Mask Detection AWS Web Application

Operation instruction

In this assignment, we are using AWS Educate account for launching EC2 instance. After you log into the AWS Educate account, go to the AWS console page and then you will see our instance has been stopped. Then you should start the instance and SSH'd into it using the provided key_pair.pem file. The pass phrase of the key (if needed) is: 'ece1779pass'. Then you will see our web app in the Desktop folder with a file named App. After you connect to the instance, to start our application in the terminal window, there are two ways:

1. Type 'flask run --host=0.0.0.0 --port=5000', hit Enter, then you will see the information 'Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)' within seconds, meaning that our app is running on this address.
2. The shell script called start.sh that can initialize our web app is included in the folder named App. Inside this shell script, gunicorn is used to start our Flask app. To start the app using this shell script, first type 'chmod +x start.sh' in the terminal to allow for the gunicorn execution, then type './start.sh' to run the shell script, you will see then information 'Listening at: http://0.0.0.0:5000' in the terminal within seconds, meaning that our app is running on this address.

To get access to our web app and test its functionality, you have to replace the host address 0.0.0.0 with the public IPv4 address of the instance, which is 54.208.156.49 in our case, and keep the port 5000 unchanged since we have already opened that port for all users outside the instance.

Application functions

Create account:

The default admin accounts have already been defined in our database, and only the administrators could create new users. The administrator accounts are (username: Shixiong, password: 123456) and (username: Qixuan, password: 654321). To create a new user, you should input a never registered username and email that would help users to recover the password from sending a new random password to that email. Inside the register function, we would first check whether every required field is filled, and then check the existing username and email in our database to make sure there is no duplicated one. The last checkpoint is to check the format of provided email. After everything goes well, we would hash and salt passwords and store all the registered information in the database.

Login & Recovery:

To log into your account, the app promotes users to give their username and password, and users cannot use email to login. The server would check every field is filled and check the input data with the data from our database. If all matched, it would navigate users to user pages. If users forget their passwords, the user should provide the registered email to server. Then the

server would generate a random password and send it to that email and would print the successful message to the webpage. The sender is a new registered gmail for this assignment. Then the database is updated with the new generated password (hashed and salted). The user should be able to login with this new password.

Change passwords:
After users log into the account page, they could change the password by clicking the button. To change the password, the user should provide the username and old password, which would be checked by servers to make sure all provided data match with data from the database. Then the user should type twice the new passwords, which would also be checked to make sure that they are identical, and then the server would update the database. After all things are done, it would lead you to the login page.

Upload:
For uploading photos, the user should retype the username and password, then choose a photo to upload. This file would store in the uploads folder first. The server would check username and password as before; in addition, the server would not allow other types of file to upload, only types of image. Then there is a limitation for the storage size of the image, which cannot exceed 8MB. Then the mask detection algorithm would be used to get the output information of that image. Before storing it to the database, if two images share the same name, the server would check the contents of images. When they are different images, the server would add a timestamp at the end of the second filename, then store it in the solutions folder and store the filename to the database. After successfully uploading, it would give the successful message including number of face, number of masked and number of unmasked; also, there is a button navigating the users to account page.

Upload history:
In the user account page, it would contain a menu and four image listview for different categories so that they are able to browse the lists of previously uploaded images.

API Testing:
We use Postman to create two collections named “Register” and “Upload”. For “Upload”, the relative URL is “/api/upload” with the “POST” method posting three parameters: “username”, “password” and “file”.
For “Register”, the relative URL is “/api/register” with the “POST” method. Since our app required email to register, which is different from the description of assignment requirements, the solution to figure it out is that if testers, who use “/api/register” to test, did not pass the email parameter, the server would default to use the server email address (“uoft.qxz.sxg@gmail.com”) as a tester email address, and disable the duplicated email checking algorithm. Although we know this action is not safe and the testers cannot recover their password from sending email, in the current situation we have to do that in order to meet the requirements from the assignment.pdf for autonomic testing.

General architecture

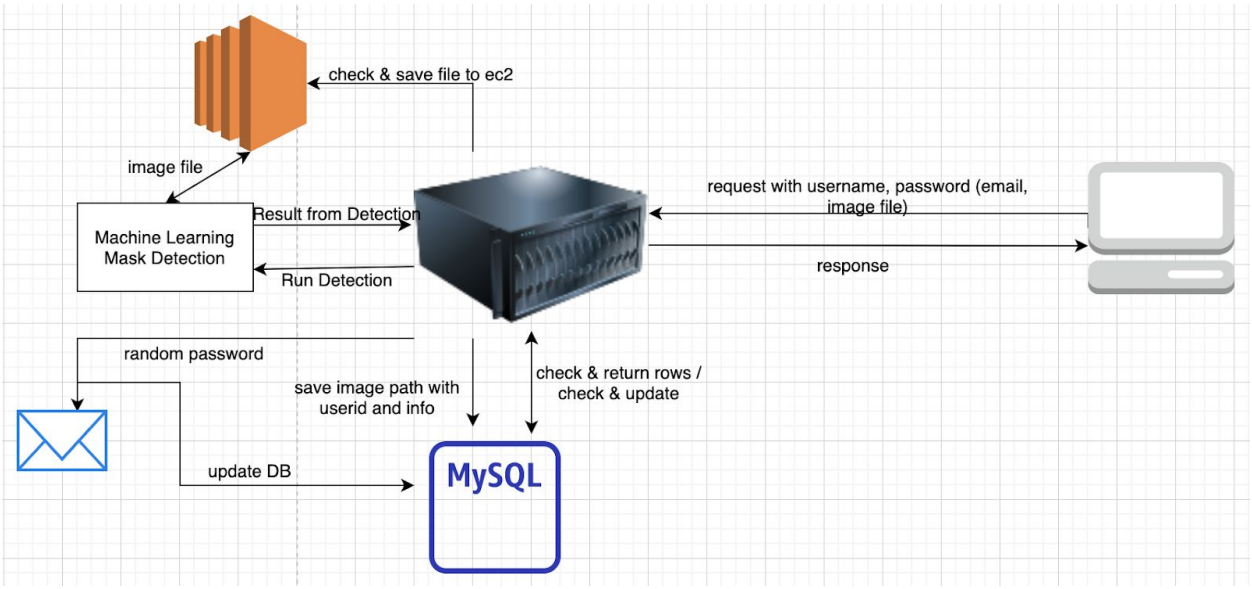

Figure 1. Architecture of Web application

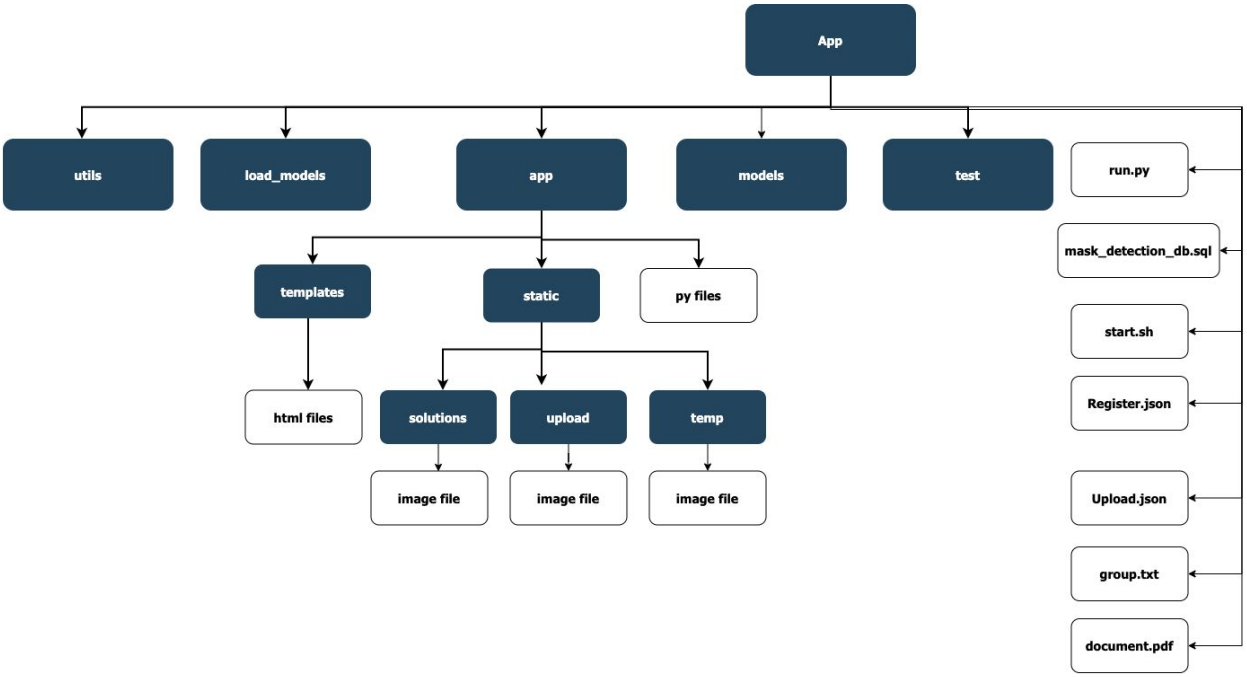

Figure 2. File Layout(darker is folder, whiter is file)

Database schema

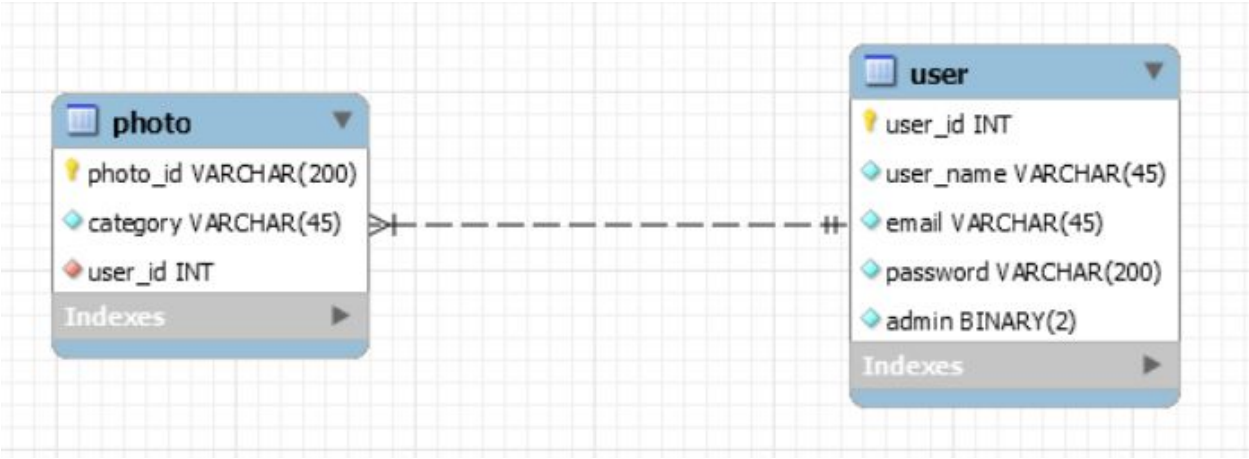

Figure 3. Database schema

The database schema is shown in Figure 3. There are two tables in our database: ‘photo’ and ‘user’. The relationship between these two tables is many-to-one. In the ‘user’ table, ‘user_id’ is a column with auto-increasing integers that index the records, which means that this is the primary key of the table. The account name as a string value is stored in the column named ‘user_name’ in ‘user’ table. The other two strings to be stored in the database are user email address and account password, and these two are put into the columns named ‘email’ and ‘password’ respectively. For the sake of security, the passwords are not stored in the clear format. As mentioned in the above, the hash of the password with a per-user salt value is stored in the database. The last column in the ‘user’ table is called ‘admin’, where it stores the binary values that make regular users and administrators distinguishable. We give 0 for administrators and 1 for regular users. In the ‘photo’ table, ‘photo_id’ is a column that stores the saving paths (string value) of the uploaded photos. Since the photo_id column indexes all the records in the ‘photo’ table, it is set to be the primary key of this table. The category of the uploaded photos are stored in the column named ‘category’ in the ‘photo’ table, and the categories include “No Face”, “No Mask”, “All Faces with Masks” and “Some Faces with Masks”. The foreign key of this table is named as user_id, which refers to the user_id in the ‘user’ table. We do this to keep track of the relationship between the table of users and the photos.

Potential problem during application deployment

After you connect to the instance and get ready to run our web app, there might be a problem. Actually, we just ran into this problem when we tried to deploy our app in the evening of Oct. 19. The terminal window just stuck and shows ‘Worker timeout’. And then we tried a few times and still didn’t work properly. There is another weird situation, that we can let the app run but the website keeps loading without responding, and sometimes we are able to log into the system but the page stuck when we tried to upload photos. All those situations happened only on the EC2 instance, and it never happened in our local machine. Our solution for that was to create a new instance and deploy our app on the new instance again. After that, our app runs smoothly. Our guess is that during that time, the server is in high need since the public AMI is shared

across the class, and thus the resource that is assigned to our web app is less than usual, which is a possible explanation why the app deployment stuck.