**Image Api**

**My understating**

When the server starts, it will first read the ‘image.txt’ file and prepares the http request, to hit the Predict API REST endpoint. Doing this, it will get the list of all the tags (in the json) with the value of the probabilities associated with the image. Therefore, I have to create an in memory data structure (Hashmap in my case), which will store the tags as the keys and list of images associated with the tag. Thus when the user enters the tag in the textbox, the server looks for the tag in the TAG\_MAP in O(1) time and if present, it returns the json containing the list of images associated with that tag along with the value of probabilities of the image being associated with the tag. The client thus displays the images to the user.

**Data Structure**

The in memory data structure used to implement the backend server is Hashmap. So when the server gets started, it first reads the images.txt file. Then it hits the predict rest api endpoint (https://api.clarifai.com/v2/models/aaa03c23b3724a16a56b629203edc62c/outputs) using http post to get the json of the tags with probability of the tag being associated with the image.

Output Json of the predict endpoint:

{

"status": {

"code": 10000,

"description": "Ok"

},

"outputs": [

{

"id": "f619f2c6739f42d1b1863469c0d26db7",

"status": {

"code": 10000,

"description": "Ok"

},

"created\_at": "2018-06-13T16:54:58.344302542Z",

"model": {

"id": "aaa03c23b3724a16a56b629203edc62c",

"name": "general-v1.3",

"created\_at": "2016-03-09T17:11:39.608845Z",

"app\_id": "main",

"output\_info": {

"message": "Show output\_info with: GET /models/{model\_id}/output\_info",

"type": "concept",

"type\_ext": "concept"

},

"model\_version": {

"id": "aa9ca48295b37401f8af92ad1af0d91d",

"created\_at": "2016-07-13T01:19:12.147644Z",

"status": {

"code": 21100,

"description": "Model trained successfully"

}

},

"display\_name": "General"

},

"input": {

"id": "ec0f6003d0a44a7490c49f42eaf7ae0e",

"data": {

"image": {

"url": "https://farm7.staticflickr.com/5769/21094803716\_da3cea21b8\_o.jpg"

}

}

},

"data": {

"concepts": [

{

"id": "ai\_XclQc6bP",

"name": "traditional",

"value": 0.918077,

"app\_id": "main"

},

{

"id": "ai\_15WdDpTW",

"name": "site",

"value": 0.90343666,

"app\_id": "main"

}

]

}

}

]

}

After receiving the json, I create a reverse hashmap which stores the tag of the image as the key and list of 10 most probable images with its corresponding probability as the value. Following is the example json:

{

"tag": "**portrait**",

"length": 7,

"data": [

{

"**URL**": "https://c2.staticflickr.com/4/3113/3197300261\_c9a0d7ee93\_o.jpg",

"**probablity**": 0.9781549

},

{

"URL": "https://c6.staticflickr.com/4/3661/3291298501\_e958aea912\_o.jpg",

"probablity": 0.9760653

},

{

"URL": "https://c8.staticflickr.com/9/8078/8343384446\_e346d9c2a5\_o.jpg",

"probablity": 0.9708998

},

{

"URL": "https://c3.staticflickr.com/4/3750/9031823026\_48ebe4d436\_o.jpg",

"probablity": 0.95477974

},

{

"URL": "https://farm5.staticflickr.com/3710/10921149963\_e04ba75721\_o.jpg",

"probablity": 0.94277596

},

{

"URL": "https://c8.staticflickr.com/1/33/48901634\_c792cbab58\_o.jpg",

"probablity": 0.861478

},

{

"URL": "https://farm1.staticflickr.com/8087/8563450894\_8f2fda7196\_o.jpg",

"probablity": 0.8022853

}

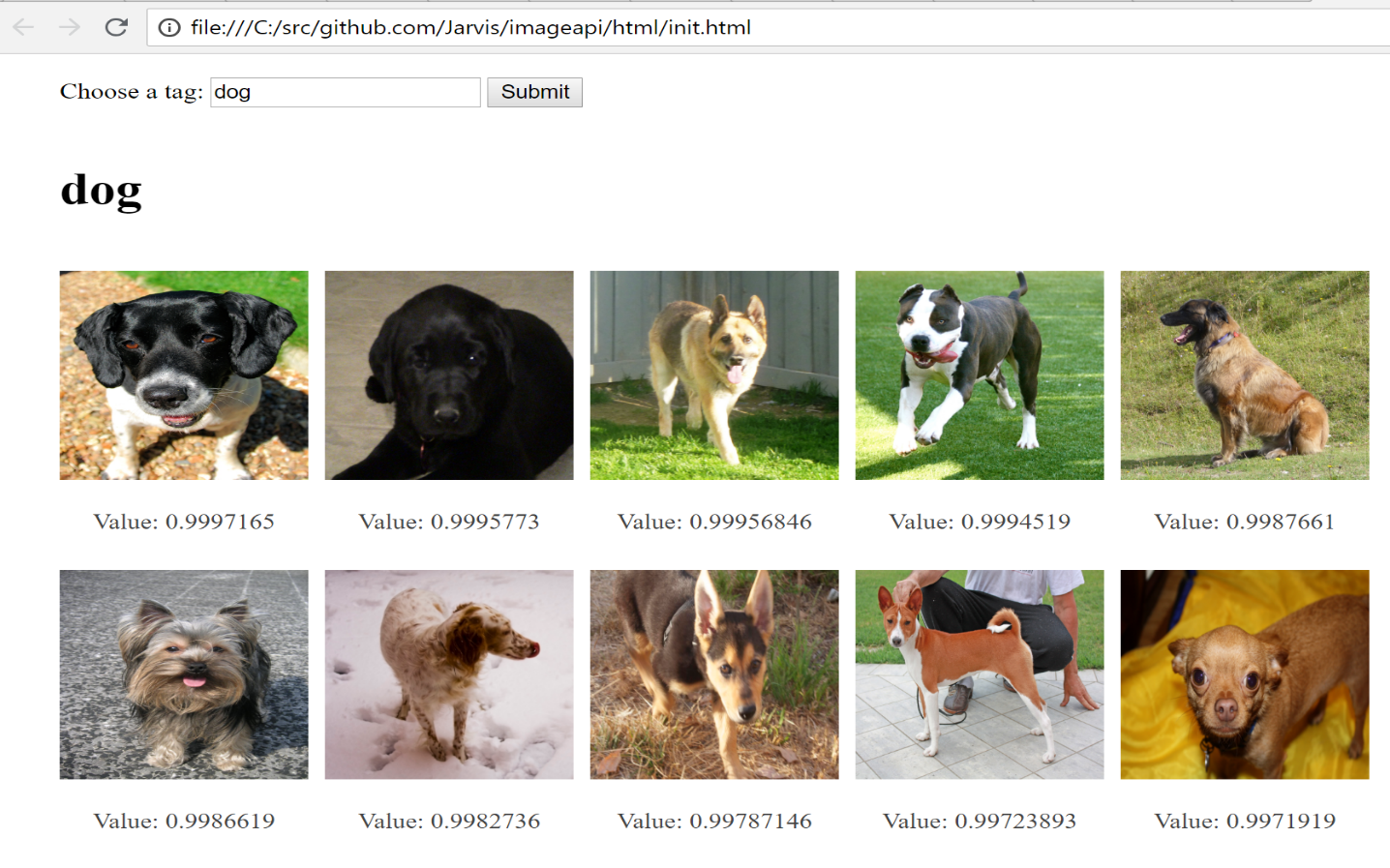
],

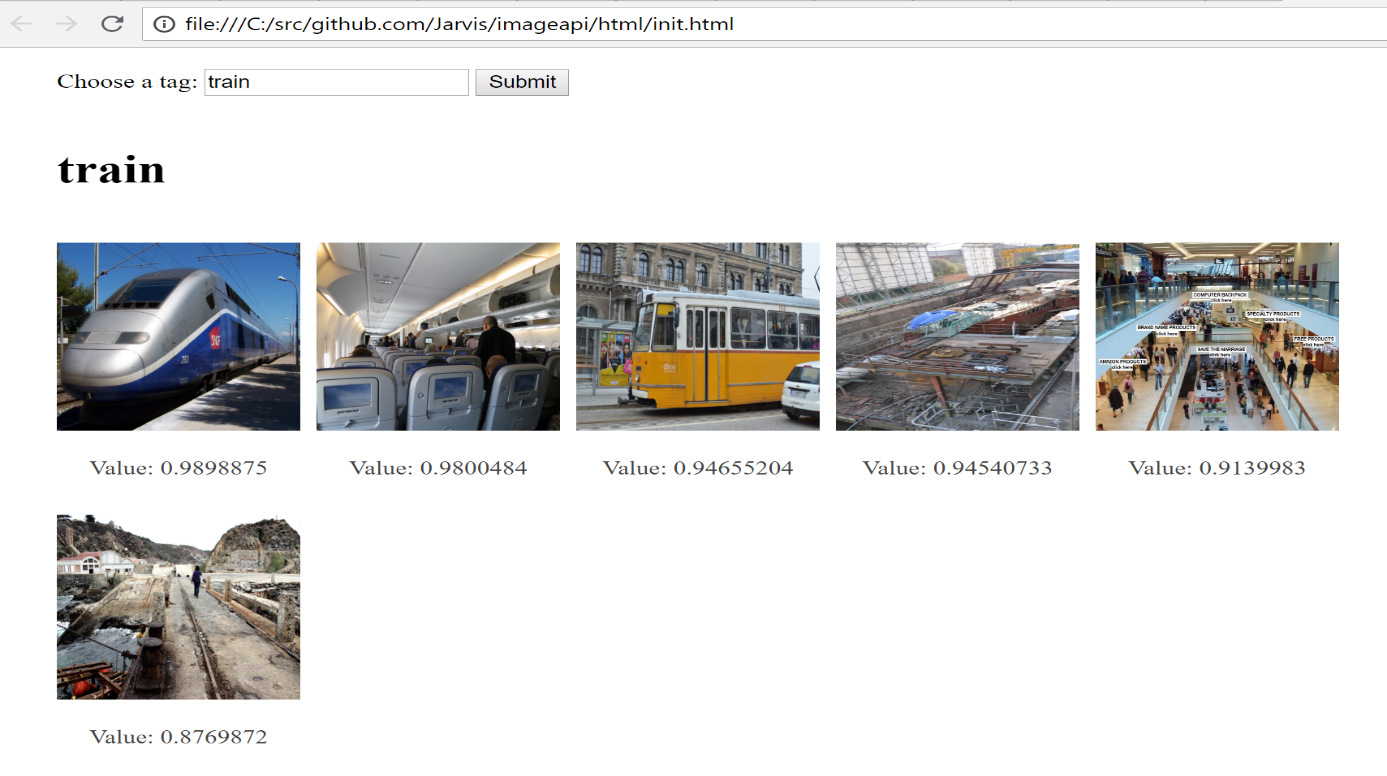
"status": 200

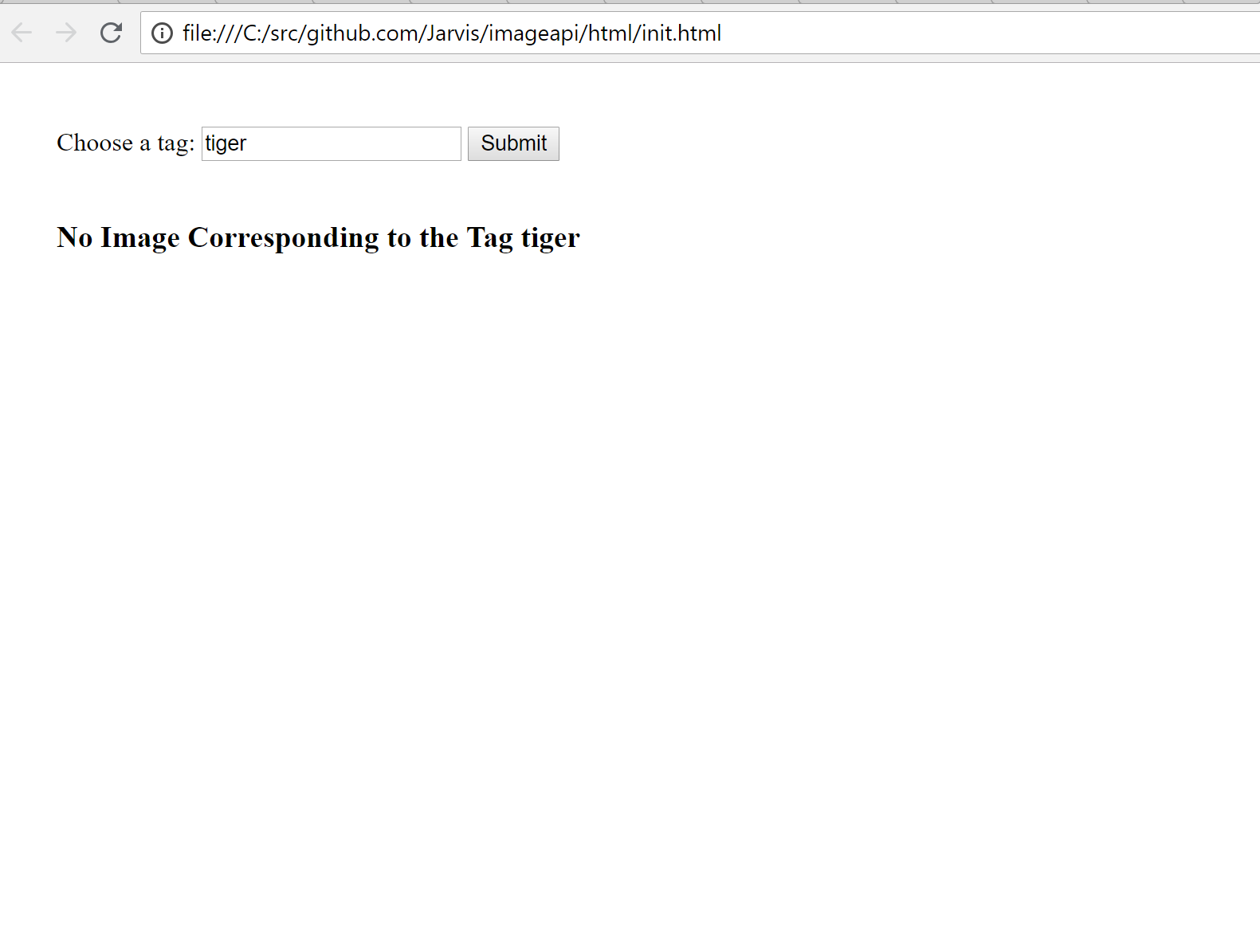
}

So, when the user enters the tag in the textbox, the server simply looks for the tag in the TAG\_MAP. If the tag is available server forms the above json and send it to the client. The client uses javascript to parse the json in order to display the thumbnail of the images.

Following is the screenshot of the demo of the Webapp:







**Directory Structure**

|--imageapi

|--html

|--init.html

|--script.js

|--style.css

|--main.go

|--model.go

|--images.txt

**Run:**

The backend server runs on the 8080 port. In the terminal or power shell go to the directory containing the main.go file. Execute the command “go build” and “./imageapi.exe” (for windows) and wait for the server to turn up. The server takes time to turn up because it prepares the http request for each image, to find the tags to fill the TAG\_MAP data structure. Make sure the necessary app permissions and windows firewall defender permissions are given to the server to turn up. I have used mux router to route the handle in the webapp. Complete documentation of the mux can be found here <https://github.com/gorilla/mux> .

For hosting the html templates, I have used CORS toggle chrome extension which allows cross platform. Go in the html folder and open init.html in the chrome browser. Make sure that CORS toggle is activated. One can also use the Postman application to check the output json returned by the server.

**Improvements:**

There are plenty of improvements that can be done to this assignment if more time is available. Listing some of the improvements which can be very useful.

1. There is no check for the duplicate image url in the image.txt file. I have assumed that each link refers to the unique and different image.
2. While creating the tag map I have used url directly instead some **short unique identifier** can be used which can refer to the image uniquely. By doing this we can save lot of memory waste because long url occupies more memory and are stored multiple times.
3. The web app currently works for **single tag** at a time. For e.g. if you search for {bus, train} it won’t show the results instead we have to either look for bus or train at a time.
4. Use of regular expressions can improve the performance of the system drastically. For e.g., if the user looks for **transport** or **transport** tag, he gets nothing but instead if he looks for the **transportation system,** he will get the images showing the same.
5. User preferences can also improve the user experience like he should be able to set the number of image he would like to see instead of at most 10 most probable images.
6. I have maintained the list of the images in the TAG\_MAP corresponding tag. The list is basically sorted based in probabilities values. Each time I found the image with the existing tag, it looks for the appropriate index for the image to be put such that the sorted behavior is not hampered. This require shifting of the array if the image is put at the index which is not at the end of the array. This process can be improved by using **max heap** which always returns the maximum probable image when the tag is searched.
7. So far, I have given my best to split the code into three components model, view and controller. The html folder contains the view of the application while main.go and model.go are the controller and the model respectively. Since I am new to Golang and this being my first encounter with it, I couldn’t use the golang standard directory structure to implement MVP in given time frame. MVP architecture will surely improve the scalability of the application. If given more time, I can develop a full-fledged MVP application with standard directory structure using golang.