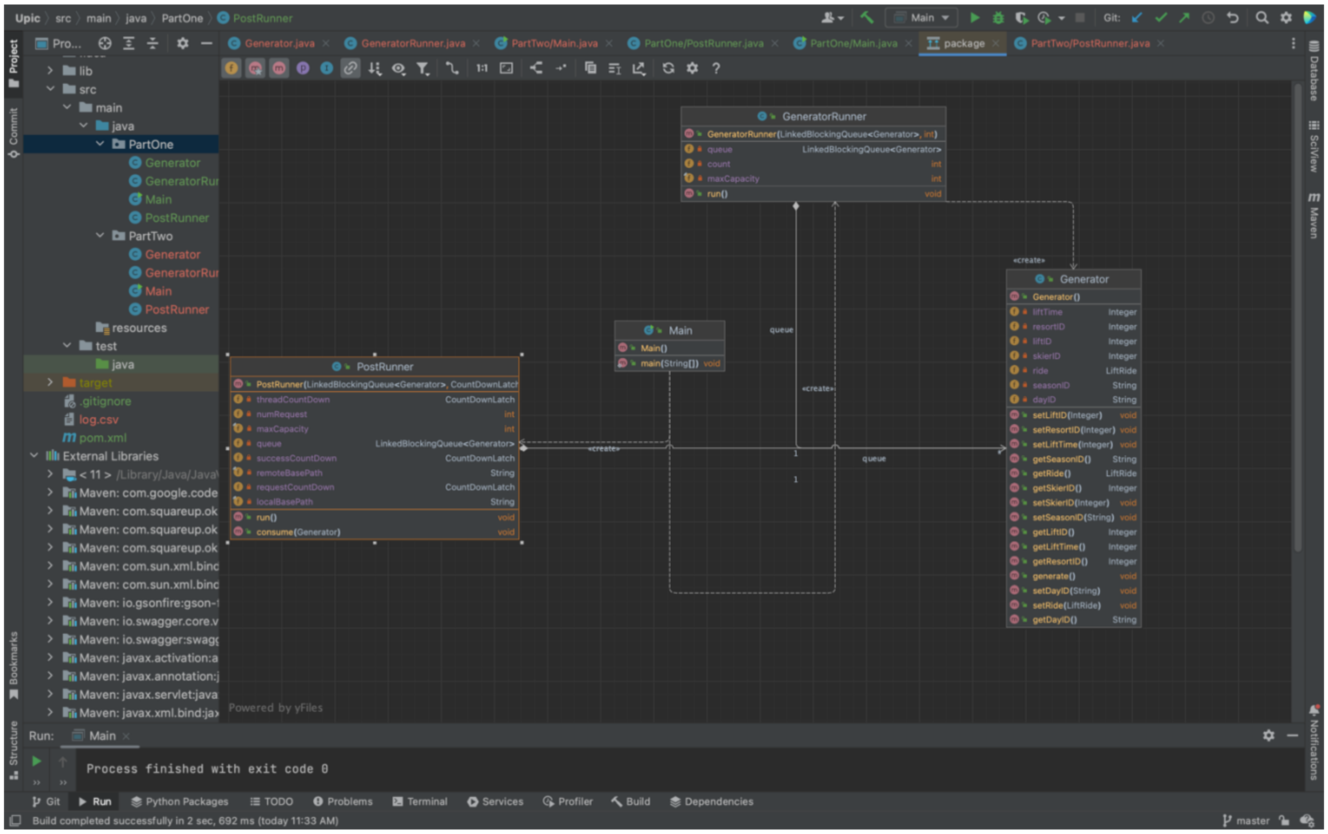
1. URL for repo:
   https://github.com/edellhou/CS6650/tree/main/Assignment

2. Client Design:
   The project is multithreaded and thread design is based on producer and consumer model. Producer is responsible for generating random lift ride event and send to the queue, and consumer here is responsible take event from queue and sends the requests to server.

   Below is the UML diagram for the project. There are four classes, Main, GeneratorRunner, Generator and PostRunner.



```
Process finished with exit code 0
```

Build completed successfully in 2 sec, 692 ms (today 11:33 AM)

GeneratorRunner is the producer and PostRunner is the consumer. They both implement runnable class. Generator is the helper class that generate the random event of lift ride that include many different parameters, such as time, lift ID, seasonID .. etc.

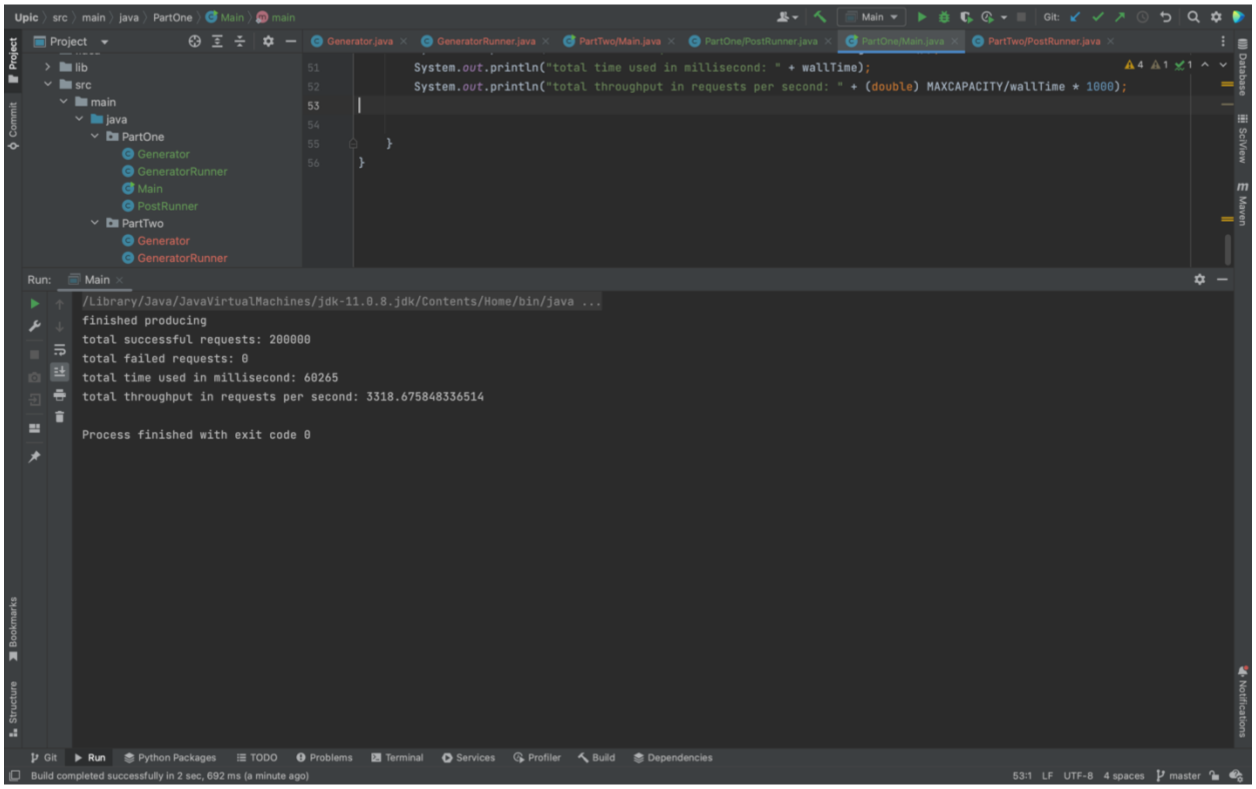
The project used Api from swagger package.

In the main class, you can see, I have a one thread allocated for producer to produce 200K lift ride event and pushed to linked Blocking queue. One the consumer side, things are a little complex. First I started off with 32 threads, and once any thread finished, the rest 168 threads can start working and send request to server. Each thread sends 1000 requests. To efficiently manage creating and efficient use of thread, I created fixed size thread pool of 168.

The project is designed with thread safety, efficiency, and simplicity in mind. It uses thread safe collections, and thread pool for efficiency, and it only has 4 classes with encapsulation. It is also a reusable code that can be changed easily when number of requests or threads need to be changed.

Little's Law prediction:
A single thread test is run which sent 1000 request. The total response time is 25270 millisecond. Response time per second per request is 0.025270.
The project has max 168 thread, however, when the program first started, it only runs 32 threads, so can be 32 threads finished at the same time and proceed with 168 thread. What I can expect is only estimate of throughput which (32/ 0.025270 + 168/0.025270) / 2 = 3957.26, because I cannot predict how the scheduler decide to run the initial thread, it could be all 32 thread finished at about the same time, then continue with 168 threads, or it can just finished one of 32 thread really quick, and immediately use the full capacity. You can see my actual throughput is in the range of my prediction.

3. This is the stats for part one. A fixed size thread pool with 168 threads is created. It starts with 32 threads, and once any thread finished, then it can continue to create rest 168 threads, each send 1000 requests



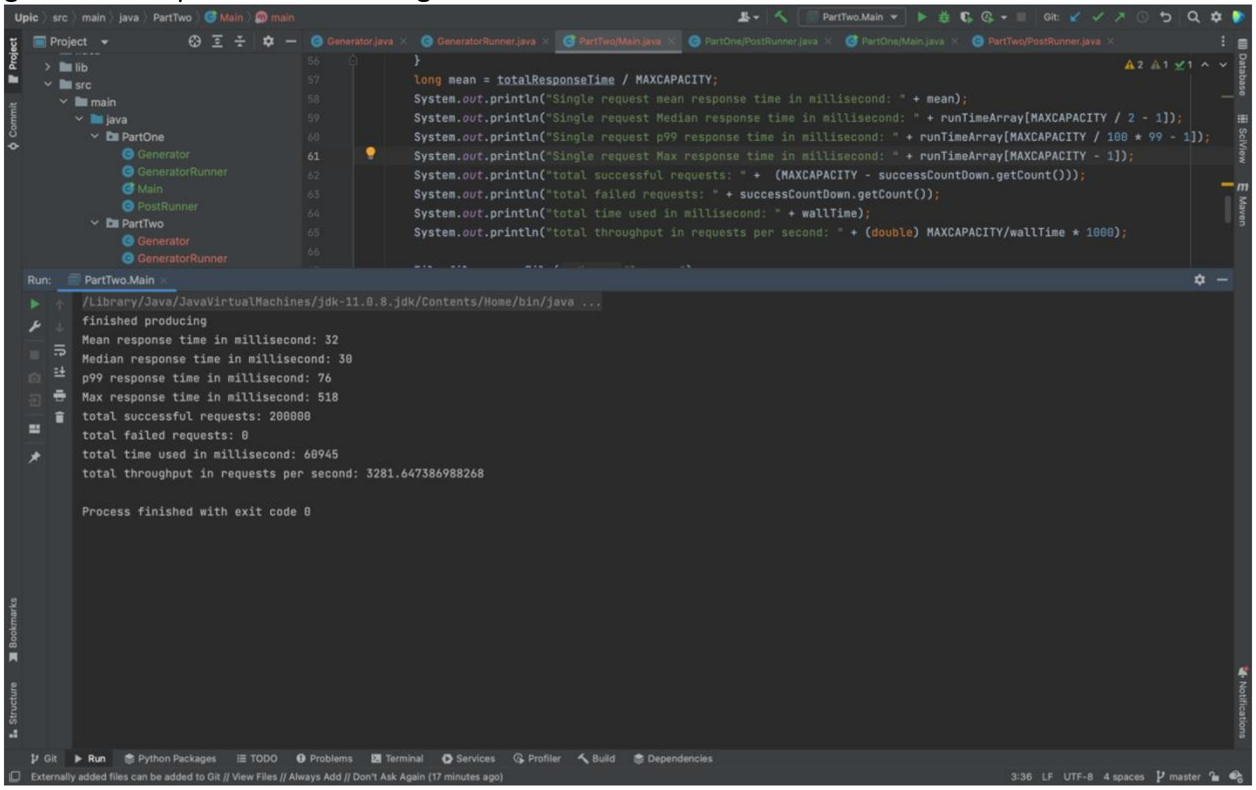
```java
        System.out.println("total time used in millisecond: " + wallTime);
        System.out.println("total throughput in requests per second: " + (double) MAXCAPACITY/wallTime * 1000);


    }
}
```

```
/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin/java ...
finished producing
total successful requests: 200000
total failed requests: 0
total time used in millisecond: 60265
total throughput in requests per second: 3318.675848336514

Process finished with exit code 0
```

4. Stats for part two. Throughput of part two is with 5% of part one. A csv file is also generated in part two named log.csv



```java
        }
        long mean = totalResponseTime / MAXCAPACITY;
        System.out.println("Single request mean response time in millisecond: " + mean);
        System.out.println("Single request Median response time in millisecond: " + runTimeArray[MAXCAPACITY / 2 - 1]);
        System.out.println("Single request p99 response time in millisecond: " + runTimeArray[MAXCAPACITY / 100 * 99 - 1]);
        System.out.println("Single request Max response time in millisecond: " + runTimeArray[MAXCAPACITY - 1]);
        System.out.println("total successful requests: " +  (MAXCAPACITY - successCountDown.getCount()));
        System.out.println("total failed requests: " + successCountDown.getCount());
        System.out.println("total time used in millisecond: " + wallTime);
        System.out.println("total throughput in requests per second: " + (double) MAXCAPACITY/wallTime * 1000);
```

```
/Library/Java/JavaVirtualMachines/jdk-11.0.8.jdk/Contents/Home/bin/java ...
finished producing
Mean response time in millisecond: 32
Median response time in millisecond: 30
p99 response time in millisecond: 76
Max response time in millisecond: 518
total successful requests: 200000
total failed requests: 0
total time used in millisecond: 60945
total throughput in requests per second: 3281.647386988268

Process finished with exit code 0
```

Bonus Point: I built the spring server, and tested it with postman both locally and on EC2, however, when running the client side with spring, there is a api exception from swagger package, I don’t have time left to figure out the cause