscs6650-1923097914.us-east-1.elb.amazonaws.com:8080/CS6650-hw4-Server-deploy/>

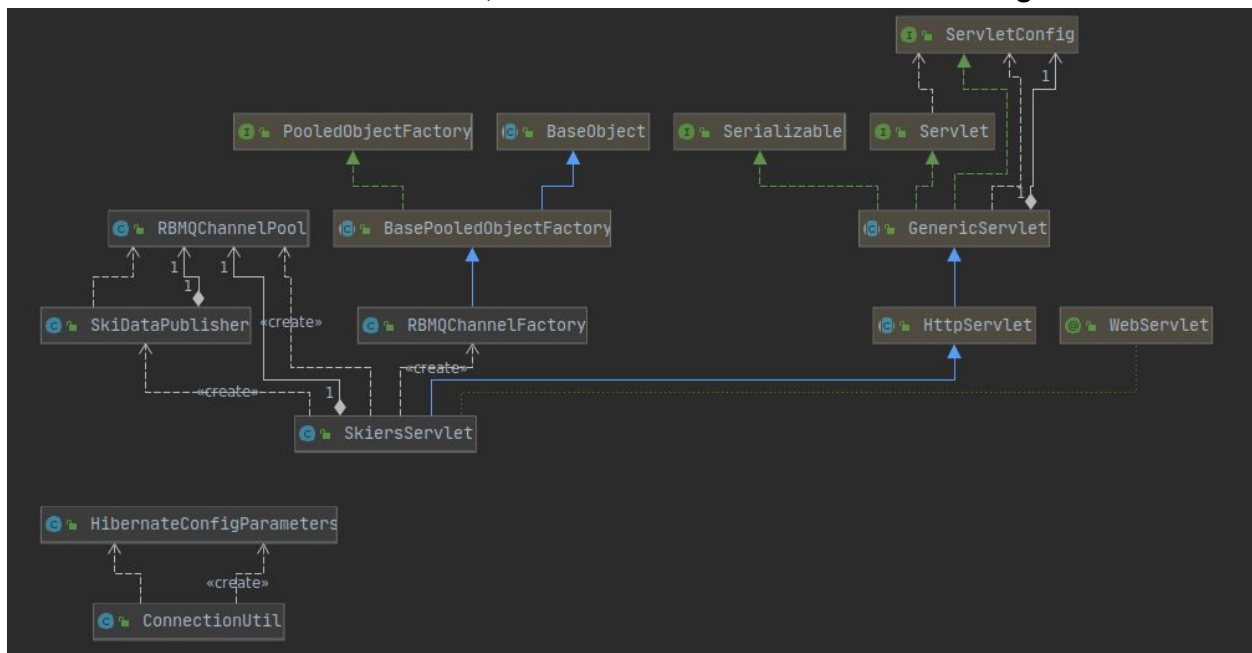
Note: If no `config.properties` is provided it reads default `config.properties`

The Server design description

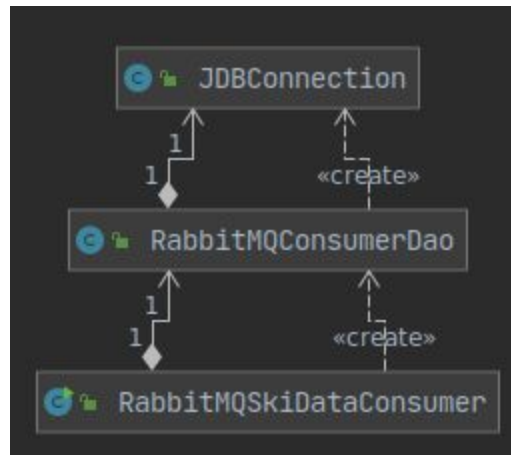
The Server Class diagram



The ConnectionPool for RabbitMQ, and Publisher to RabbitMQ Class diagram:



The Consumer Class diagram:



The server exposes below API using Java Servlets:

Skiers

POST/skiers/liftrides

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

GET/skiers/{skierID}/vertical

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.

Eventual Consistency to Database using RabbitMQ and Subscribed consumer

To get rid of Database locks and improve responsiveness of the application, the data is chose to eventually persist. The RabbitMQ queue is used to temporarily persist the data and later many consumers are employed to pull data from this queue and persist it to the RDS (MySQL database).

Data publish to RabbitMQ:

The POST calls to skiers api's are redirected to the publisher that is called per request to store the data into RabbitMQ server instead of saving this to the database. The publiser looks something like below code, which first acquires the channel and publishes the specified bytes to the queue.

```
// channel per thread  
Channel channel = channelPool.getChannel();  
  
//Second parameter to make queue durable, Changing it to non durable for testing  
channel.queueDeclare(QUEUE_NAME, false, false, false, null);
```

```
//Serialize the data
byte[] yourBytes = SerializationUtils.serialize(liftRide);

//Publish converted bytes on the queue
channel.basicPublish("", QUEUE_NAME, null, yourBytes);

//Return channel to the channel pool to be used by other requests
channelPool.returnChannel(channel);
```

Connection Pooling for RabbitMQ Channels:

To process the input requests faster and improve the response rate, a pool of channels is created in the server's `init()` method and each time the publisher requests for a channel, it is served from the pool.

The channel pool *RBMQChannelFactory* is created by extending *BasePooledObjectFactory<Channel>*

Channel Pool class:

<https://github.com/rahulpandeycs/bsds6650-Course-fall2020/blob/master/Assignment%204/CS6650-hw4-Server/src/main/java/RabbitMQConnectionPool/RBMQChannelPool.java>

Factory channel pool class:

<https://github.com/rahulpandeycs/bsds6650-Course-fall2020/blob/master/Assignment%204/CS6650-hw4-Server/src/main/java/RabbitMQConnectionPool/RBMQChannelFactory.java>

The channel pool is configured as below:

```
// Declare channel pool
private RBMQChannelPool rbmqChannelPool;
public static GenericObjectPoolConfig defaultConfig;

// Configuration
static {
    defaultConfig = new GenericObjectPoolConfig();
    defaultConfig.setMaxTotal(200);
    defaultConfig.setMinIdle(16);
    defaultConfig.setMaxIdle(200);
    defaultConfig.setBlockWhenExhausted(false);
}

// Establishing connection to RabbitMQ
ConnectionFactory factory = new ConnectionFactory();

factory.setUsername("*****");
factory.setPassword("*****");
```

```

factory.setVirtualHost("/");
factory.setHost("ec2-**-**-132-**.compute-1.amazonaws.com");
factory.setPort(5672);

//Getting the connection
connection = factory.newConnection();

//Getting channel pool
// A generic pool of channels is created using Apache Simple Pool
rbmqChannelPool = new RBMQChannelPool(new GenericObjectPool<Channel>(new
RBMQChannelFactory(connection), defaultConfig));

```

The pool of connections to RabbitMQ was also tried but it was observed as not very effective. Also the number of pooled RabbitMQ channels are currently number at 200, considering the number of threads accessing the tomcat server.

Subscribed Consumer to RabbitMQ:

The queue size is ever growing during the client access, hence the queue data needs to be pulled out quickly at a faster rate otherwise the server will run out of memory.

Initially I started with 1 consumer but realized the consumer take a lot of time to pull the data from the queue and persist it in the database. I tried different variations and increased it gradually.

Right now my consumer implementation uses 20 threads i.e 20 consumers to pull the data and persist it in the single RDS instance.

I tried increasing this number but when the application is load balanced with 4 servers, each having the pool of connections themselves the Single RDS instance run out of connections and i get “Too Many connections error”. Hence to keep it within safe bound’s i would 20 consumer’s is a good number. But this can definitely increase based on the number of RDS instances and number of application/Publisher servers connecting to RDS.

The consumer is fault tolerant and does 3 retries with appropriate wait time in case of failures, ex:

```
//Retry the save to DB in case of save failure. retrySaveToDB(liftRide, 1);
```

The consumer implementaion looks like below:

```

final Channel channel = rbmqConnectionUtil.getChannel();
channel.queueDeclare(QUEUE_NAME, false, false, false, null);

// max one message per receiver
channel.basicQos(1); DeliverCallback deliverCallback = (consumerTag, delivery) -> {

```

```

//output from queue is deserialized into object
LiftRide liftRide = SerializationUtils.deserialize(delivery.getBody());
channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
try {
    //Persist data to database, MYSQL
    liftRideDao.updateLiftRide(liftRide);
} catch (SQLException throwables) {

    //Retry the save to DB in case of save failure.
    retrySaveToDB(liftRide, 1);
}
}; // process messages
channel.basicConsume(QUEUE_NAME, false, deliverCallback, consumerTag -> {});

```

Handling read requests: Server Cache using Redis

It has been observed that the performance results obtained with server with eventual consistency was not so great as it still need to serve a lot of GET requests from the database. Many of these GET requests are similar and can be served from memory if kept stored.

Hence, I leveraged Redis server for caching my read results and invalidating them during an update. The jedis is used to interact with the redis server running on a different EC2 instance. After the introduction of the Cache, the load on the database decreased, thus decreasing the time to get eventual consistency as now server will not have to take unnecessary load to serve duplicate GET requests.

On every update call it is redis is checked against two keys, representing our two GET request to get Skier vertical, if any of these exists in the cache the cache is updated with new totalVert + current value. Another approach to delete such cache was tried but considering performance implications, the former is better.

If the key doesn't exist in the cache the read request is served from the database.

Jedis Configuration:

```

//Setting up jedis pool
final static JedisPoolConfig poolConfig = buildPoolConfig();

```



```
static JedisPool jedisPool;
```

```
//Pool setup:
```

```
private static JedisPoolConfig buildPoolConfig() {  
    final JedisPoolConfig poolConfig = new JedisPoolConfig();  
    poolConfig.setMaxTotal(128);  
    poolConfig.setMaxIdle(128);  
    poolConfig.setMinIdle(16);  
    poolConfig.setTestOnBorrow(true);  
    poolConfig.setTestOnReturn(true);  
    poolConfig.setTestWhileIdle(true);  
    poolConfig.setMinEvictableIdleTimeMillis(Duration.ofSeconds(60).toMillis());  
    poolConfig.setTimeBetweenEvictionRunsMillis(Duration.ofSeconds(30).toMillis());  
    poolConfig.setNumTestsPerEvictionRun(3);  
    poolConfig.setBlockWhenExhausted(true);  
    return poolConfig;  
}
```

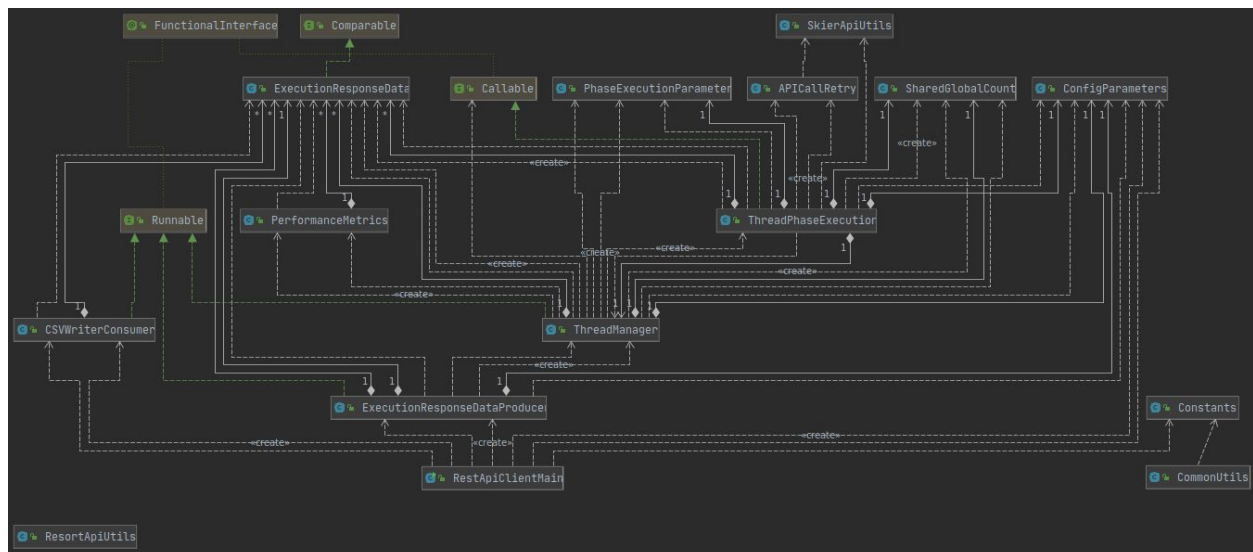
```
//instantiation the jedis pool connection
```

```
jedisPool = new JedisPool(poolConfig, "localhost", 6379, 4000);'
```

```
//using the connection to get resource to either retrieve or update the cache.
```

```
try (Jedis jedis = jedisPool.getResource()) {}
```

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 *ThreadPhaseExecution* 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:

```
[ec2-user@ip-172-31-89-98 bsds_assignment2]$ java -jar bsds-cs6650-hw2-clientPart.jar -f config.properties
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Reading file passed as input: config.properties
Entered into Job execution producer!
Entered into Consumer for writing file!
Number of failed requests: 0
Number of Successful requests: 388480
Mean responseTime:175 ms
Median responseTime:89 ms
The total run time (wall time) :192095 ms
Total Requests: 388480
Throughput: 2022 requests/sec
p99 (99th percentile) response time :1183 ms
Max response time:11551 ms
Exiting now: Data stored, Results are stored at: /usr/bsds_assignment2/performance_metrics.csv
Control came back to main
```

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 countDownLatch) 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 means 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 the AWS Load balancer service is used to implement the load balancer.