Project Description

This project aims to develop a distributed system that enable digitalization of a sport industry. Assume ski resorts use RFID lift ticket readers so that every time a skiier gets on a ski lift, the time of the ride and the skier ID are recorded. Suppose that geographically distributed resorts adopt the same technology and lift rides from all the participating resorts can be collected. This data can then be used as a basis for data analysis, answering such questions as:

- which lifts are most heavily used?
- which skiers ride the most lifts?
- How many lifts do skiers ride on average per day at resort X?

In stage one of this project, we'll build a client that generates and sends lift ride data to a server in the Oracle cloud (https://www.oracle.com/cloud/free/#always-free).

The server will accept and validate requests, and send an HTTP 200/201 response.

Implement the Server API

The server side requires a Web server to host Java Servlet. A Web server can be a Tomcat or Jetty server or Spring framework.

In this task, you need to implement the POST API for the /skiers endpoint using Java servlets. The API should:

1. Accept the parameters for the operations as per the specification
2. Do basic parameter validation, and return a 4XX response code and error message if invalid values/formats supplied
3. If the request is valid, return a 200/201 response code and some dummy data as a response body.

The full APIs are defined by Swaggerhub.
https://app.swaggerhub.com/apis/yan.liu2/SkiRideAPI/1.0

Good simple illustrations of how to handle JSON request payloads are

1. https://edwin.baculsoft.com/2011/11/how-to-create-a-simple-servlet-to-handle-json-requests/
2. https://www.baeldung.com/servlet-json-response

Test each servlet API with POSTMAN or an equivalent HTTP testing tools.

Deploy your server on the Oracle cloud free tier instance you have created that runs tomcat or jetty.

Build the Client

We want a multithreaded Java client we can configure to upload a day of lift rides to the server and exert various loads on the server.

First you need to get a Java client to call your server APIs.

Write a simple test that calls the API before proceeding, to establish that you have connectivity.

To connect to your remote server, you need to call a client method using the Java 11 HTTP client classes or the Apache Java HTTP API. Each client per thread in a multithreaded environment to simulate multiple clients in concurrency.

Data Generation

Your client will send 10K POST requests to your server. You should create an object that generates a random skier lift ride event that can be used to form a POST request. Follow the rules below:

1. skierID - between 1 and 100000
2. resortID - between 1 and 10
3. liftID - between 1 and 40
4. seasonID - 2022
5. dayID - 1
6. time - between 1 and 360

Multithreaded Client

Your aim is to upload all 10K lift ride events to your server *as quickly as possible*. To do this we'll use multiple threads. The only constraints on your design are as follows:

- At startup, you must create 32 threads that each send 1000 POST requests and terminate. Once any of these have completed you are free to create as few or as many threads as you like until all the 10K POSTS have been sent.
- Lift ride events must be generated in a single dedicated thread and be made available to the threads that make API calls. You need to design this mechanism so that:
    - a posting thread never has to wait for an event to be available. This would slow down your client
    - it consumes as little CPU and memory as possible, making maximum capacity available for making POST requests

The server will return an HTTP 201 response code for a successful POST operation. As soon as the 201 is received, the client thread should immediately send the next request until it has exhausted the number of requests to send.

Handling Errors

If the client receives a 5XX response code (Web server error), or a 4XX response code (from your servlet), it should retry the request up to 5 times before counting it as a failed request. This probably means your server or network is down.

On Completion

When all 10k requests have been successfully sent, all threads should terminate cleanly. The programs should finally print out:

1. number of successful requests sent
2. number of unsuccessful requests (should be 0)
3. the total run time (wall time) for all phases to complete. Calculate this by taking a timestamp before you start any threads and another after all threads are complete.
4. the total throughput **in requests per second** (total number of requests/wall time)

You should run the client on your laptop. Thus means each request will incur latency depending on where your server resides. You should test how long a single request takes to estimate this latency. Run a simple test and send eg 500 requests from a single thread to do this.

You can then calculate the expected throughput your client will see using Little's Law and use this to guide your design.

If your throughput is not close to this estimate for your test runs, you probably have a bug in your client.

Profiling Performance

With your load generating client working wonderfully, we want to now instrument the client so we have deeper insights into the performance of the system.

To this end, for each POST request:

- before sending the POST, take a timestamp
- when the HTTP response is received, take another timestamp
- calculate the latency (end - start) in milliseconds
- Write out a record containing {start time, request type (ie POST), latency, response code}. CSV is a good file format.

Once all phases have completed, we need to calculate:

- mean response time (millisecs)
- median response time (millisecs)
- throughput = total number of requests/wall time (requests/second)
- p99 (99th percentile) response time. Here's a nice article about why percentiles are important and why calculating them is not always easy. (millisecs)
- min and max response time (millisecs)

You want to do all the processing of latencies in your client after the test completes. The client should calculate these and display them in the output window in addition to the output from the previous step, and then cleanly terminate.

Submission Requirements

Submit your work to Canvas Assignment 1 as a pdf document. The document should contain:

1. the URL for your git repo. *Make sure that the code for the client part 1 and part 2 are in seperate folders in your repo*

2. a 1-2 page report of your client design. Include major classes, packages, relationships, whatever you need to convey concisely how your client works. Include Little's Law throughput predictions. The report should following the IEEE conference paper template

   https://www.ieee.org/conferences/publishing/templates.html

3. Building Client- This should be a screen shot of your output window with your wall time and throughput. Also make sure you include the client configuration in terms of number of threads used.
4. Profiling Performance - run the client in item 3, showing the output window for each run with the specified performance statistics listed at the end.

Grading:

1. Server implementation working (5 points)
2. Client design description (5 points) - clarity of description, good design practies used
3. Building Client - (10 points) - Output window showing best throughput. Points deducted if actual throughput not close to Little's Law predictions.
4. Profiling Performance - (10 points) - 5 points for throughput within 5% of Client Part 1. 5 points for calculations of mean/median/p99/max/throughput (as long as they are sensible).

Additional Useful Information

Building Swagger Client with Java 11

You need to modify your POM, add the following dependencies:

```
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.2.11</version>
</dependency>
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-core</artifactId>
    <version>2.2.11</version>
</dependency>
<dependency>
    <groupId>com.sun.xml.bind</groupId>
```

```
    <artifactId>jaxb-impl</artifactId>
    <version>2.2.11</version>
</dependency>
<dependency>
    <groupId>javax.activation</groupId>
    <artifactId>activation</artifactId>
    <version>1.1.1</version>
</dependency>

<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
```

Problems with GSON import in .war file in Intellij

Sometimes there's an issue building the GSON jar file into a servlet, such that when the servlet is deployed it fails because GSON is missing.

Try adding the gson jar to your project from the Maven website

For IntelliJ you also need to:

1. Go to project structure
2. choose the artifacts tab
3. select the gson maven package and put it into the lib directory of the artifact.

Compile and deploy!