UNIVERSITÉ Concordia UNIVERSITY

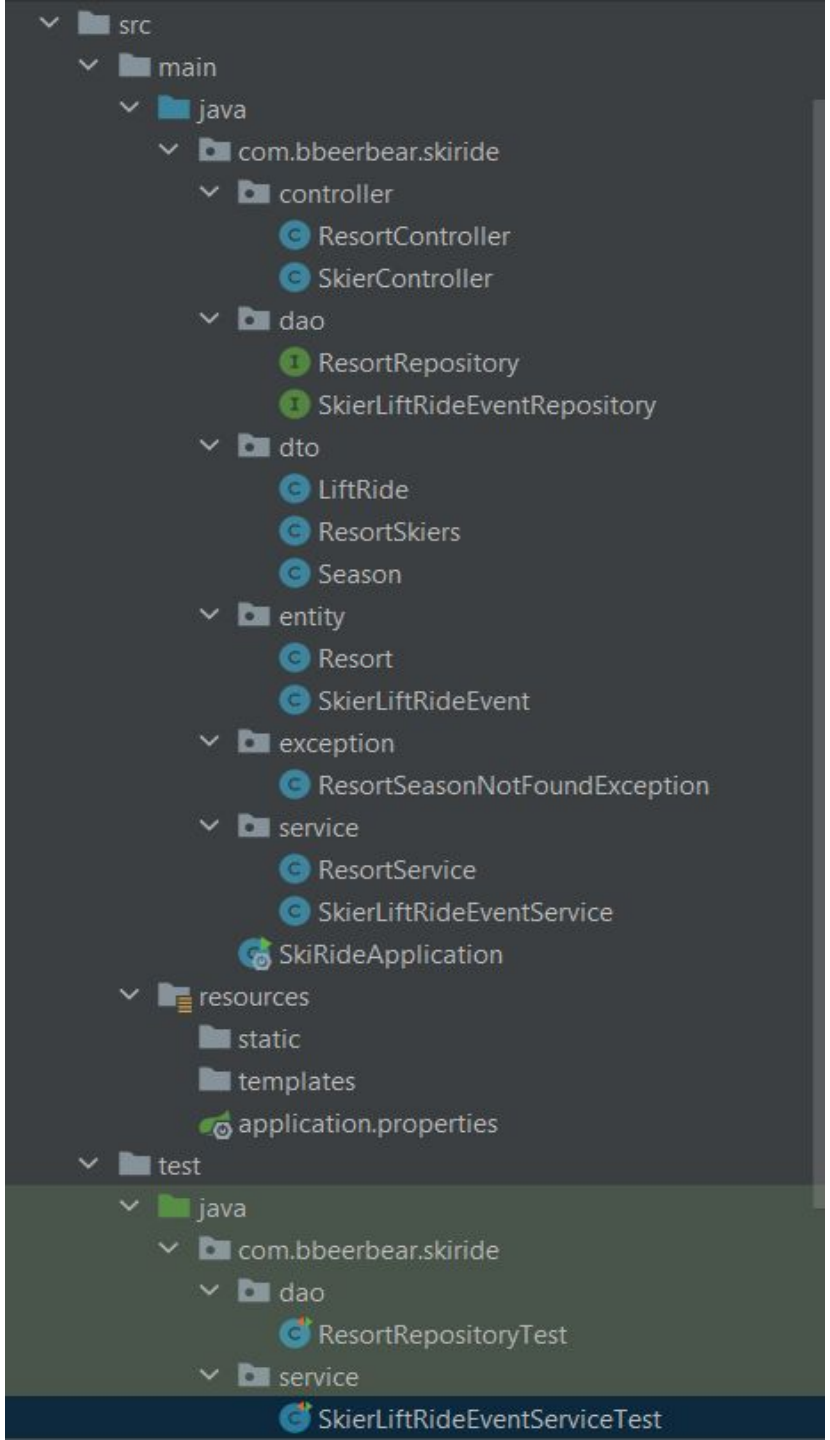
Digitalizing the Ski Sport Industry with a Cloud-Based Distributed System

Team member: Limin Xiong
Pranav Jha
Sahibmeet Singh
Vishruth Khatri

Presenter: Limin Xiong, Pranav Jha, Sahibmeet, Vishruth

Technology

- Server
  - Spring Boot
  - REST API
  - DTO
  - MongoDB
  - Oracle Cloud Deployment
- Client
  - Thread Concurrency



Server-side

Spring Data MongoDB

- MongoDB Template and MongoDB Repository
- Provides predefined methods

Spring Web Module

- Handling HTTP requests and responses, handling exceptions

Spring Test Module

- Provides a set of easy-to-use annotations and utility classes

Deployment

- On Oracle Cloud

Realize REST API and Save data to MongoDB

```
spring.data.mongodb.uri=mongodb+srv://bigbea                cluster1.pvadida.mongodb.net/?retryWrites=true&w=majority
spring.data.mongodb.database=skier_rider
```

```
3 usages  Limin
public interface SkierLiftRideEventRepository extends MongoRepository<SkierLiftRideEvent, String> {
}
```

```
@RestController
public class SkierController {
    2 usages
    private final SkierLiftRideEventRepository skierLiftRideEventRepository;
    2 usages
    private final SkierLiftRideEventService skierLiftRideEventService;
    no usages  Limin
    public SkierController(SkierLiftRideEventRepository skierLiftRideEventRepository, SkierLiftRideEventService skierLift
        this.skierLiftRideEventRepository = skierLiftRideEventRepository;
        this.skierLiftRideEventService = skierLiftRideEventService;
    }

    // Write a new lift ride for the skier
    no usages  Limin
    @PostMapping("/skiers/{resortID}/seasons/{seasonID}/days/{dayID}/skiers/{skierID}")
    public void createSkierLiftRide(@PathVariable int resortID, @PathVariable String seasonID,
                                     @PathVariable String dayID, @PathVariable int skierID,
                                     @RequestBody LiftRide liftRide){
        LiftRide liftRide1 = new LiftRide(liftRide.getLiftId(), liftRide.getTime());
        SkierLiftRideEvent skierLiftRideEvent = new SkierLiftRideEvent( id: null, resortID, seasonID, dayID, skierID, liftR
        skierLiftRideEventRepository.save(skierLiftRideEvent);
    }
```

Lift Ride Event Generator Thread

- **Store** the generating lift ride event in the **Blocking Queue** (thread-safe)
- Run the Thread **separately** and the client take the data from the queue.

```
// Create a blocking queue for lift ride events
2 usages
private final BlockingQueue<LiftRideEvent> queue = new LinkedBlockingQueue<>();
```

```
// Start the lift ride event generator thread
LiftRideEventGeneratorThread generatorThread = new LiftRideEventGeneratorThread(queue,num_of_thread,
        num_of_requests_each_thread);
generatorThread.start();
```

```
LiftRideEvent liftRideEvent = queue.take();
```

Thread Concurrency

- Run threads with ExecutorService

```
for(int i = 0; i < num_of_thread; i++) {
    executorService.submit(()->{
        for(int j = 0; j < num_of_requests_each_thread; j++) {
            try {
                long requestStartTime = System.currentTimeMillis();
                LiftRideEvent liftRideEvent = queue.take();
                String url = "http://localhost:8080/coen6731/skiers/" + Integer.toString(liftRideEvent.getResortID()) +
                        liftRideEvent.getSeasonID() + "/days/" + liftRideEvent.getDayID() + "/skiers/" + Integer.toStrin
                String url = "http://155.248.230.86:8080/skiers/" + Integer.toString(liftRideEvent.getResortId()) + "/se
                        liftRideEvent.getSeasonId() + "/days/" + liftRideEvent.getDayId() + "/skiers/" + Integer.toStrin
                String requestBody = new Gson().toJson(liftRideEvent.getLiftRide());

                HttpRequest request = HttpRequest.newBuilder()
                          .header( name: "Content-Type",  value: "application/json")
                          .uri(URI.create(url))
                          .POST(HttpRequest.BodyPublishers.ofString(requestBody))
                          .build();

                HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
                System.out.println(System.currentTimeMillis() - requestStartTime);
                eachRequestTimes.add(System.currentTimeMillis() - requestStartTime);

                int retries = 0;
                // Retry up to MAX_RETRIES times for 4XX and 5XX response codes
                while (response.statusCode() >= 400 && retries < MAX_RETRIES) {
                    retries++;
                    System.out.println("Request failed with status code " + response.statusCode() + ", retrying (attempt
                    response = client.send(request, HttpResponse.BodyHandlers.ofString());
                }
```

Terminate all threads

- Shutdown the ExecutorService gracefully
  - executorService.shutdown();

- Wait for all threads to complete
  - executorService.awaitTermination(10, TimeUnit.MINUTES);

```
try {
    generatorThread.join();
    executorService.shutdown();
    executorService.awaitTermination( timeout: 10, TimeUnit.MINUTES);
} catch (InterruptedException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
}
```

Client 1

```
long endTime = System
long wallTime = endTi
int successfulRequest
int unsuccessfulReque
double throughput = (

int totalRequestTime
for(long time: eachRe
    totalRequestTime
}

System.out.println("N
        + num_of_requ
// expected throughpu
double w_averageTimeE
double λ_expectedThro

System.out.println("E
System.out.println("Number of successful req
System.out.println("Number of unsuccessful r
System.out.println("Total run time: " + wall
System.out.println("Total throughput: " + th
System.out.println("According to Little's Law, expected throughput is: " + λ_expectedThroughput + " requests/second");
```

```
You have connectivity to call the API...

Number or threads us
Each request latenc
Number of successful
Number of unsuccessf
Total run time: 486
Total throughput: 6
According to Little

You have connectivit

Number or threads us
Each request latenc
Number of successful
Number of unsuccessf
Total run time: 100
Total throughput: 9
According to Little
```

```
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Exception in thread: HTTP/1.1 header parser received no bytes
Number or threads used: 1000, number of requests each thread: 10
Each request latency: 9389.11001905526ms
Number of successful requests sent: 9971
Number of unsuccessful requests: 29
Total run time: 99881 ms
Total throughput: 100.11914177871667 requests/second
According to Little's Law, expected throughput is: 106.5063672670246 requests/second
```

Client2 - Output - Profiling Performance

```
double meanResponseTime = (double) totalResponseTime / eachRequestTimes.size();
double medianResponseTime = MedianCalculator.calculateMedian(eachRequestTimes);

long endTime = System.currentTimeMillis();
long wallTime = endTime - startTime;
double throughput = (double) num_of_thread * num_of_requests_each_thread / (wallTime / 1000.0);

long p99ResponseTime = P99Calculator.calculateP99(eachRequestTimes);

long minResponseTime = Collections.min(eachRequestTimes);
long maxResponseTime = Collections.max(eachRequestTimes);

// expected throughput λ = L / W
double w_averageTimeEachRequest = (double) tota
double λ_expectedThroughput = num_of_thread / (

System.out.println("Number or threads used: " +
        "number of requests each thread: " + nu
System.out.println("Mean response time: " + mea
System.out.println("Median response time: " + m
System.out.println("Throughput: " + throughput + " requests/second");
System.out.println("According to Little's Law, expected throughput is: " + λ_expectedThroughput + " requests/second");
System.out.println("P99 response time: " + p99ResponseTime + "ms");
System.out.println("Min response time: " + minResponseTime + "ms");
System.out.println("Max response time: " + maxResponseTime + "ms");
```

```
Mean response time: 994.8844ms
Median response time: 1001.0ms
Throughput: 99.91207737191273 requests/second
According to Little's Law, expected throughput is: 100.51419039237122 requests/se
P99 response time: 1076ms
Min response time: 120ms
Max response time: 1231ms
```

Little's Law

- According to Little's Law: **L = λW**
- λ = throughput
- L = total number of requests
- W = average time of each request
- In 200 threads, each with 50 requests,
  - The total throughput is **99.75360858679063** requests/second
  - Expected throughput is **101.365335315769** requests/second

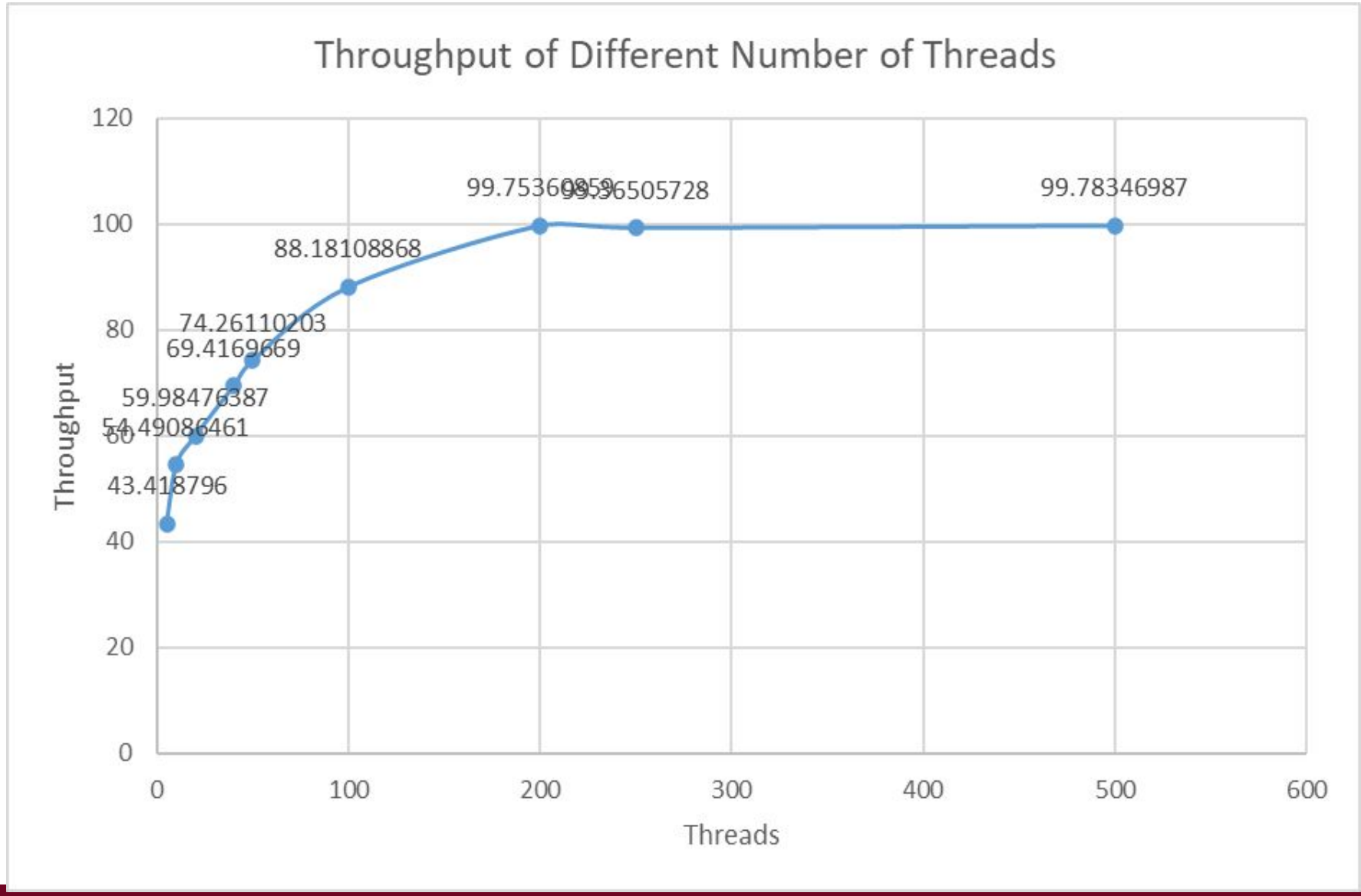
```
Number or threads used: 200, number of requests each thread: 50
Each request latency: 1973.0611ms
Number of successful requests sent: 10000
Number of unsuccessful requests: 0
Total run time: 100247 ms
Total throughput: 99.75360858679063 requests/second
According to Little's Law, expected throughput is: 101.365335315769 requests/second
```

Maximum of the throughput(10k requests)

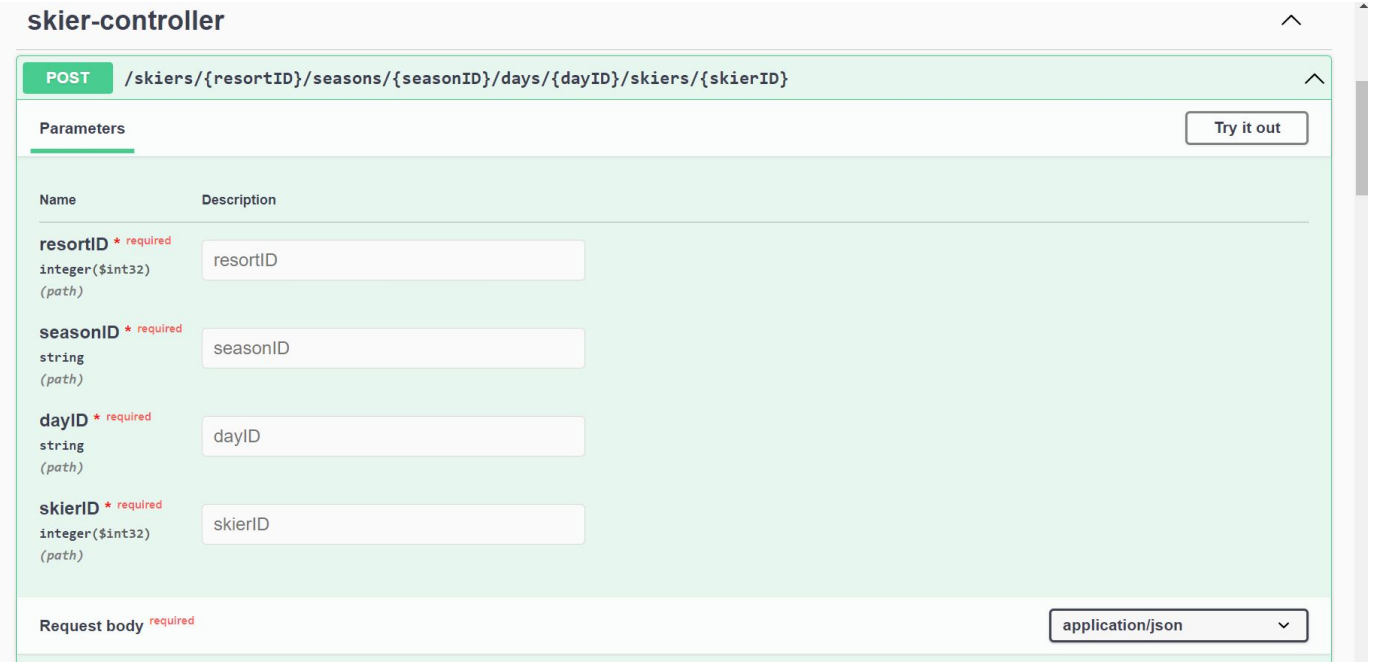
Testing the throughput in different number of threads

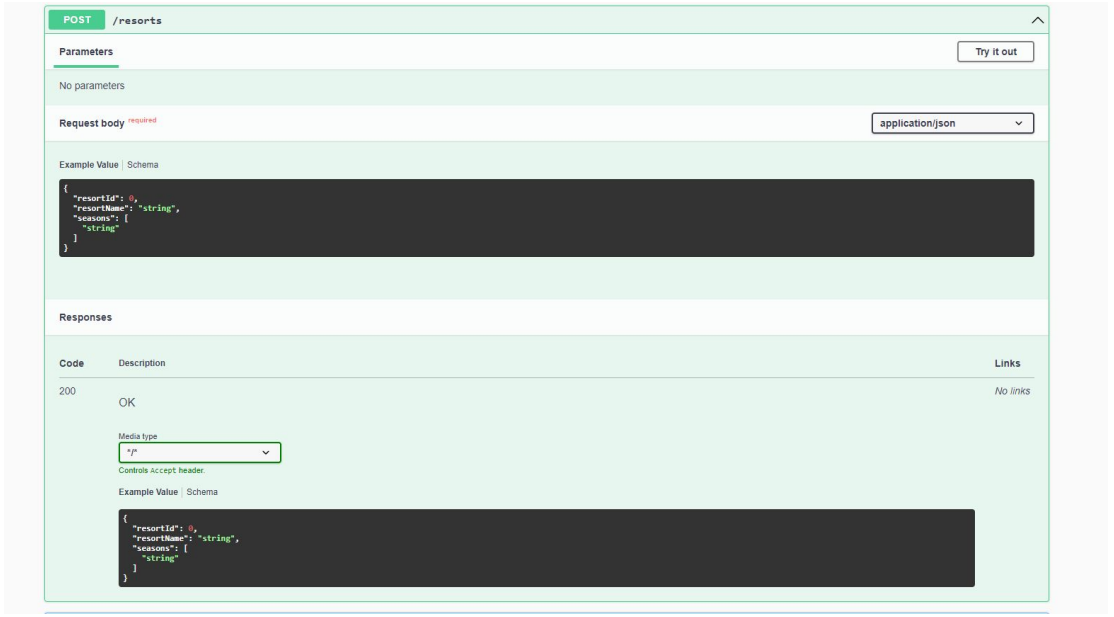
Threads	Throughput(requests/s)
5	43.418796
10	54.49086461
20	59.98476387
40	69.4169669
50	74.26110203
100	88.18108868
200	99.75360859
250	99.36505728
500	99.78346987

Because of the thread concurrency overhead, the **throughput** is **bounded** by the capacity



API Swaggerhub





Link for the Swaggerhub API -:
http://155.248.230.86:8080/swagger-ui/index.html#/