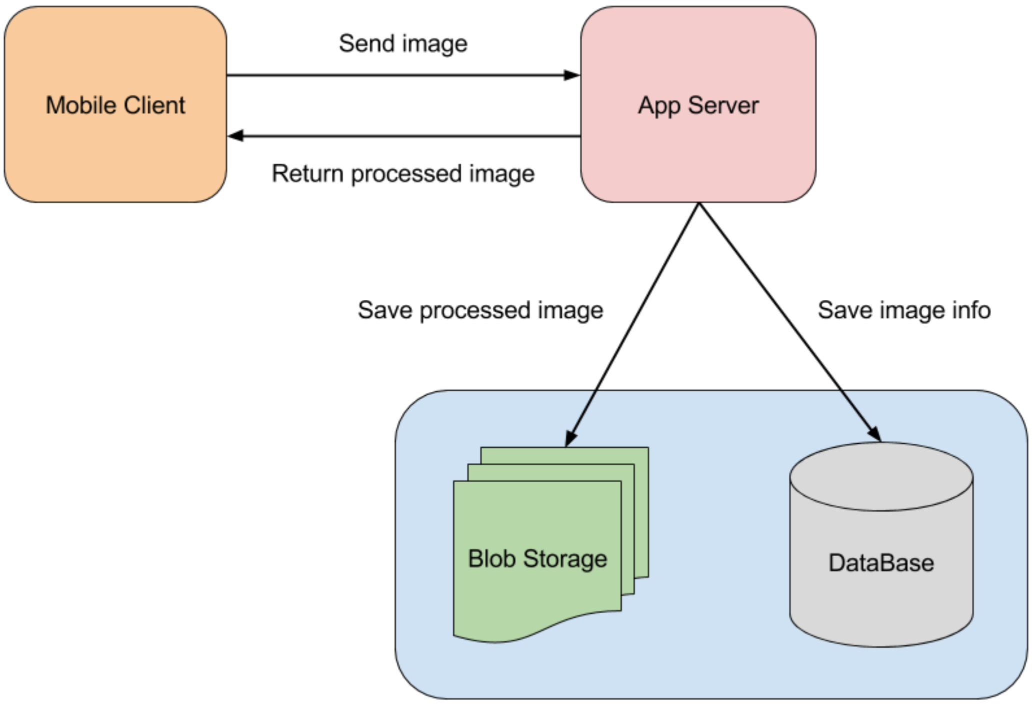
1) Sketch out a diagram representing the architecture for the following scenario:
   a) receive images from a native mobile app installed in an user's' mobile device
   b) process such images
   c) store results of the processing stage (point b)
   d) return processed images to users.



Assume only 10 users. The simplest architecture is Single app server + MySQL(DB) + Local File System(Blob Storage). The application server handles HTTP and processes images as shown in the above diagram. In order to avoid storing the actual image in a database, we can store an image into local file system and save the file path in the database.

As the number of users grows, we need to consider the performance. To get an idea, here is a quick statistics for RGB to grayscale conversion.
Spec) Processor: 2.4GHz Intel Core i5, Memory: 4GB 1600MHz DDR3, Execution Env: Golang

Size	Execution Time	Number of Photo / Sec
1 MB	83.993047 ms	12
2 MB	105.88302 ms	9
10 MB	156.310057 ms	6

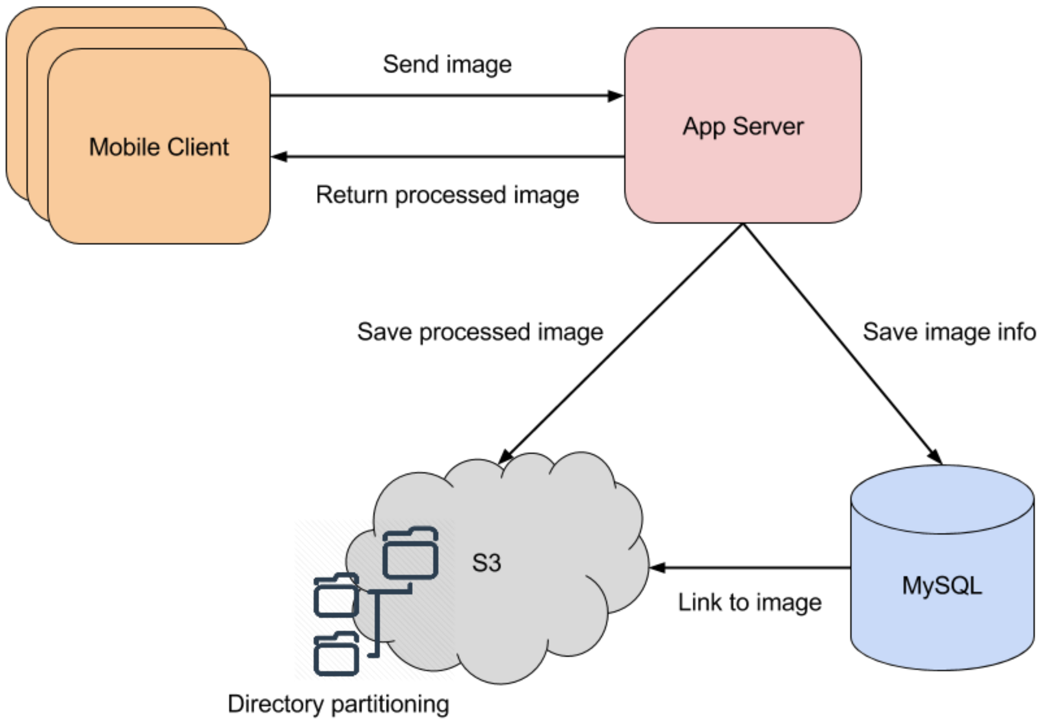
Table 1: Statistics of Image Processing

Considering network latency and disk write will cost almost the same time, this performance is acceptable.

2) Describe if you'd make any modification to architecture from Task 1 when you have

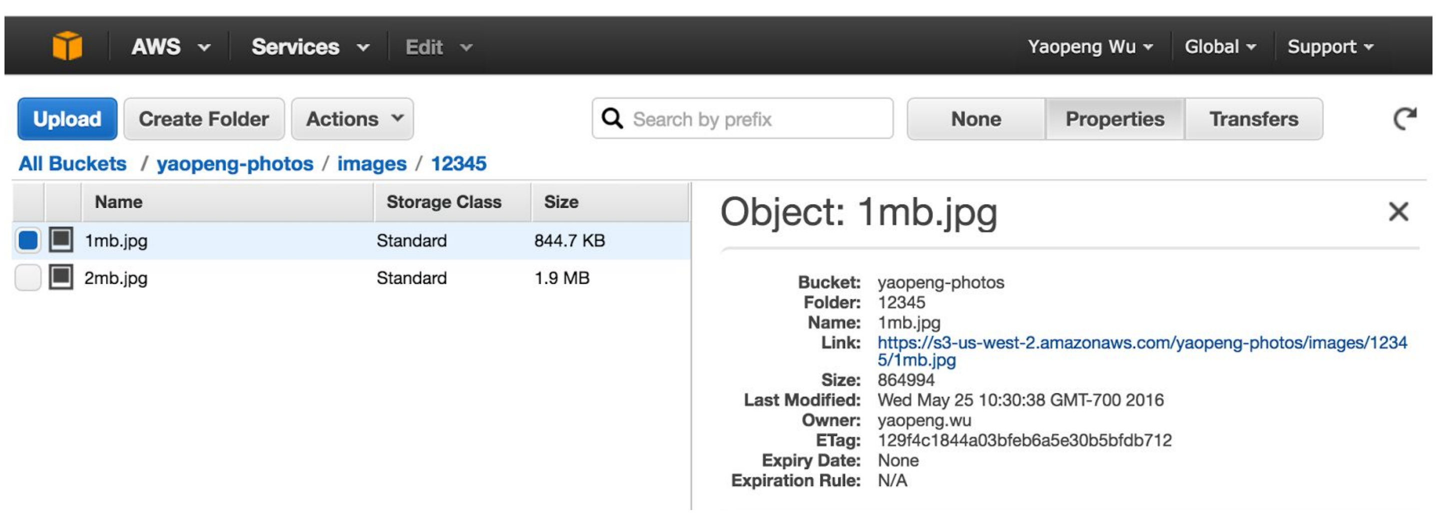
a) 1000 users :

Architecture: Single app server + MySQL + S3



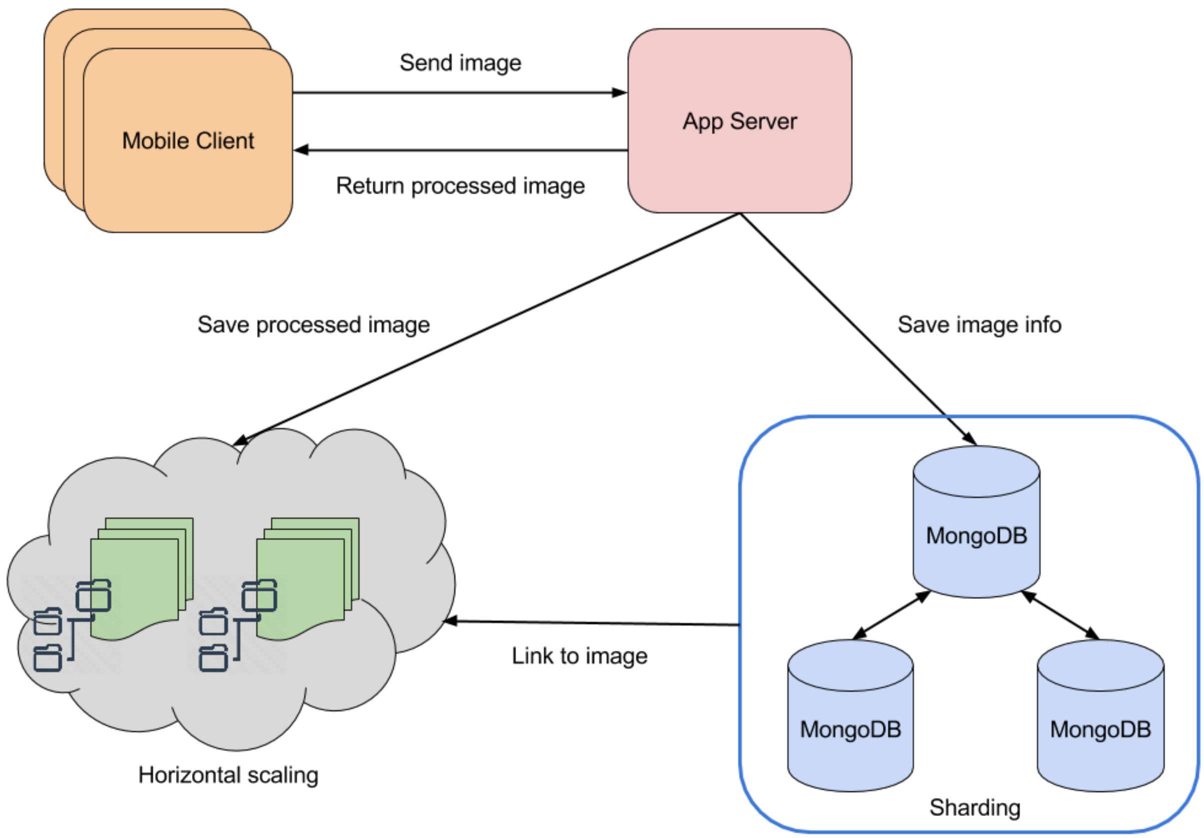
Changes:

1. Move to a cloud storage (Amazon S3) because of the storage cost
   Assume in the worst case, every user uploads 10 MB image once a day. The storage will grow 10 MB * 1000 users * 365 days = 3,650 GB in a year.
2. Partition the storage folder by user ID (Hashing)
   To improve the file access speed, we use folder hashing. Every time a new user registers, we create a new image folder by the user ID. Hence, the path to an image file becomes "*/images/userID/fileName.jpg*".
3. To access an image in S3, we can save the image link in the database.
   i.e) https://s3-us-west-2.amazonaws.com/yaopeng-photos/images/12345/10mb.jpg



b) 100,000 users :
Assume users are active 10 hours during a day. (e.g 10 * 3600 = 36,000 sec)
100,000 / 36,000 ~ 3 users/sec => 3 photos/sec. Considering the system can process at least 6 photos/sec according to Table 1, a single app server is sufficient.

Architecture: Single app server + MongoDB + S3



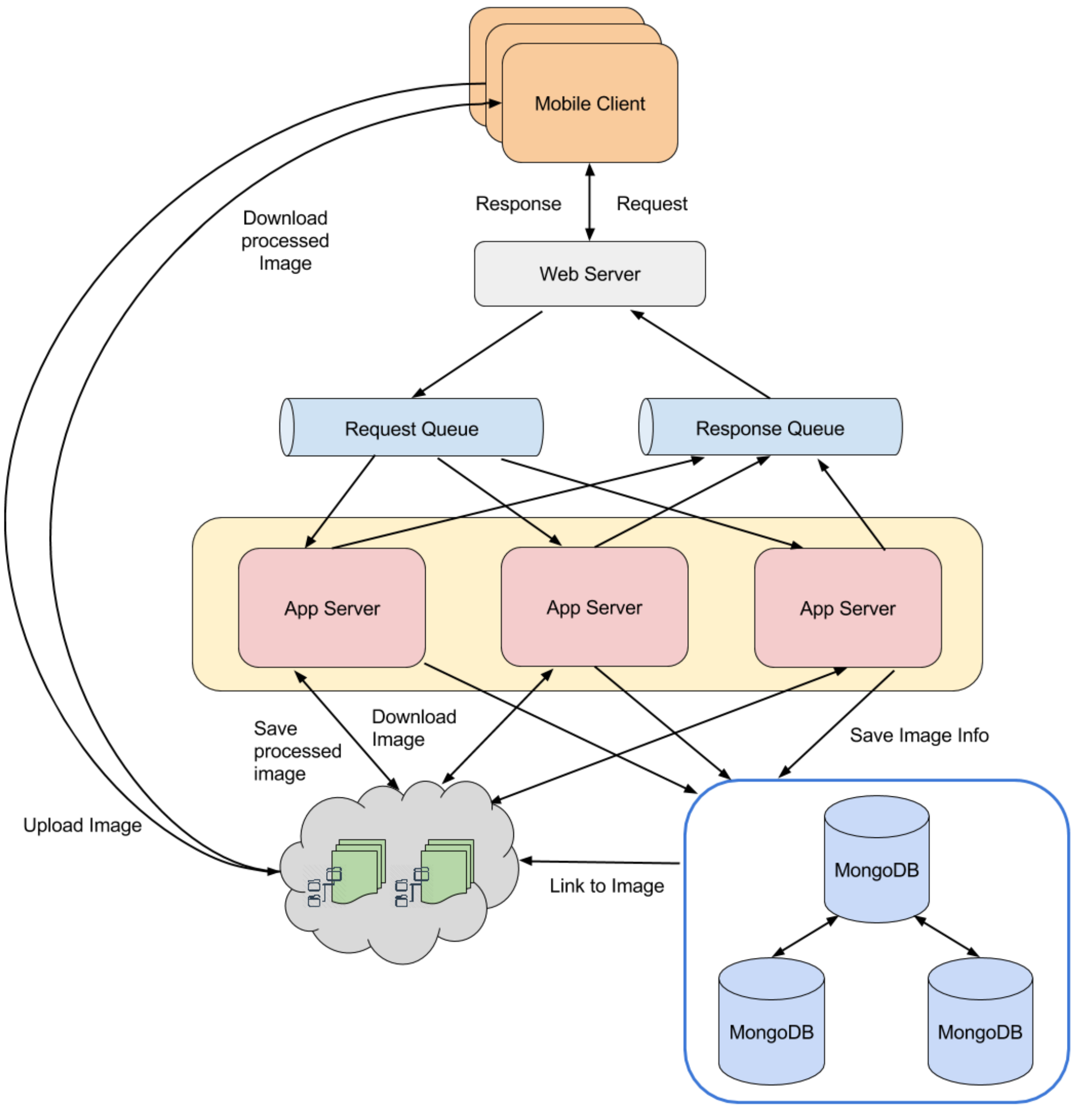
Changes:
1. Migrate MySQL to MongoDB
   Considering the possibility of 10 concurrent accesses, we'd better to migrate MySQL to MongoDB because MongoDB handles concurrent connection better.
2. Database & Storage horizontal partitioning (Sharding)
   Assume the image size 10 MB, and the metadata of the image 5 KB. The database and storage will grow to the following size respectively in a year. In order to improve access performance, we should scale them horizontally.
   - 10 MB * 100k users * 365 = 365 TB/year [S3]
   - 5 KB * 100k users * 365 = 180 GB/year [MySQL]

c) include timelines it would take to implement functioning architecture for a) & b)
Gantt chart is a popular tool to plan and track of a project. Please navigate to this link for the full chart.

Assume the scenario from Task 1 happens only once a day per user. Please make modifications to your sketch as needed. If you're describing use of different tools, describe how you'd implement these tools.

3) Suppose that all the users from 2a and 2b upload their images to the architecture every day twice a day, all at the same time (e.g. 10am and 5pm). Please explain if the architecture proposed at point 2 would be able to correctly process all the requests; if not, please describe what should be added or how the architecture should be modified.

Architecture: Multiple app server + MongoDB + S3



Changes:

1. Single server to multiple server:
   To handle such large number of requests at only particular times (e.g. 10am and 5pm), we can utilize Amazon's auto-scaling to easily scale our image processing engine. Concretely, we can put the app server in an auto-scaling group and Amazon will add more EC2 instances according to traffic.
2. Introduce web server and message queue
   To guarantee delivery such large number of requests, we introduce message queue. To avoid the situation where web server proxy message round robin, and one particular app server always gets a large size of image, we add a message queue behind the web server. This approach also solves the complex failure handling problem as well. The overall data flow will be as follows:

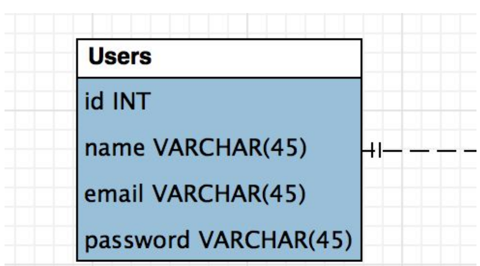
1) The client sends a request to the web server
2) Instead of proxying it to a server directly, the web server puts the request on a message queue.
3) One or more servers are subscribing to this request queue and one of them pulls in the message containing the request.
4) After the request is handled and response produced, the server puts the response on the response message queue.
5) The web server is subscribed to the response queue and forwards the response back to the client.

3. Client uploads an image directly to S3
   Because of containing an image, the payload could be heavy, thus too much stress on the web server. Since a native mobile app is installed in an user's' mobile device, we can ask the mobile app to upload an image directly to S3[1]. The mobile app then sends the request with the link to the image in replacement of the actual image. This way, we can reduce the size of payload significantly.
   App server can then download the uploaded image from the bucket using the link and process it and save the metadata of the image and resulting grayscale in the database.
4. If we introduce more a complicated image processing method at later stage, we can consider using GPUs to offshore computing intensive tasks.

4) Coding question

a) Write the code to implement the section of the architecture responsible for the storage of the information (see point 1a). It is possible to use any kind of programming language and database solution, such as entity relationship or NoSQL, or a combination of multiple techniques.

In addition to a user database, we have an image database for storing image metadata, where we can refer the user as "user_id". Because it is not straightforward to store array in MySQL, MongoDB is more suitable for the image database in this case. If we use MySQL as the user database and MongoDB as the image database, the schema will be like the following:




```
{
    "_id": {},
    "user_id": "",
    "file_name": "",
    "image_url": "",
    "timestamp": {},
    "histogram": []
}
```

Please also refer to the model file imageInfo.go in the repo.

[1] For security reason, we need to created a private S3 bucket with an aggressive expiration policy. The native mobile apps each gets a unique IAM role with **write-only** access to this bucket.

b) List the endpoints of a RESTful API that could be used by a client to interact with the described architecture.

The client can be:

- The native mobile app for transaction
- Backend system for analysis

Resource	GET	POST	PUT	DELETE
/images	Return a list of all images	Method not allowed	Bulk update of all images	Delete all images
/images/:user_id	Return a list of all images for a specific user	Create a new image for a specific user	Bulk update of all images for a specific user	Delete all images for a specific user
/images/:user_id/:image_id	Return a specific image	Method not allowed	Update a specific image	Delete a specific image

c) Write the code which would allow the backend to perform the following operations:

All working code is available in here: https://github.com/gyoho/image-processing

- ➔ To run the the program, execute the following command
  $ ./server
- ➔ To recompile the files, please paste the credentials[2] in server.go and s3.go and execute the following command
  $ go build server.go

■ receive an input image from a client, convert it to grayscale, compute the histogram and store both the original image and the histogram into a database;

- Functions/Methods involves in this task:
  - HTTP Router
  - Create Image Method
  - Helper Function
  - ConvertToGrayScale
  - UploadImage

- For testing, please refer to the following screenshots

[2]dbUser string = "yaopeng"
dbPassword string = "admin"
aws_access_key_id := "AKIAJ3MIC5B3I7O3HWXQ"
aws_secret_access_key := "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"

REST API test

Posting an image with user ID in URL parameter successfully return the an appropriate response.

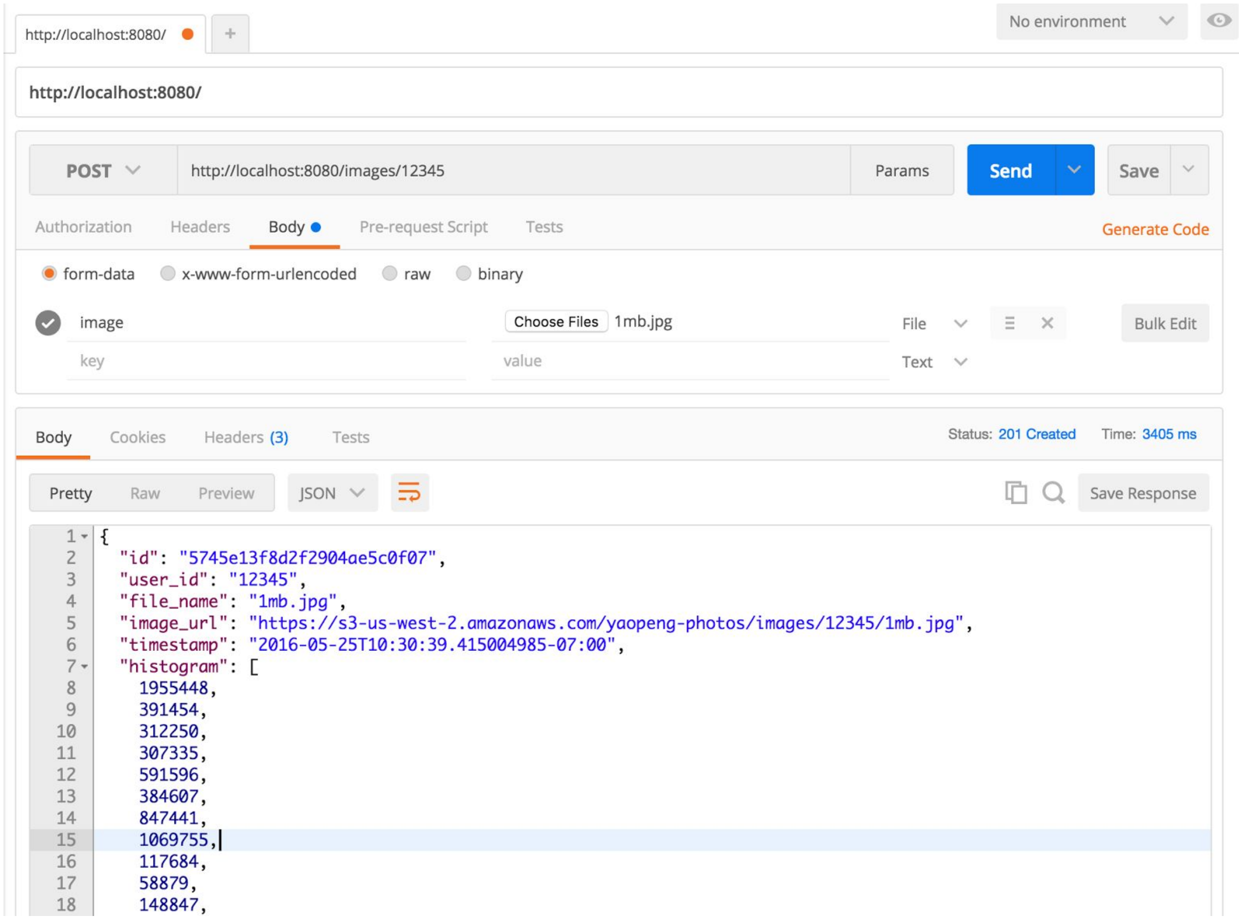
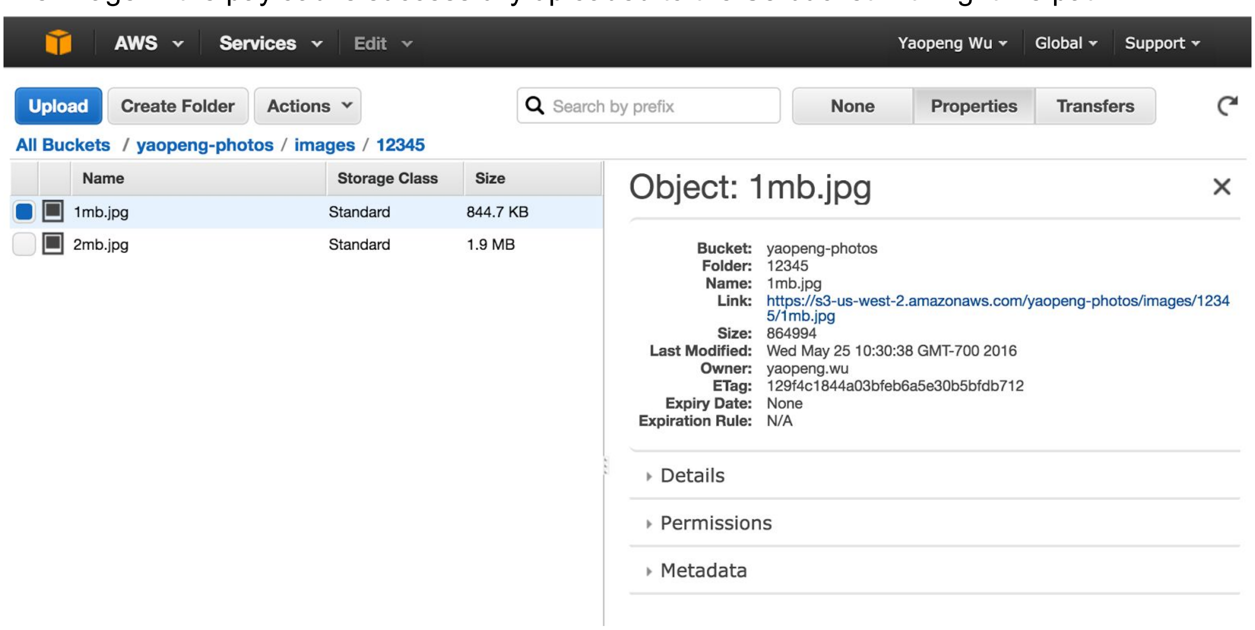


Image in Amazon S3 bucket

The image in the payload is successfully uploaded to the S3 bucket with right file path.

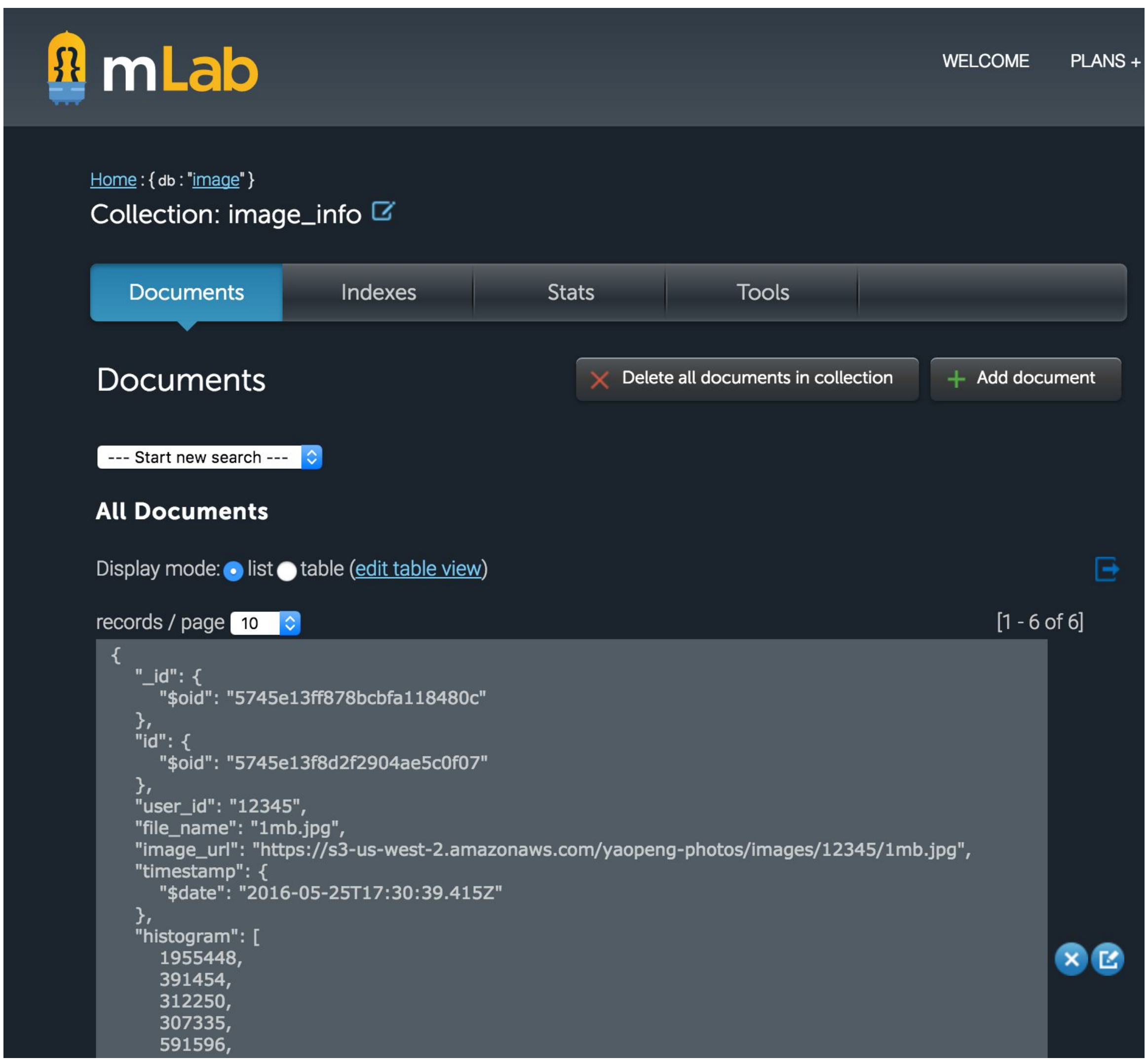


Processed image in grayscale

The original image is successfully converted into grayscale.

Image metadata in mLab

The metadata of the uploaded image with the histogram of the image in grayscale is successfully stored in the Image_info collection in the image database in mLab.

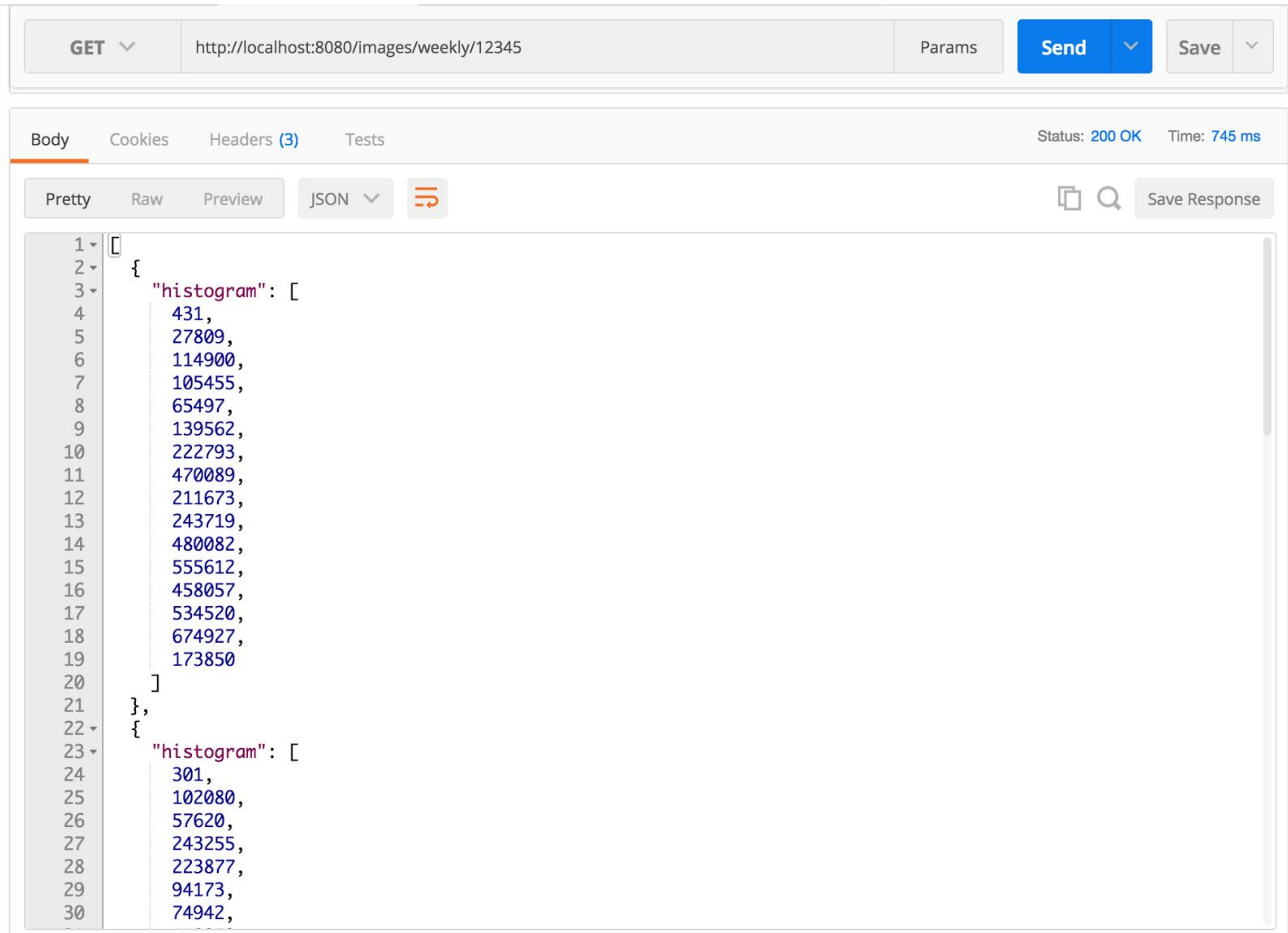


■ extract the histograms of the current week for a single user;

- Functions/Methods involves in this task:
    - [HTTP Router]
    - [GetWeeklyHistograms Method]
    - [retriveHistogramsOfWeek Function]
- For testing, please refer to the following screenshots

REST API test

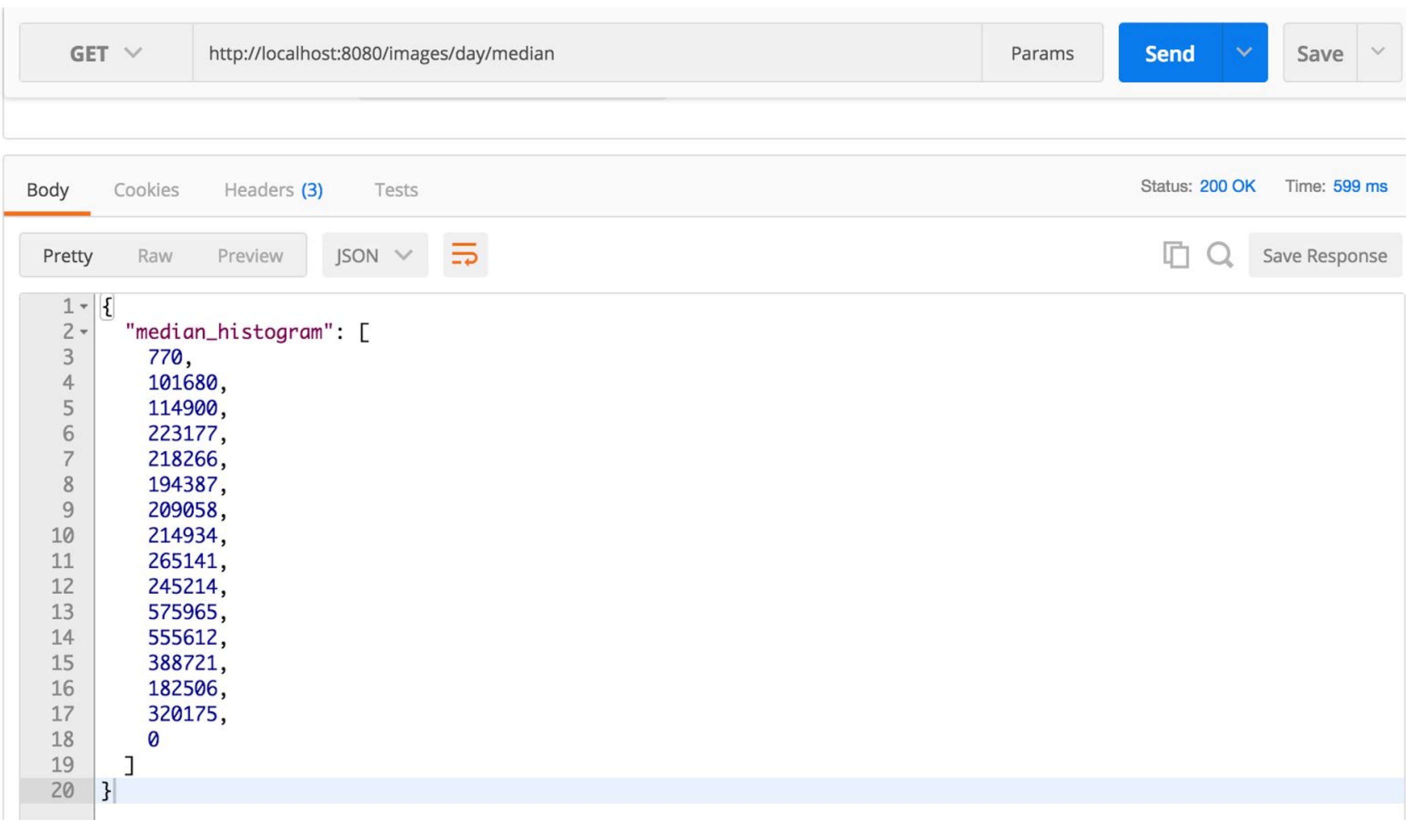
The GET request with user id in URL parameter successfully returns the all histograms of the current week for the specific user.



■ extract the median histogram of the current day for all users (the output is a single histogram); Because Mongo does not yet have an internal method for calculating the median, We need to implemented our own median function. I used the 2 Heap approach to efficiently and dynamically calculate the median with time complexity O(logN). If we're concerned more in time and memory than in precision, we can consider applying quantile approximations available in VividCortex/gohistogram package using *bin size = 0.5 (code ref)*. Algorithm-wise, we can also consider Median of Medians Algorithm for efficiency.

- Functions/Methods involves in this task:
    - HTTP Router
    - GetMedianHistogram Method
    - Helper Function
    - retriveHistogramsOfDay Function
    - calculateMedianHistogram Function
    - Files for the 2 heap implementation is located at the struct folder

- For testing, please refer to the following screenshots



d) Write a piece of code that, given a user id as input, returns n user ids who have the most similar histograms with respect to the input user id.
Assume the scenario where an user uploads only one photos per day, and we're calculating the above requirement for the current day.
Because Golang doesn't fully support OpenCV yet. I couldn't use OpenCV's CompareHist function, which seems the best choice to calculate the difference/similarity between histogram for this problem. Therefore, I implemented my own function to compare the difference/similarity between histograms. Because our histogram is an array with size 16, the problem can be translated to simply comparing between arrays. A delta between two arrays can be simply calculated by summing up the weighted delta of each element of the same index.
Mathematically, it can be presented as: $\Sigma\ \mathrm{Abs}(x_1 - x_2) / \mathrm{max}(x_1, x_2)$. This equation measures the difference between two arrays, so the smaller the result the more similar the arrays are. Moreover, we can use max heap with size n to keep track of the n most similar histograms efficiently.

- Functions/Methods involves in this task
  - HTTP Router
  - GetUserIDWithSimilarHistogram Method
  - Helper Function
  - retriveHistogramsOfDay Function
  - extractUserIDWithSimilarHistogram Funciton
  - Files for heap implementation of similarity histogram is located at the struct folder

- For testing, please refer to the following screenshots