ECE1779 Assignment 2

# Workers

## Description

Each individual worker application is written in python and html using the flask library and it performs as an online text detector. It automatically draws rectangular boxes around the text on the photos which users uploaded. Users can explore all the photos they uploaded and compare the photos before and after text detection in pairs.

## Prerequisite

A Python 3.7 interpreter is required.

The following python packages are required in order to start the server.

* Click
* PyMySQL
* Flask
* Flask-SQLAlchemy
* Jinja2
* MarkupSafe
* SQLAlchemy
* Werkzeug
* aiofiles
* aiohttp
* async-timeout
* attrs
* chardet
* idna
* imutiles
* itsdangerous
* multidict
* opencv-python
* pip
* setuptools
* yarl
* awscli
* requests

## EC2

The worker applications are running on separate EC2 instances. Each instance is given a custom role so that it is allowed to access objects stored in the S3 bucket without specifying an access key and a secret access key. This role is immediately attached to the instance when it is created to make sure each worker instance can read and write objects in the S3 bucket.

Moreover, the worker instances are assigned to a custom security group for two reasons. First, these instances are open to public so that anyone from anywhere can create accounts and upload images. Second, accesses to the database are only granted to instances in such security group.

## Initialization

The individual worker application is initialized throughout executing the h.sh script on the Desktop. This application is designed to run at port 5000. Therefore, please make sure port 5000 is available on the client side.

## Using the application

If users are not log in, the nav bar has a login button as well as a sign-up button. Users can use the log in button to log in and use the sign-up button to sign up.

While user has logged in, nav bar has home, log out, and upload button. Log out button will delete the user information in sessions and let the user log out. Home button is directed to home page at which users can view and upload the images. On the upload page, users can upload images with ‘png’, ‘jpg’, ‘jpeg’ and ‘fig’ extensions. After uploading, web app will detect and use green rectangles to mark the text in the images. Users can view images they uploaded on the view page. By click the ‘view’ button under the images, users are able to see the original image and the processed image at same time.

## API

Tests can be done by sending API requests. You are allowed to create new users and upload images using an existing account.

The register request requires the following interface.

URL: <http://0.0.0.0:5000/api/register>

method = POST

POST parameter: name = username, type = string

POST parameter: name = password, type = string

The upload request requires the following interface.

URL: <http://0.0.0.0:5000/api/upload>

enctype = multipart/form-data

method = POST

POST parameter: name = username, type = string

POST parameter: name = password, type = string

POST parameter: name = file, type = file

## Database

Our database is powered by the MySQL engine and stored using the AWS RDS service. It can only be accessed by the AWS EC2 instances within a specific security group and all other accesses will be denied.

The database stores two pieces of information when an accounted is created. The first one is the username, which is stored in the form of a string. The second one is the password corresponding to the username. The password is salted and hashed using a one-way function so that even if two users have the same password, in the database the hashed passwords are different.

The database consists of two tables, image and user, which are connected to each other by sharing the user id. Hence, each user is only allowed to access the image he or she uploaded. Moreover, since each image is given a unique image id, different images with the same filename will be treated as different images. The detailed database structure is shown below. The uploader\_id in table ‘image’ is refer to id in table ‘user’.

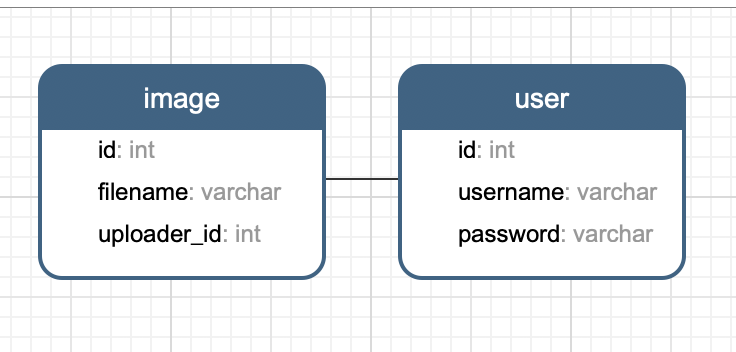


Fig 1

The application structure is shown as follows.

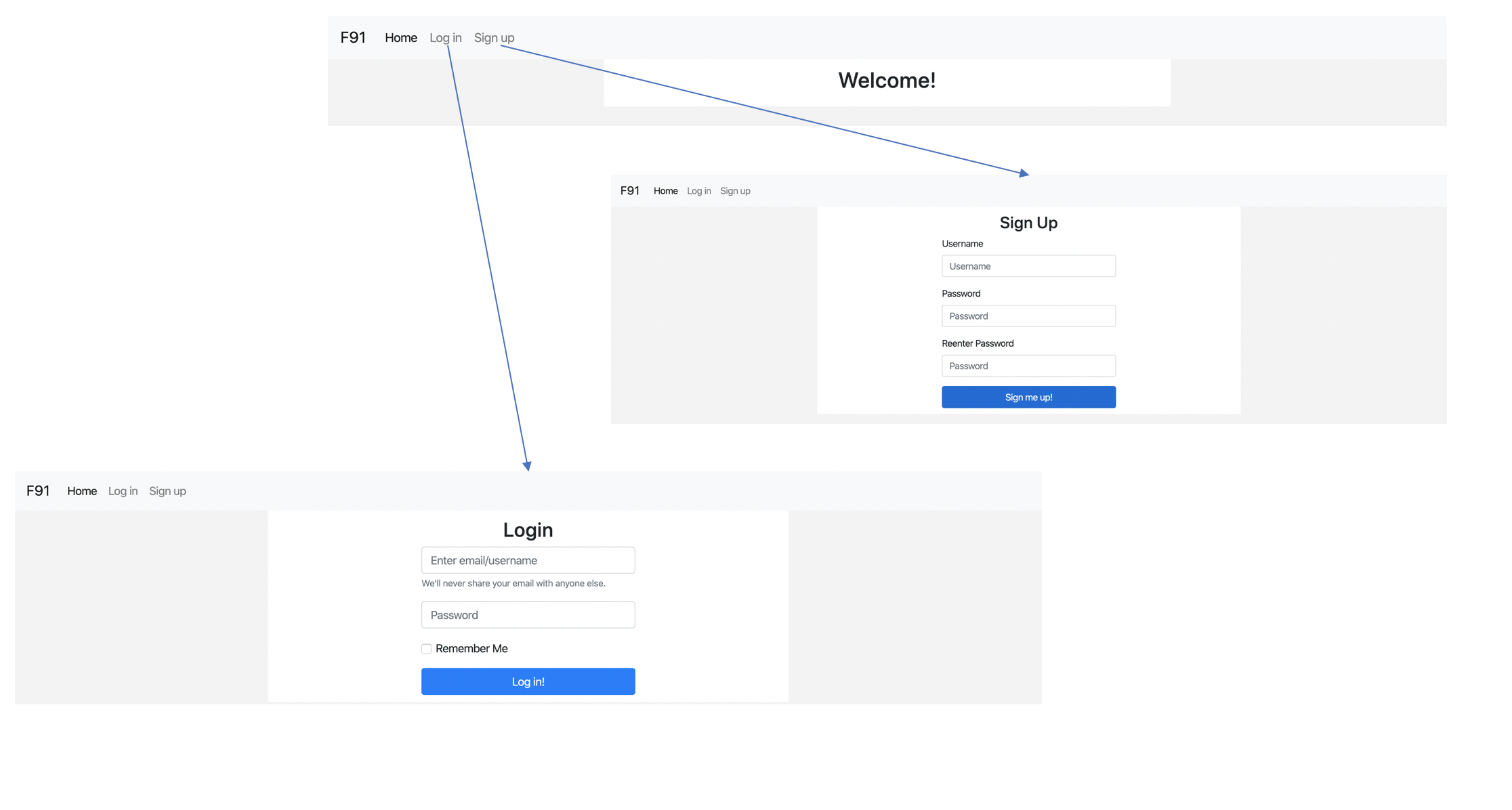


Fig 2

There are two buttons on the main page. The ‘Login’ button will direct the user to the login page while the ‘Sign up’ button will lead the user to the sign-up page. After logging in or creating an account, the user will be directed to the user page.

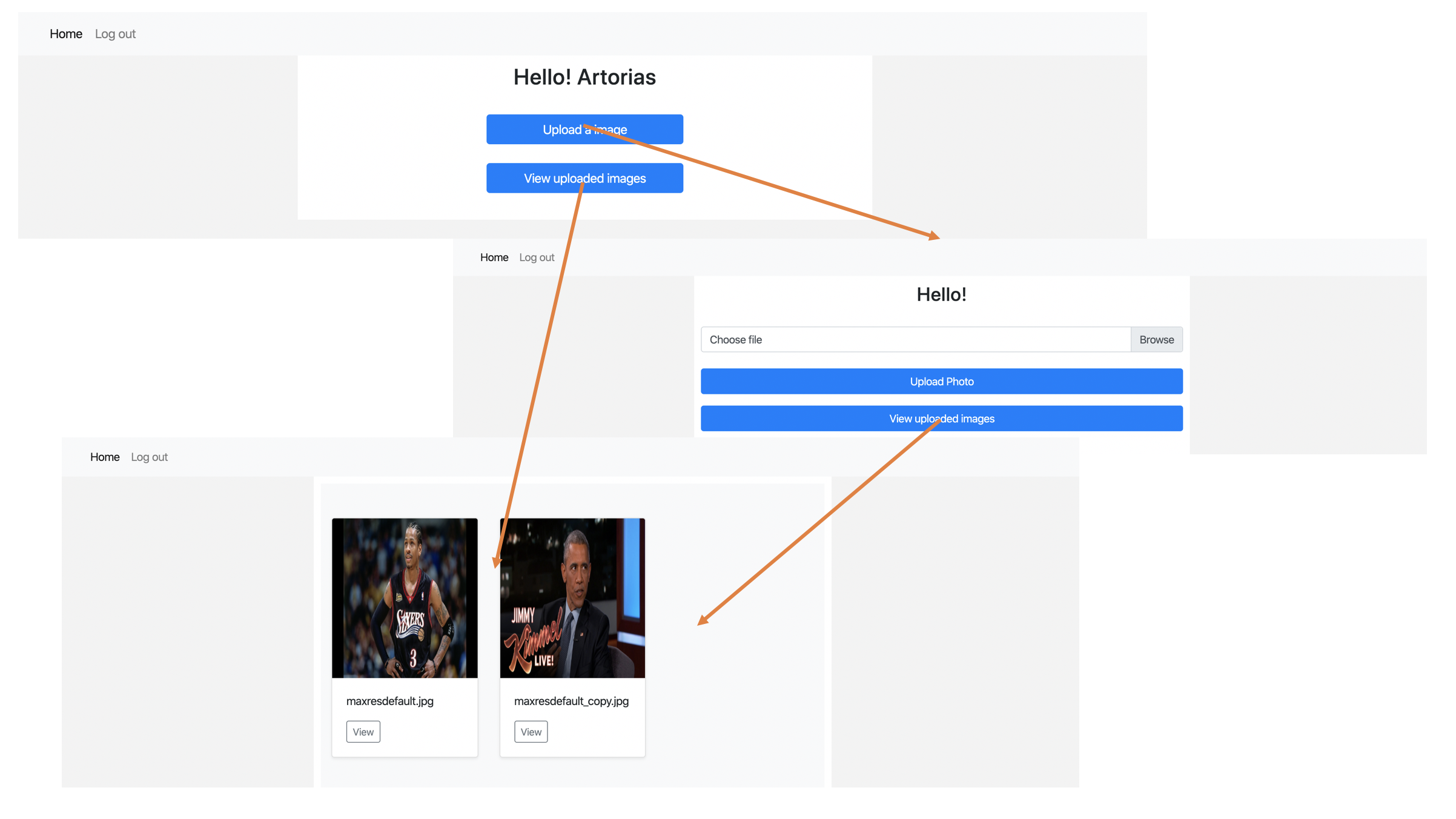


Fig 3

When clicking the ‘view’ button, users can see the difference between the original photos that they upload and the photos after text detection.

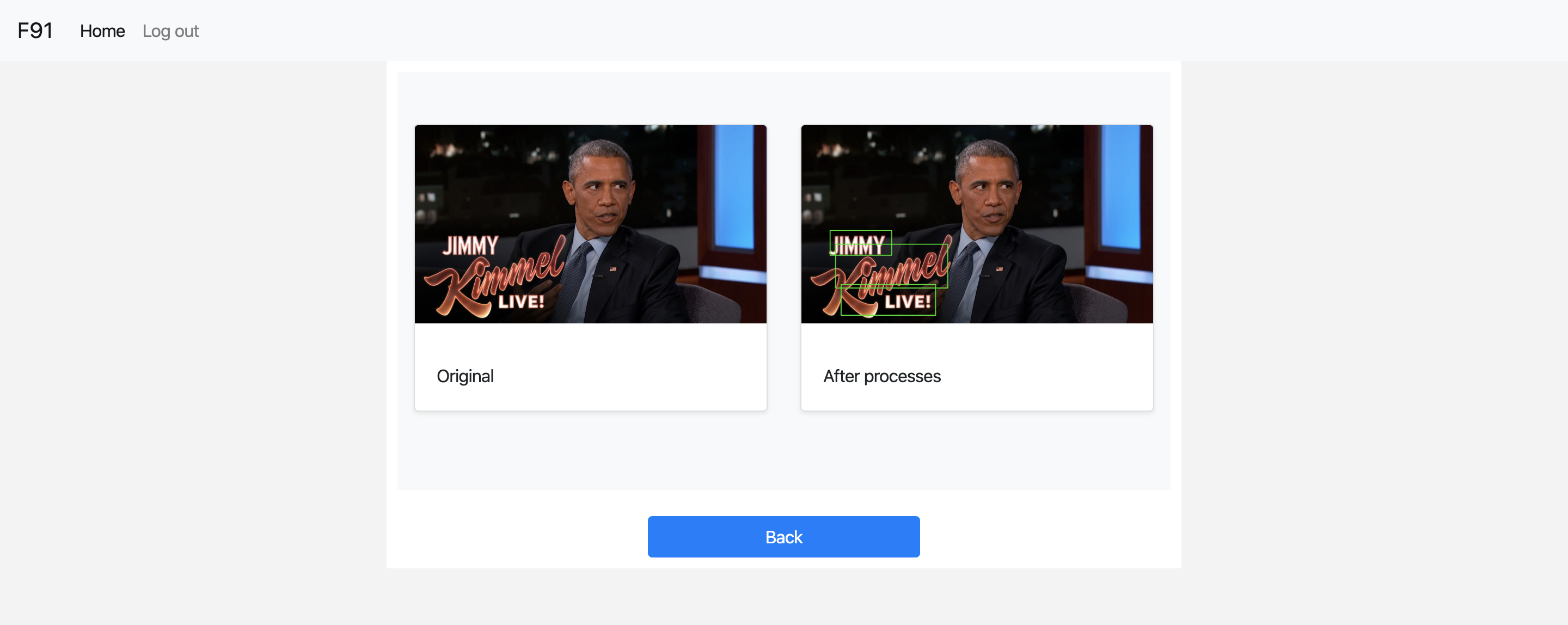


Fig 4

## Code Structure

The body of the application is packed in the “app” directory. When the application starts, “\_\_init\_\_.py” will be executed and basic parameters and settings in order to run the program will be initialized. The core of the application is the “views.py” file. This file is written to handle all the requests received from the clients, such as showing users the photos they uploaded. The structure of the database is defined in “model.py”.

A static directory is created to store all the css files as templates.

“text\_detection.py” contains the function to draw rectangles around texts in the photo. When an image is uploaded, another image with rectangles around texts will also be created and saved to the user’s directory so that he or she can see the difference.

All the html templates are stored in the “template” folder under the “app” directory. This allows us to separate the html files from the python files. Moreover, one html template can be used in several different situations.

## http request count

In order to monitor the http request of user app by AWS cloud watch service, a http request counter has been implemented in user app. The counter use **flask before\_request** decorator. Every time, before handling the http request, the **Uploadhttpcounts** function will be executed. It will increase the httpcnt by one. And send httpcnt data to AWS by using **boto3.client.put\_metric\_data** method. After sending data, the httpcnt will be set to zero. So, AWS cloud watch has the http requests data of each user app with one-minute resolution.

## S3

When a user creates an account, a directory with his or her username will be created in the S3 bucket to store the images he or she uploaded later. If the user uses clicks the delete button in the web app, the image under the corresponding directory will be removed.

The session encryption key is stored in a text file in the S3 bucket to make sure that each worker app has the same key.

## Cookies

For security purposes, cookies are used so that we can know the status of the user. When the application starts, a *session* object is created to store the login information of the user. All the data in the *session* object is encrypted using a secret key, which is stored in a text file in the S3 bucket. Therefore, users can stay logged in using any worker instances. Once the user logged in, the username will be stored in *session* for further verification. Users are required to reenter their usernames and passwords every 24 hours for the purpose of safety. They can also manually log out to drop their information in *session*. The application will automatically redirect the user to the login page if it cannot find the user’s corresponding information in *session*.

## File restriction

Users are only allowed to upload images with the extensions mentioned above. A file is verified twice before being posted on the user’s page. First, when the server receives the file, it will check the file’s extension. If the extension is not allowed, the client request will not be accepted. For instance, the server will not let a user upload a file with a pdf extension. Second, the server will check the readability of the file during image processing. If the file is unreadable, it will be rejected by the server. For example, the user can manually change the extension of a file from others to jpg, making it look an image file. However, such file cannot be read by the text detection function and hence these kinds of files are blocked from uploading.

The application only accepts files with sizes less than 10MB. Otherwise the load of the server will be too heavy.

# Manager

## Description

The manager app use boto3 to manipulate ec2 instances. In manager app, users can view a worker list which includes instanceID, type of instance, availability zone and status. In the list, by clicking the *Details* button, users can view CPU utilization and HTTP requests count rate charts with resolution of 1 minute. By clicking *Destroy* button, users can terminate the corresponding instance. The manager app also has a manage page, in which users can manually add or remove a worker.

## Access EC2 instance

The manager app is running on ec2 instance attached with the following IAM roles: *manager*, *Security groups: launch-wizard-1* and *Key pair name: keypair*. An AWS user is provided with a credentials.csv file. Using the user role, users can access the ec2 instance named manager, which includes manager and auto scaler apps.

## Initialization

An individual worker application is initialized throughout executing the h.sh script on the Desktop. This application is designed to run at port 5000. Therefore, please make sure port 5000 is available on the client side.

## App structure

run.py imports webapp from views.py and start the webapp. views.py contains decorators which is used by flask to response http requests. The views.py imports two classes: ec2tool and cloudwatch from app/helper.py. These two classes help views.py manipulate ec2 instances. Templates for flask are under /app/templates directory. The manager app is using **highcharts** and **jquery** to draw charts, which relies on highcharts.js and jquery.js under /app/static.

Ec2tool and cloudwatch are based on **boto3**. Boto3 is the Amazon Web Services SDK for Python. It provides an easy to use, object-oriented API, as well as low-level access to AWS services.

## Realtime CPU and http requests charts

The monitor app uses high chart to draw charts. The page is using ajax to request new data from sever and redraw the charts every minute.

## ec2tool main methods

### create\_default():

Create a default worker and start user app with configure:

* + - * Instance type: t2.small
      * Key pair: keypair
      * Security group: launch-wizard-1
      * IAM role: ecs3(enable ec2 access s3 and RDS)
      * Detail monitoring: enabled

The worker will be added to target group after initialized. ELB will forward http request to the target group. Thus, user can access worker via ELB’s DNS.

### destroy(id):

Terminate the instance with the given id.

### ec2GJRList():

Return a list of workers.

### removeOneWorker():

Terminate a worker instance. The total number of instances must be greater than one.

## cloudwatch methods:

### CPU\_metric:

Return [[timestamp,CPUUtilization]]. The list has 30 elements ordered by timestamp. It represents the CPUUtilization in last 30 minutes.

### http\_metric:

Return [[timestamp,httpRequestCounts]]. The list has 30 elements ordered by timestamp. It represents the http Request Counts in last 30 minutes.

See more at [user app http request](#_http_request_count).

## AWS ELB

AWS ELB forwards http requests to target group.

The DNS of AWS ELB is: <http://manager-2106785547.us-east-1.elb.amazonaws.com/>

## Examples of manager app

Fig 5: real time CPU utilization and http request charts.

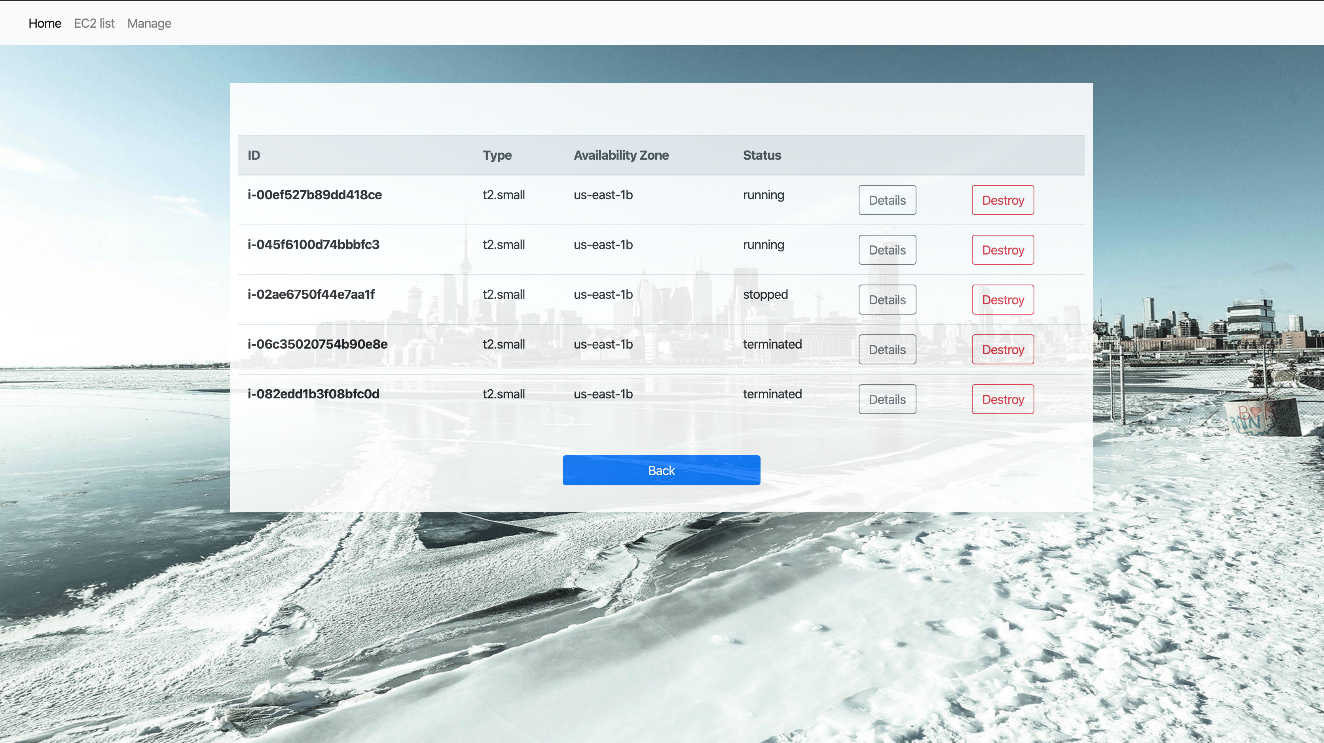


Fig 6: Instances list/ Worker pool



Fig 7: Manage page. Clicking new instance will create a new worker. Clicking delete an instance will terminate a worker and remove it from worker pool.

## **Auto-scaler**

Due to the restriction of AWS cloud watch services’ resolution, the auto-scaler is naïve. An up to two minutes delay of monitoring significantly influence the strategy of auto-scaler.

Normally, the auto-scaler depends on AWS cloudwatch’s CPU utilization data. If the average utilization of all workers is greater than 80, then the size of worker pool will be increased by 1/3 of size. Ideally, each worker’s CPU load will be reduced by 25%. Due to the latency of monitoring, we don’t want to increase the worker pool when CPU utilization actually goes above 80. Thus, if the CPU utilization is above 60 and number of workers is more than 3, auto-scaler will increase the pool by one. If the CPU utilization is below 30, auto-scaler will decrease the pool by 1/3 of the size of pool.

Auto-scaler will not execute expand operation twice in 3 minutes. Also, it will not execute the shrinking operation twice within 5 minutes. This is because of the delay of monitoring CPU utilization. Without these restrictions, the auto-scaler cannot converge smoothly.

However, when worker instances face an unexpected large load, the user app might crash. ELB won’t forward http request to those unhealthy targets anymore. Therefore, other instances will have to deal with more request, which would make them more likely to shut down user apps. The consequence of large unexpected load is all workers’ user app being shutted down. To avoid this, auto-scaler will check if the user app is still running. If not, auto-scaler will reboot the instance and restart the user app. Thus, auto-scaler is able to notice the load increasing by check the status of the user apps. Since auto-scaler checks the user apps’ status every 10s, the reboot alarm is more sensitive than AWS cloudwatch. When reboot happens, auto scaler will increase the size of worker pool by 1/3. It takes about one minute to initialize a user app, causing the auto-scaler to have a limitation that the expand operation cannot be executed twice within 2 minutes.