Client Design Description

Server:
The homework includes two servers(Java and Go). The Java server(MusicServlet) uses tomcat while the Go server(GoServer) applied Gin. Both the two servers implement two restful APIs(Get and Post). The GET request will return an album object that user tried to fetch. The POST request can upload an image and a profile object. The json format of that object and the size of image will be printed out in terminal.

Go:
GET path: http://host:8080/albums/:id
POST path: http://host:8080/albums

Java:
GET path: http://host:8080/MusicServlet_war/albums/:id
POST path: http://host:8080/MusicServlet_war/albums

When designing the POST API, I applied **@MultipartConfig** annotation and provided to deal with logic related to multiple files upload so that I can deal with image and profile object separately.

Client:
There are two clients for this assignment. The client1 only calculates the wall time and final throughput. The outcome has been attached to the images folder. While the client2 takes advantage of CSV data to calculate wall time, final throughput, min latency, max latency, mean latency, median latency and P99 latency.

Logic of calculating these statistics:
1:Applied CSVPrinter to record the start time, latency, responseCode and requestType. When the the statusCode is between 200 and 300, I will record it to target csv file. When writing the file, I used the **'synchronized'** key word because there are multiple threads writing the same csv file at the same time. This key word can make sure that every writing task will not be interrupted by other threads.

2: Used file reader to read the data stored in my csv file and created an ArrayList to store all the valid data.

3: Sort all records based on their latency, the first record has the minimum latency while the last one has the maximum latency. I can also easily get the mean and median latency when they are sorted.

Each client consists of a SingleThread and a MultipleThreadProcessor class. Each single thread will implement the get and post requests each for 1000 times. Each request can try five times at maximum. If all of them failed, the request will be skipped. In the MultipleThreadProcess, I used for loop to separate each group of threads. Besides, I passed a countDownLatch instance to each single thread. This can make sure that my following data processing will not started until all threads finish their tasks.