Facial Recognition Security System
Final Report

INFO 4290 A10
Integration Project II

Philip Intile – 100335192
Stephen Dereniowski – 100319110
Davis Fulton – 100228199
Sunveer (Sunny) Sandhu – 100308494

August 16th, 2021

Table of Contents

Abstract

In a traditional corporate security setting, access control is likely handled by some form of badge. These access badges are likely equipped with a Radio-frequency identification (RFID) tag, barcode, or magnetic stripe that can be combined with a reader at each secure door to provide and verify access to employees. Access control badges such as these have numerous potential downsides including being unable to fully verify that the correct user is in possession of the card, a small form factor that can lead to easy loss or theft, and the inconvenience of having to carry the card around.

A potential solution for the inconveniences of traditional badge-based access control is the implementation of a Facial Recognition Security System (FRSS). A FRSS would allow recognized employees to be stored within a secure database, and quickly verified when approaching a secure door to negate the need to carry a badge with them. Using a FRSS would also guarantee that potential bad actors would be unable to access secure areas through taking advantage of lost or stolen access badges.

To determine the best way to implement a FRSS, secondary research was conducted to analyze how existing systems for facial detection and facial recognition work. Additionally, research was done on a wide variety of software, tools, and utilities to determine the best methods of hosting and accessing a database, copying files from a virtual machine to a server, and observing a directory using Python. Using existing qualitative data and secondary research allowed for the most efficient planning and development of the FRSS, as it was possible to learn from the successes and mistakes of other systems.

Through the complete development of the FRSS, the aim was to allow a company to provide access control to employees for specific locations, such as a room, or building. Additionally, the company should be able to monitor this access using access logs in something like a database. The access would be enabled by client computers with a camera attached that would detect and capture an image of a user's face, and send it to a remote server to be processed for recognition. This would allow for a scalable client-server model, where the majority of the complex and secure work would take place on the server. As such, a user would not need any external keys or hardware like an access badge, and would only need to walk up to a door in order to be allowed entry to secured areas.

Ultimately, it was determined that software including Python, OpenCV, phpMyAdmin, MariaDB, and Watchdog would allow for the ideal implementation of the FRSS. Development would be done using Ubuntu, with the client machine existing as a virtual machine, and the server machine existing as an instance on the Google Cloud Platform. Implementing the FRSS this way allowed for simple access to future scalability and support. The following report will describe the detailed design and development process of the completed FRSS.

Introduction

Aim

The aim of this report is to detail the background work, research, design, development, and complete implementation of the Facial Recognition Security System (FRSS). As a whole, this system was developed to provide ease of use and access to employees, while remaining a secure and monitorable system that can be used for the protection of a company's secure internal assets. In addition, the implementation of the FRSS aims to test the hypothesis that a FRSS may be a more convenient and secure solution than traditional badge-based access control systems.

Objectives

The objectives of this project are split into two main categories - project objectives, and design objectives.

The project objectives are as follows:

- This report should primarily serve the purpose of detailing the design, development, and implementation of the FRSS
- The FRSS should take into account both physical and digital security
- The FRSS should make use of thoroughly researched facial recognition libraries
- The FRSS should be thoroughly tested prior to project completion to ensure reliability and robustness
- The FRSS should have complete project documentation to aid learning and future development

The design objectives are as follows:

- The FRSS should be capable of recognizing a face in under 5 seconds
- The FRSS should be capable of recognizing multiple, unique faces
- The FRSS should be integrated with a password-protected database
- The secure database should store access requirements and logs
- The FRSS should have a secure front-end for administrators for the provision of access to specific areas
- The FRSS should be able to receive images from remote machines and return access information in under 5 seconds
- The FRSS should have complete project documentation to aid learning and future development

The remainder of this report will fully detail the implementation and fulfillment of these objectives.

Background

Research Methods

In order to effectively develop an application that takes advantage of facial recognition, it was important for effective research to be carried out. In order to develop the FRSS, a significant amount of secondary research for qualitative data and information was done. Due to the short time constraints of this project, primary research would not be feasible, and quantitative data would not be quite as useful when compared to qualitative data. Ultimately, being able to learn from the successes and mistakes of other developers and researchers would allow for the most efficient path to success for this system.

Initially, some research was done into basic implementations of facial detection using OpenCV, which eventually progressed to basic implementations of facial detection. Typically these systems would be something like one short, single-task Python script. As development progressed, research was done to understand implementations of MySQL, which eventually led to the XAMPP stack, which contained phpMyAdmin and MariaDB (MySQL).

From there, the majority of research completed was for distinct tasks in the development process. These tasks included subjects like how to copy a file from a virtual machine to a server, how to observe a directory in Python, regular expression syntax, database connections in Python, and more. Typically, these short bursts of research would lead to program documentation, which proved to be valuable over the course of the implementation. As these technologies are rapidly changing, documentation pages were constantly one of the better sources of information during the course of development.

Overall, this secondary research for qualitative information proved to be the most useful method.

Literature Review

How does facial recognition work?

As technology undergoes exponential growth as the years go on, there are always new and improved types of technology that are being created and introduced to the world. In recent years, one of the biggest forms of this is facial recognition. This type of technology is very powerful and can be used in many ways. Prominently, if you have a recent smartphone, you experience the use of this every time you pick it up. Once the phone turns on it immediately scans your face and determines if you are the owner of it. If you are, it unlocks and grants you access. This is done in the matter of seconds as the technology has improved and is being improved as the years go on.

Simply put, it is "a subfield of computer vision research focused on building software systems that can analyze the similarity between faces in images and video"(Crumpler & Lewis, 2021). In their paper, Crumpler and Lewis state that although facial recognition is mostly automated, in the end it is a system that is updated and adjusted by humans based on the accuracy levels and efficiency of the system. Along with that comes the implication that the data is being used for illicit purposes by companies, which will be discussed further.

But how does it work? Nowadays, facial recognition is used with deep learning AI. This AI uses artificial neural networks to process its data. When a picture or video is being monitored, the system is "capable of transforming face images into numerical expressions that can be compared to determine similarity"(Crumpler & Lewis, 2021). This "template" is then used to compare with other images that are alike to determine a confidence level. The more images that the system learns, the more confident that it becomes at identifying the correct individual. In a security system it is crucial for the system to be flawless so that it does not grant access to the wrong person. Advances in deep learning AI and algorithms have made it so this form of technology is one of the most secure ways to identify people.

As previously stated above, accuracy of a facial recognition system is the most crucial aspect of it. Crumpler and Lewis state that "there is no single value that provides a complete picture of a facial recognition's "accuracy"" (Crumpler & Lewis, 2021) which means that it is hard to compare what different companies define as accurate. A system used by one company could be identifying different aspects of the face than another, therefore making their accuracies different in their own way. Not every system can be 100% accurate since there are two errors that can be created by the system. They state that this is because "there are two different kinds of mistakes that a facial recognition system can make"(Crumpler & Lewis, 2021). These two mistakes are:

1. False negative
    a. Occurs when the system rejects two images of the same person and says they do not match
2. False positive
    a. Occurs when the system accepts two images from different people as the same person

To show how advanced the systems have become, Crumpler and Lewis cite a study that was done in 2013 and 2021. This study was completed by the National Institute of Standards and Technology (NIST) with their Face Recognition Vendor Test (FRVT). This test was done by feeding over a million mugshots into their system and seeing what the accuracy of the returned values were. In 2013, “the best algorithm returned the wrong photo 4.1 percent of the time” (Crumpler & Lewis, 2021) whereas in 2021, “the leading algorithm... returned the wrong photo less than 0.1 percent of the time” (Crumpler & Lewis, 2021). This test proved that as time goes on the advances in the technology are becoming much more accurate and efficient. Over those eight years, facial recognition systems and algorithms underwent massive upgrades. Crumpler and Lewis state, though, that “it is important to note that this degree of accuracy is contingent on a number of factors” (Crumpler & Lewis, 2021). These factors include:

- Algorithm being used
- Quality of images
- Size of search space

The purpose of that study had pictures that had good lighting and consistency in the position of the faces. To go from 4.1 percent to 0.1 percent, however, is a feat that is impressive and holds good value for the future of accuracy of these types of systems.

In conclusion, the report by Crumpler and Lewis was a great way to gain initial knowledge of how facial recognition systems work. It gave great insight into the technical aspects of them and allowed for a greater understanding of them as time went along. They covered everything from what they are, how they work, and their accuracy. However, they also covered comparison thresholds which were not covered by the system implemented in this report. Their examples showed how the systems compared dogs to food and how accurate the system determined which is which. For the purpose of this project, there was no comparison testing in that manner and although it was not used it was beneficial to learn that side of facial recognition systems as well.

Design

Development Approach

The development of the FRSS took place in the form of an iterative software development lifecycle. Specifically, it closely followed the ‘Rational Unified Process’ (RUP), which was created by Rational, a division of IBM (Christensson).

RUP divides development into four phases, each of which include all aspects of the system development life cycle. The four phases are:

1. Inception
    - Project idea determined, initial research
2. Elaboration
    - Architecture and resources researched, possible applications and costs evaluated
3. Construction
    - Design, development, testing
4. Transition
    - Software released, final adjustments made

(Christensson)

Following this system allowed for an incredibly iterative approach, where it was possible to adapt the use-cases of the FRSS and change course in the event of major barriers. It also allowed for any problems to be caught very early in the process, and allowed for new ideas to form that were significant improvements to the design.

By starting initial development in the inception phase, it was possible to quickly determine the direction development should take for the FRSS, and what would be possible given current technology for facial recognition.

In the elaboration phase, determining the hardware and software resources available alongside their associated costs allowed for an early definition of the constraints for that aspect of the project.

The construction phase contained the heavy development processes for the FRSS. This phase involved constant analysis, design, development, and testing. In this phase, a significant portion of the required software and modules were installed and implemented into the code.

Finally, the transition phase involved some of the final changes like uploading the final code to the server, ensuring the correct credentials and addresses were used, and making some final progressions in development and testing.

These rapidly iterative, but firm phases allowed for the most efficient use of time across the short duration of the design, development, and implementation of the FRSS.

System Overview

Use Case Diagram

Use cases are used during requirements elicitation and analysis to represent the functionality of the system. An actor describes any entity that interacts with the system. The identification of actors and use cases results in the definition of the boundary of the system. The actors are outside the boundary of the system, whereas the use cases are inside the boundary of the system (Bruegge & Dutoit).

The two actors in the FRSS are the Employee and System Administrator. All the employee will do is look into the camera and have the face recognised. The Systems Administrator has two use cases. They interact both with a Command line Interface (CLI), and the back end of the system. Included in these use cases are several functions that a System Administrator will perform such as viewing access logs, running scripts, and modifying the source code.

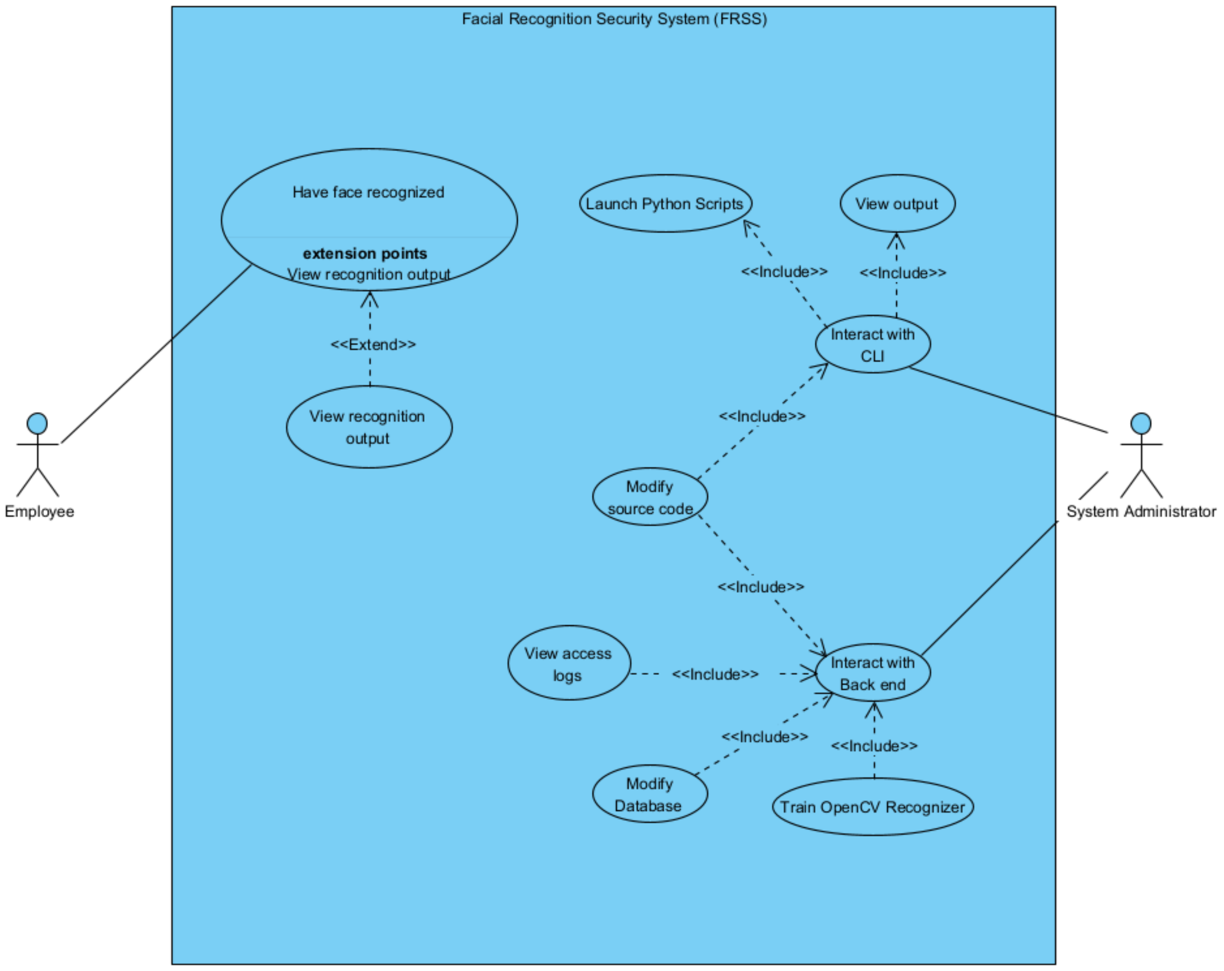


Fig. 1

Class Diagram

A Class diagram represents the structure of the system. A class diagram is also used during object design to specify the detailed behavior and attributes of classes. A Class has a mandatory name, optional attribute, and optional operations (Bruegge & Dutoit).

The important aspects of class diagrams are the relationships, associations, navigability, and multiplicity. A relationship (shown as a line) represents a connection between the two classes. Association is a verb-based action between two classes. Navigability is how one class communicates with other classes. Multiplicity is the number of objects that can participate in an instantiated relation.

The Relationships, Association, Navigability, and Multiplicity for the FRSS are as follows:

- One to many employees can have access to One to many locations.
- One location will generate 1 to many access logs.

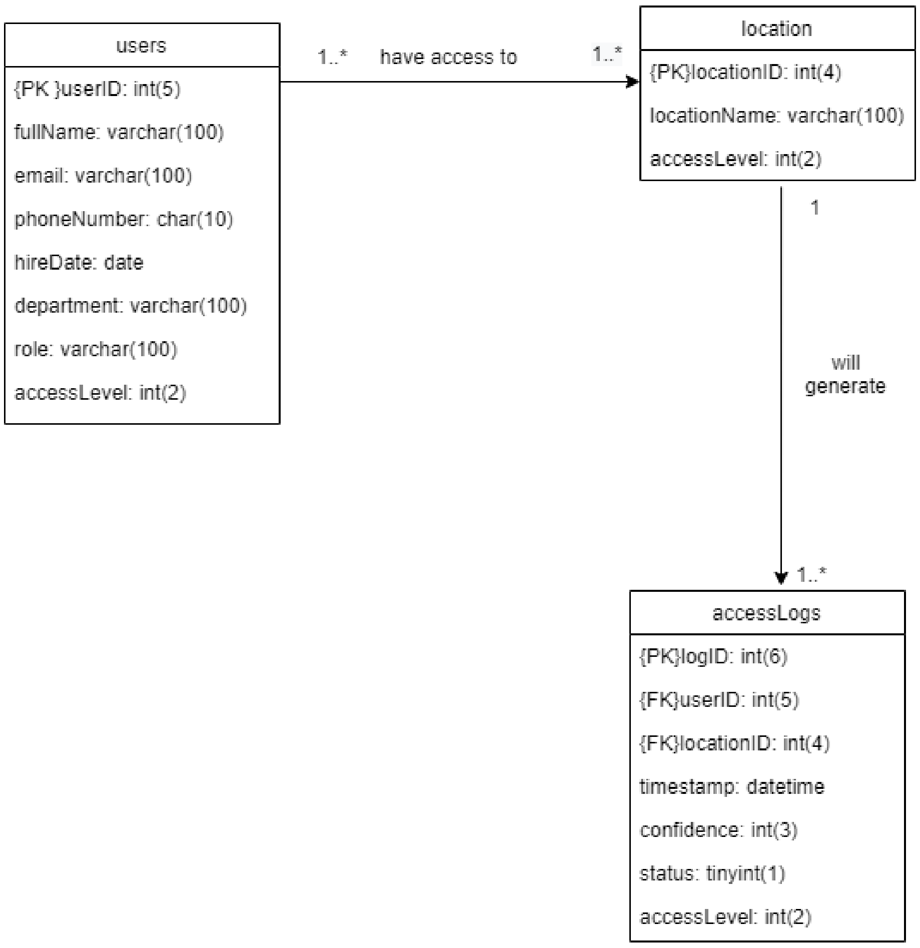


Fig. 2

Sequence Diagram

A sequence diagram describes the flow of messages, events, and actions between objects. This is typically used during analysis and design to document and understand the logical flow of data in the system (Bruegge & Dutoit). At the very top you will see a stick figure, which is called an actor, just like in the use case diagram. The three light blue rectangular boxes next to the actor are called "objects". The FRSS has three objects: the Client (VM), Server (GCP Instance), and Database (GCP Instance). The arrows represent the flow of data. Each arrow has its own numbered text that is used to describe what is happening at that moment. The very first step in this sequence is launching frss_recognition.py on the Server (GCP Instance).

The section "*FRSS System Design*" of this report goes into detail about the flow of messages and events.

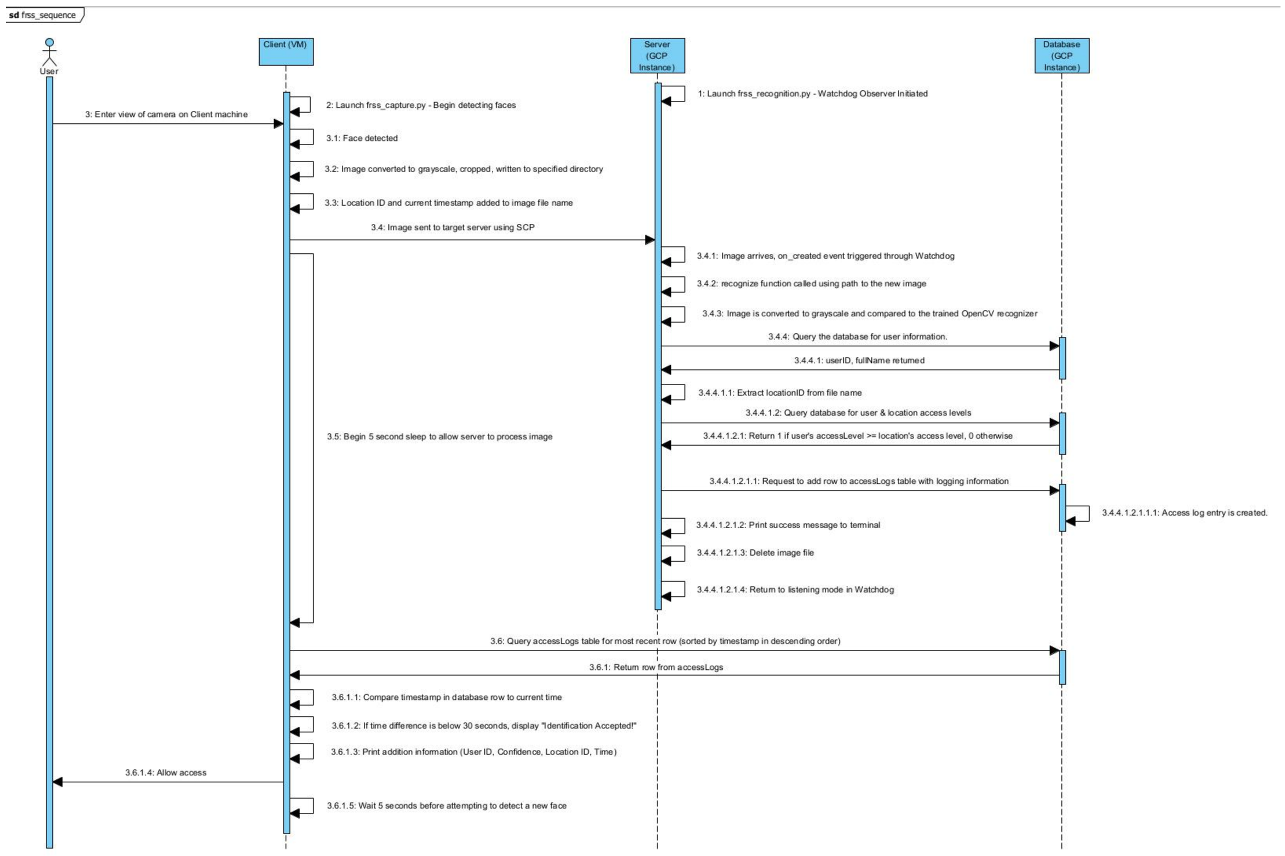

sd frss_sequence
User
Client (VM)
Server (GCP Instance)
Database (GCP Instance)
1: Launch frss_recognition.py - Watchdog Observer Initiated
2: Launch frss_capture.py - Begin detecting faces
3: Enter view of camera on Client machine
3.1: Face detected
3.2: Image converted to grayscale, cropped, written to specified directory
3.3: Location ID and current timestamp added to image file name
3.4: Image sent to target server using SCP
3.4.1: Image arrives, on_created event triggered through Watchdog
3.4.2: recognize function called using path to the new image
3.4.3: Image is converted to grayscale and compared to the trained OpenCV recognizer
3.4.4: Query the database for user information.
3.4.4.1: userID, fullName returned
3.4.4.1.1: Extract locationID from file name
3.4.4.1.2: Query database for user & location access levels
3.5: Begin 5 second sleep to allow server to process image
3.4.4.1.2.1: Return 1 if user's accessLevel >= location's access level, 0 otherwise
3.4.4.1.2.1.1: Request to add row to accessLogs table with logging information
3.4.4.1.2.1.1.1: Access log entry is created.
3.4.4.1.2.1.2: Print success message to terminal
3.4.4.1.2.1.3: Delete image file
3.4.4.1.2.1.4: Return to listening mode in Watchdog
3.6: Query accessLogs table for most recent row (sorted by timestamp in descending order)
3.6.1: Return row from accessLogs
3.6.1.1: Compare timestamp in database row to current time
3.6.1.2: If time difference is below 30 seconds, display "Identification Accepted!"
3.6.1.3: Print addition information (User ID, Confidence, Location ID, Time)
3.6.1.4: Allow access
3.6.1.5: Wait 5 seconds before attempting to detect a new face

Fig. 3

Design Constraints

During both the planning and implementation phases for the FRSS, a number of design constraints surfaced. In terms of hardware, it was only possible to confidently design the system for currently possessed hardware. Similarly, on the software side, it was decided that the focus would be on making the system work as flawlessly as possible on Ubuntu only. With some adaptation, it is almost certainly possible this system could work on a Windows system. While untested, the FRSS could also likely work well on other Linux distributions given the same set of software prerequisites.

Despite some original plans to create a user interface for the system, the development of the project ended up leading to the determination that a user interface was not necessarily useful or a requirement for functionality. Given additional time however, it could be a worthwhile addition. Given the user interface constraints, some compromises were made in the original plans of having a box to highlight the user's face, and having a somewhat custom system to administrate users. Parts of these features were however implemented into the FRSS using pre-existing solutions such as phpMyAdmin.

System Architecture

The FRSS consists of two major components - a virtual machine, and a Google Cloud instance.

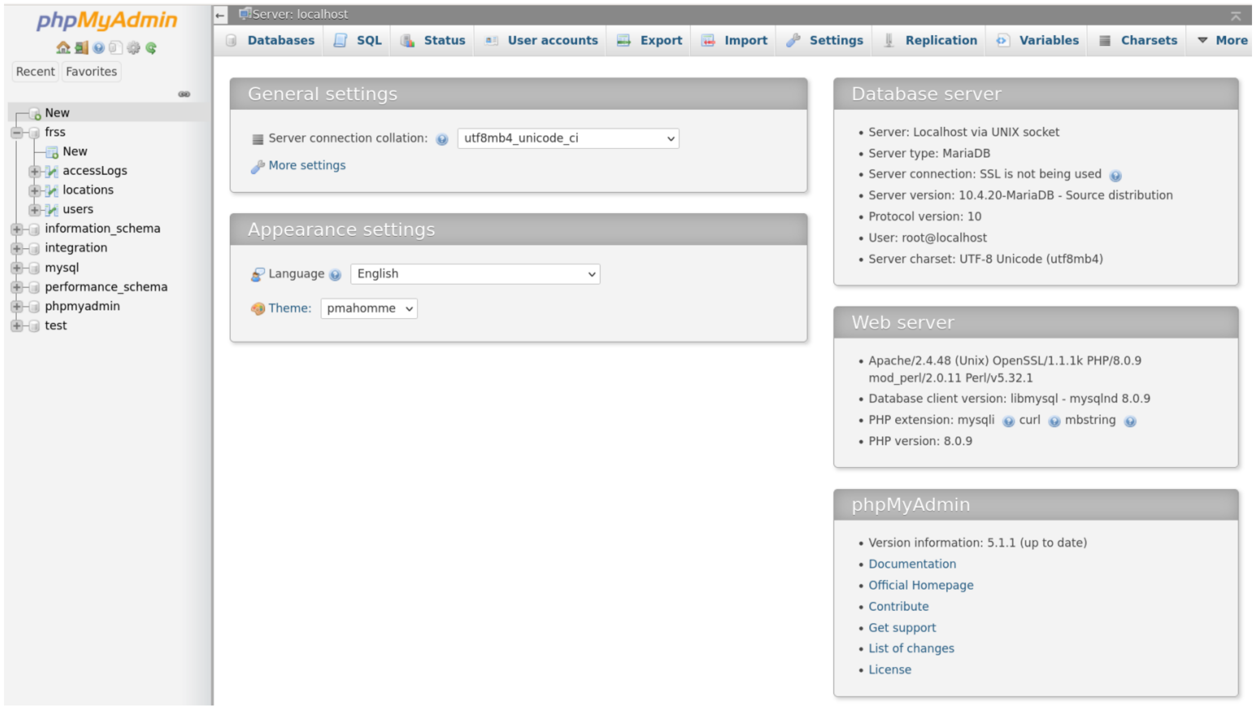
The virtual machine consists of an Ubuntu 20.04.2 LTS machine using an 8-core processor, 8 GB of memory, and 20 GB of hard disk space. The code being executed on the virtual machine is not incredibly resource intensive, so the specifications of this machine do not reflect the minimum hardware requirements.

The Google Cloud instance consists of a single 'e2-standard-4' virtual machine instance. This instance has 4 vCPUs, and 16 GB of memory. It also contains 50 GB of persistent solid-state storage. Once again, this instance is highly overpowered for what is required of it, but due to cost, ease of use, and speed, it was a desirable choice for development.

In terms of the FRSS design, the virtual machine acts as a sender and receiver, while the cloud instance acts as a processor.

Database Design

The FRSS Database is hosted using phpMyAdmin version 5.1.1, and MariaDB version 10.4.20.



General settings

Server connection collation: utf8mb4_unicode_ci

More settings

Appearance settings

Language English

Theme: pmahomme

Database server

- Server: Localhost via UNIX socket
- Server type: MariaDB
- Server connection: SSL is not being used
- Server version: 10.4.20-MariaDB - Source distribution
- Protocol version: 10
- User: root@localhost
- Server charset: UTF-8 Unicode (utf8mb4)

Web server

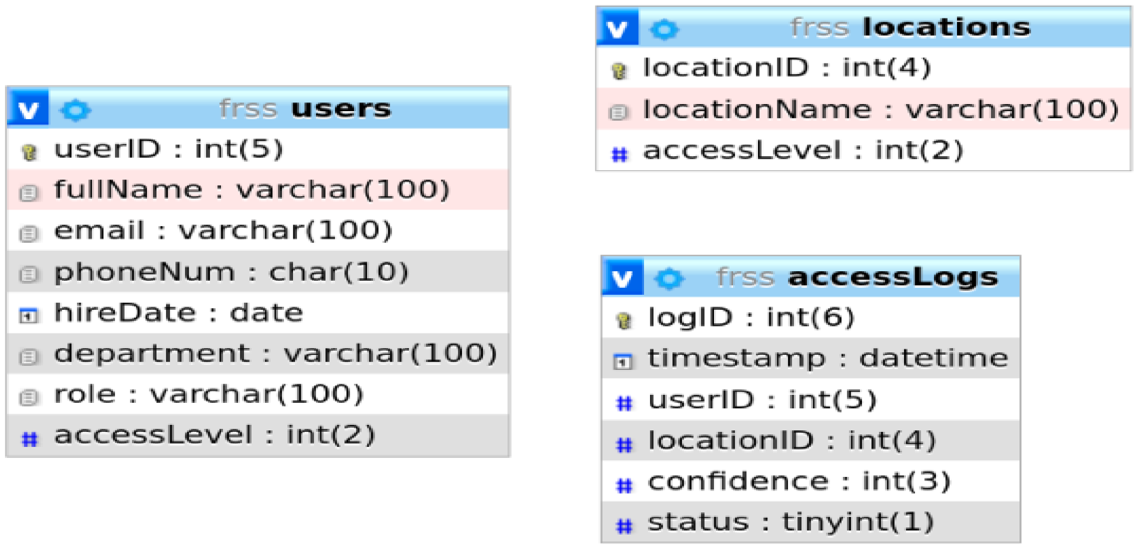
- Apache/2.4.48 (Unix) OpenSSL/1.1.1k PHP/8.0.9 mod_perl/2.0.11 Perl/v5.32.1
- Database client version: libmysql - mysqlnd 8.0.9
- PHP extension: mysqli curl mbstring
- PHP version: 8.0.9

phpMyAdmin

- Version information: 5.1.1 (up to date)
- Documentation
- Official Homepage
- Contribute
- Get support
- List of changes
- License

Fig. 4

The FRSS has three tables in the database: users, locations, and accessLogs. Each table includes a set of fields with specific constraints. Each table has a primary key that takes form a unique integer ID. The remaining fields store useful data that is essential in the functionality of the FRSS.



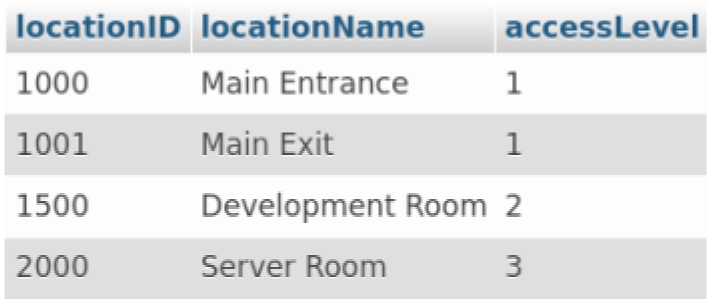
frss **users**
- userID : int(5)
- fullName : varchar(100)
- email : varchar(100)
- phoneNum : char(10)
- hireDate : date
- department : varchar(100)
- role : varchar(100)
- accessLevel : int(2)

frss **locations**
- locationID : int(4)
- locationName : varchar(100)
- accessLevel : int(2)

frss **accessLogs**
- logID : int(6)
- timestamp : datetime
- userID : int(5)
- locationID : int(4)
- confidence : int(3)
- status : tinyint(1)

Fig. 5

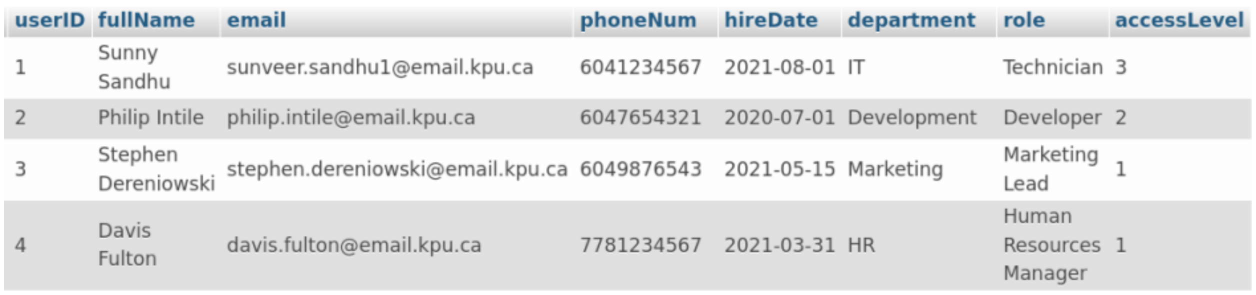
Each location has a unique locationID, a descriptive name, and an access level. When a user tries to enter a location, their access level is compared to the access level of the location.



locationID	locationName	accessLevel
1000	Main Entrance	1
1001	Main Exit	1
1500	Development Room	2
2000	Server Room	3

Fig. 6

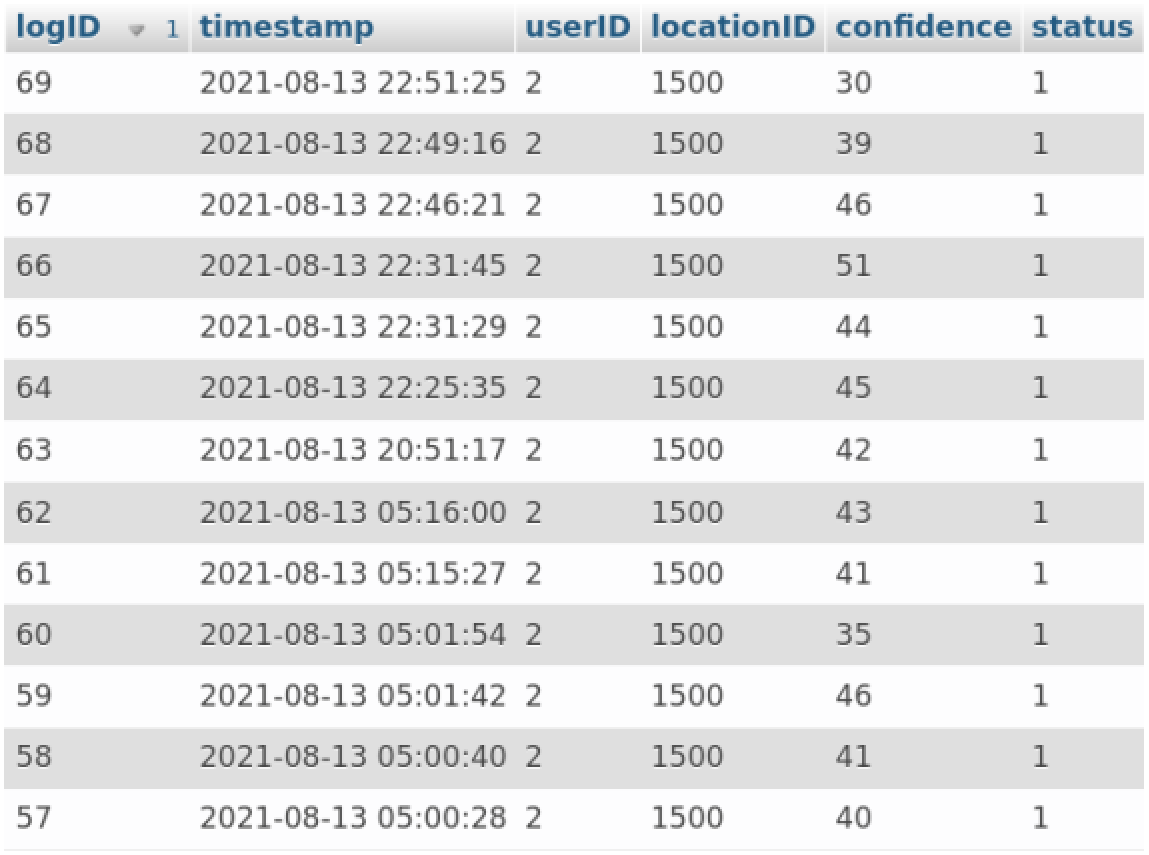
As the FRSS is targeted towards businesses, the user table is populated with relevant employee information. The FRSS could easily be integrated with an existing business database. Each user's access level is pre-determined based on company policies that cover the level of access to secured locations that the user requires. This is a security precaution to prevent employees from entering areas that they are not authorized to enter, like for example, an HR employee trying to enter a server room.



userID	fullName	email	phoneNum	hireDate	department	role	accessLevel
1	Sunny Sandhu	sunveer.sandhu1@email.kpu.ca	6041234567	2021-08-01	IT	Technician	3
2	Philip Intile	philip.intile@email.kpu.ca	6047654321	2020-07-01	Development	Developer	2
3	Stephen Dereniowski	stephen.dereniowski@email.kpu.ca	6049876543	2021-05-15	Marketing	Marketing Lead	1
4	Davis Fulton	davis.fulton@email.kpu.ca	7781234567	2021-03-31	HR	Human Resources Manager	1

Fig. 7

The logID and timestamp are automatically generated by the database when a new row is inserted, while the remaining columns are populated by the FRSS itself. The userID represents the ID of the user requesting access to a location. The locationID is the location they are trying to access. Confidence represents the system's confidence that its recognition is correct, formatted as a percentage value. Finally, the status column determines whether the user has the appropriate access level for the requested location (1 if true, 0 if false). At logID number 69, the FRSS had 30% confidence that user #2 was recognized. Given user #2's access level of 2, and location ID #1500's access level of 2, the system approved access, and updated the status column with a value of 1.



logID ▾ 1	timestamp	userID	locationID	confidence	status
69	2021-08-13 22:51:25	2	1500	30	1
68	2021-08-13 22:49:16	2	1500	39	1
67	2021-08-13 22:46:21	2	1500	46	1
66	2021-08-13 22:31:45	2	1500	51	1
65	2021-08-13 22:31:29	2	1500	44	1
64	2021-08-13 22:25:35	2	1500	45	1
63	2021-08-13 20:51:17	2	1500	42	1
62	2021-08-13 05:16:00	2	1500	43	1
61	2021-08-13 05:15:27	2	1500	41	1
60	2021-08-13 05:01:54	2	1500	35	1
59	2021-08-13 05:01:42	2	1500	46	1
58	2021-08-13 05:00:40	2	1500	41	1
57	2021-08-13 05:00:28	2	1500	40	1

Fig. 8

FRSS System Design

The FRSS currently utilizes a client-server model to carry out its security functions. In its current form, the client exists as a virtual machine hosted on a computer, but could easily exist as a standalone machine. The server is an Ubuntu instance hosted on Google Cloud. This model is used both for security purposes, and so that the clients can be as light as possible while the server does the heavy lifting of facial recognition and database updates.

The client (virtual machine) requires the following software and modules to be installed:

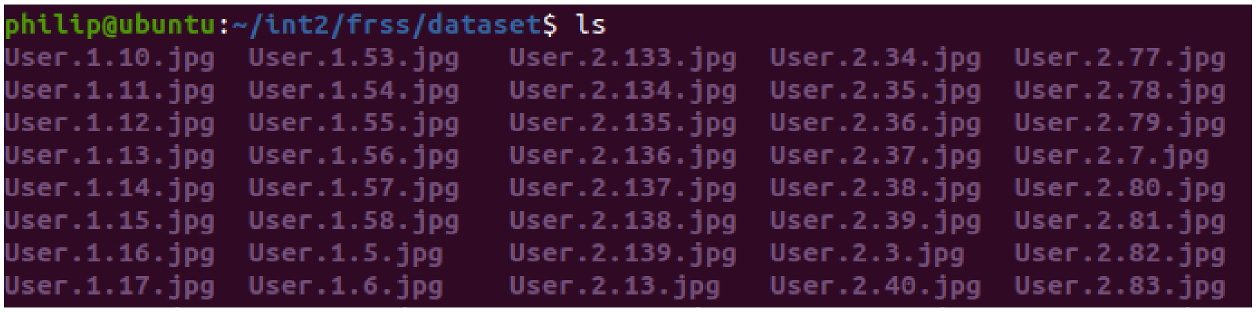
- Python 3.8.10
- OpenCV (Python library for facial recognition)
- Numpy (Library for match calculations - used to train the facial recognition data)
- Pillow (Python Imaging Library)
- Pytz (Library used for timezone objects in Python)
- Dateutil (Library used for parsing date strings in Python)
- MySQL Connector (Used to connect to MySQL database)

The server (Google Cloud instance) requires the following software and modules to be installed:

- Python 3.8.10
- XAMPP (cross-platform, Apache, MySQL, PHP, Perl)
    - This stack manifested as phpMyAdmin, and MariaDB in the FRSS
- OpenCV
- MySQL Connector
- Watchdog (Used to observe a directory for events and allow for action to be taken)

The first step in implementing the FRSS is to gather photos of the system's users. With these photos gathered and appropriately named with the unique ID of each user, the system can be trained. This is what allows OpenCV to recognize a face. With an appropriately trained model, recognition will have a higher level of confidence, which will in turn directly improve the overall security of the system.

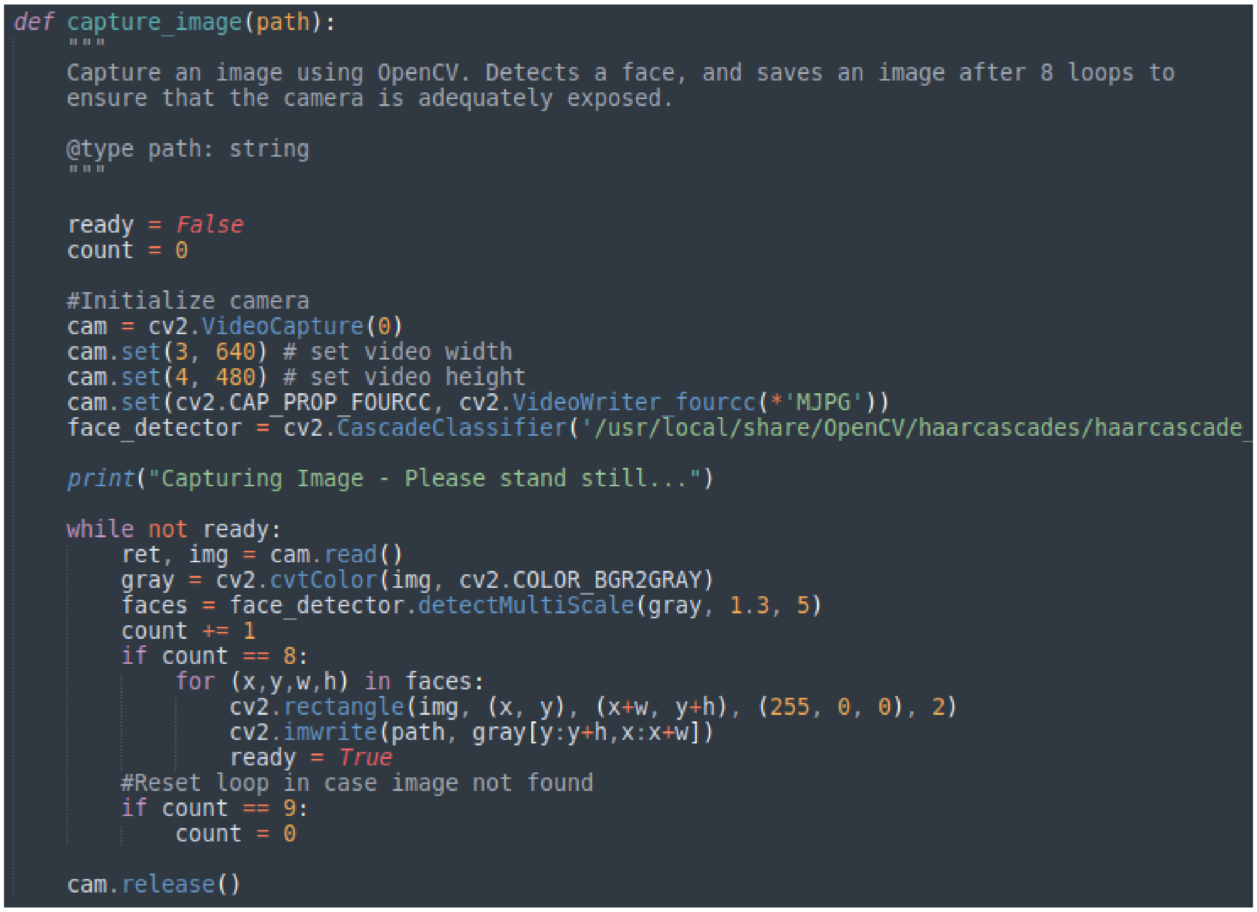
Face photos can be captured at the client level, or be imported onto the machine. Using OpenCV, those images are used to train a model and create a 'trainer.yml' file. Once created, this trainer can be relocated to the server where the actual facial recognition takes place.



```
philip@ubuntu:~/int2/frss/dataset$ ls
User.1.10.jpg  User.1.53.jpg   User.2.133.jpg  User.2.34.jpg  User.2.77.jpg
User.1.11.jpg  User.1.54.jpg   User.2.134.jpg  User.2.35.jpg  User.2.78.jpg
User.1.12.jpg  User.1.55.jpg   User.2.135.jpg  User.2.36.jpg  User.2.79.jpg
User.1.13.jpg  User.1.56.jpg   User.2.136.jpg  User.2.37.jpg  User.2.7.jpg
User.1.14.jpg  User.1.57.jpg   User.2.137.jpg  User.2.38.jpg  User.2.80.jpg
User.1.15.jpg  User.1.58.jpg   User.2.138.jpg  User.2.39.jpg  User.2.81.jpg
User.1.16.jpg  User.1.5.jpg    User.2.139.jpg  User.2.3.jpg   User.2.82.jpg
User.1.17.jpg  User.1.6.jpg    User.2.13.jpg   User.2.40.jpg  User.2.83.jpg
```

Fig. 9

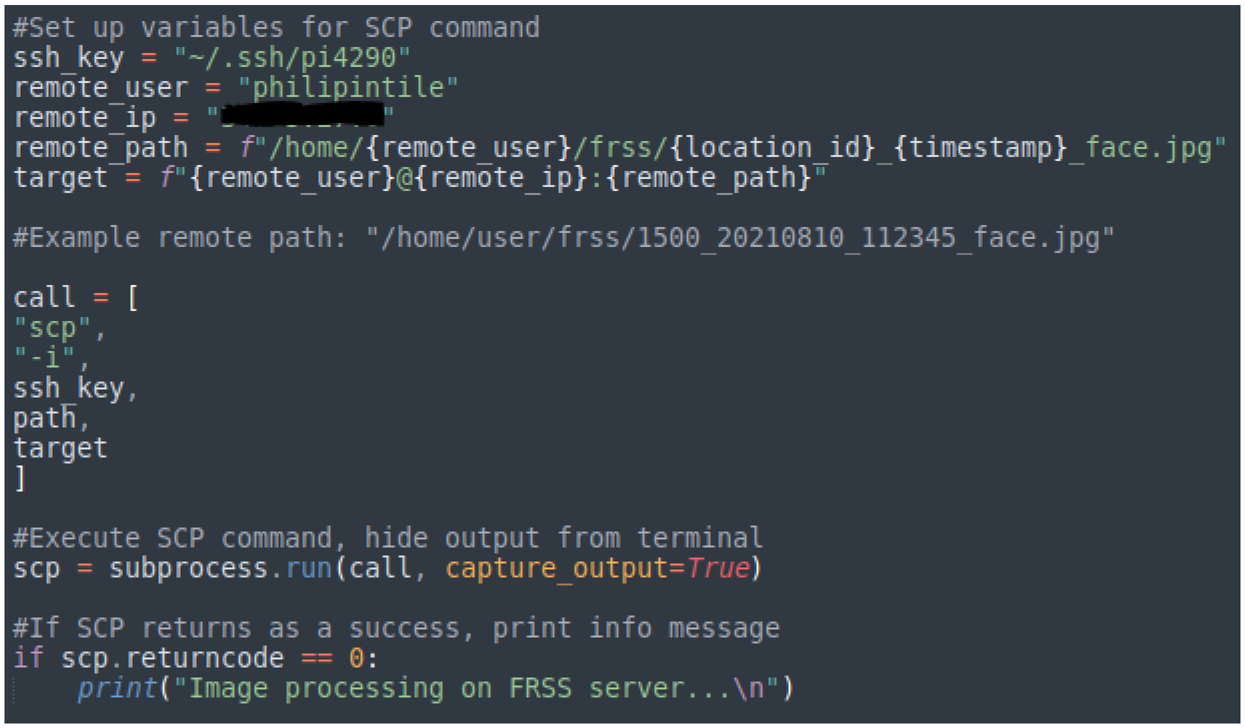
On the client virtual machine, a system administrator can launch the frss_capture script to start the process of face detection. The program will begin a loop to detect a face within the attached camera's frame. If a face is seen in the image, the image is converted from color to grayscale, cropped to include only the face, and written to a specified directory.



```
def capture_image(path):
    """
    Capture an image using OpenCV. Detects a face, and saves an image after 8 loops to
    ensure that the camera is adequately exposed.

    @type path: string
    """

    ready = False
    count = 0

    #Initialize camera
    cam = cv2.VideoCapture(0)
    cam.set(3, 640) # set video width
    cam.set(4, 480) # set video height
    cam.set(cv2.CAP_PROP_FOURCC, cv2.VideoWriter_fourcc(*'MJPG'))
    face_detector = cv2.CascadeClassifier('/usr/local/share/OpenCV/haarcascades/haarcascade_

    print("Capturing Image - Please stand still...")

    while not ready:
        ret, img = cam.read()
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        faces = face_detector.detectMultiScale(gray, 1.3, 5)
        count += 1
        if count == 8:
            for (x,y,w,h) in faces:
                cv2.rectangle(img, (x, y), (x+w, y+h), (255, 0, 0), 2)
                cv2.imwrite(path, gray[y:y+h,x:x+w])
                ready = True
        #Reset loop in case image not found
        if count == 9:
            count = 0

    cam.release()
```

Fig. 10

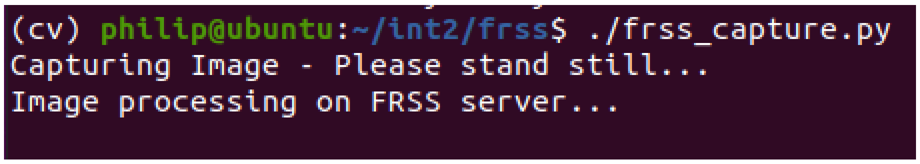
Once the image has been saved, it is sent to the server with some additional data included in the file name including the location ID of the client, and the current timestamp. This process takes place using the secure copy (SCP) program within Ubuntu.



```
#Set up variables for SCP command
ssh_key = "~/.ssh/pi4290"
remote_user = "philipintile"
remote_ip = "[redacted]"
remote_path = f"/home/{remote_user}/frss/{location_id}_{timestamp}_face.jpg"
target = f"{remote_user}@{remote_ip}:{remote_path}"

#Example remote path: "/home/user/frss/1500_20210810_112345_face.jpg"

call = [
"scp",
"-i",
ssh_key,
path,
target
]

#Execute SCP command, hide output from terminal
scp = subprocess.run(call, capture_output=True)

#If SCP returns as a success, print info message
if scp.returncode == 0:
    print("Image processing on FRSS server...\n")
```

Fig. 11

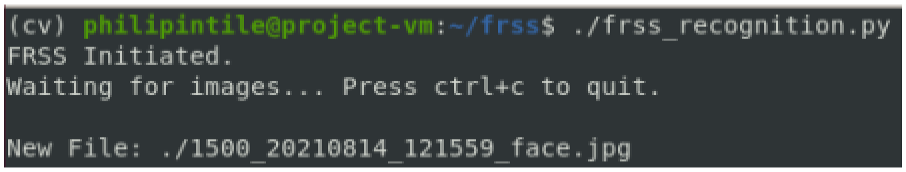
The script then sleeps for 5 seconds to allow the server to process the image.



```
(cv) philip@ubuntu:~/int2/frss$ ./frss_capture.py
Capturing Image - Please stand still...
Image processing on FRSS server...
```

Fig. 12

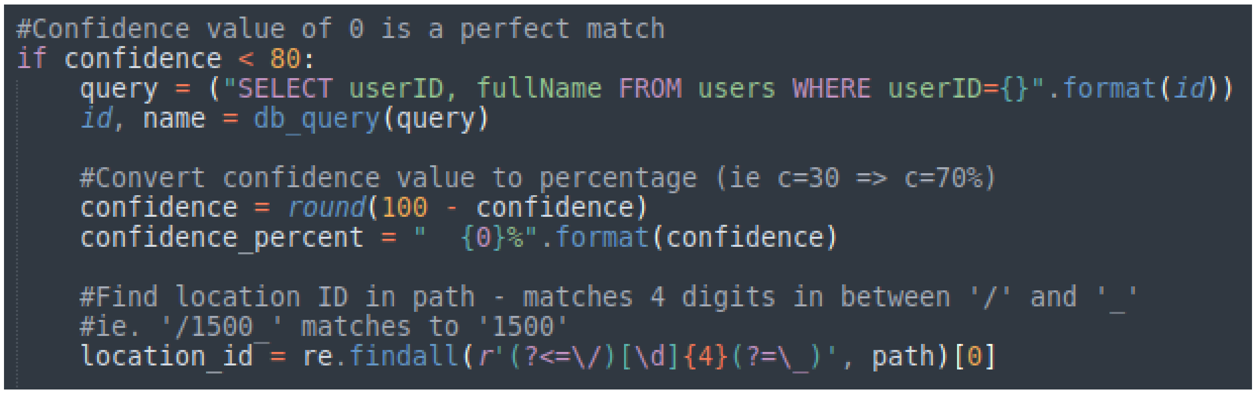
During the 5 second wait, the cloud instance will receive the image sent via secure copy. Using watchdog, an observer is created for a specific directory, and an event handler is created to perform a particular set of actions when an event occurs. In this case, when a new image arrives in the observed directory, the frss_recognition script will call its 'recognize' function with the newly arrived image.



```
(cv) philipintile@project-vm:~/frss$ ./frss_recognition.py
FRSS Initiated.
Waiting for images... Press ctrl+c to quit.

New File: ./1500_20210814_121559_face.jpg
```

Fig. 13

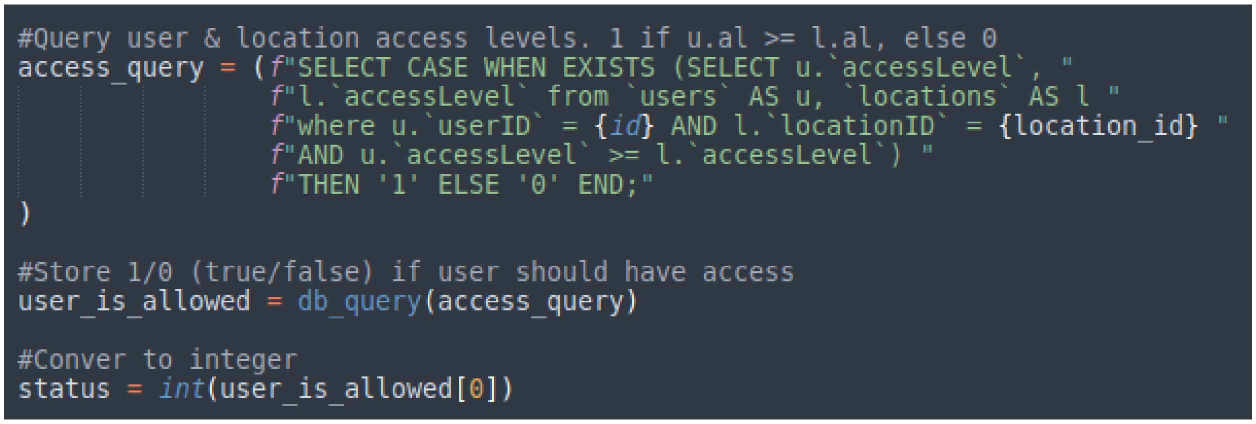
Inside the recognize function, the image is converted to grayscale and compared to the training data found in the previously created trainer.yml file. This additional conversion to grayscale is a redundancy measure in the event that a non-grayscale image is ever added to the directory. If the system recognizes the face, and is confident enough in that recognition, the next steps can occur. First, the recognized user ID is used to verify the user ID in the database, and extract the full name of the user. Next, a regular expression is used to extract the location ID from the filename of the received image.



```
#Confidence value of 0 is a perfect match
if confidence < 80:
    query = ("SELECT userID, fullName FROM users WHERE userID={}".format(id))
    id, name = db_query(query)

    #Convert confidence value to percentage (ie c=30 => c=70%)
    confidence = round(100 - confidence)
    confidence_percent = "  {0}%".format(confidence)

    #Find location ID in path - matches 4 digits in between '/' and '_'
    #ie. '/1500_' matches to '1500'
    location_id = re.findall(r'(?<=\/)[\d]{4}(?=\_)', path)[0]
```

Fig. 14

Next, a database query is used to determine if the user has the appropriate access level to gain entry to the requested location. As long as the user's access level is equal to or greater than the access level required for the location, the user will be approved.



```
#Query user & location access levels. 1 if u.al >= l.al, else 0
access_query = (f"SELECT CASE WHEN EXISTS (SELECT u.`accessLevel`, "
                f"l.`accessLevel` from `users` AS u, `locations` AS l "
                f"where u.`userID` = {id} AND l.`locationID` = {location_id} "
                f"AND u.`accessLevel` >= l.`accessLevel`) "
                f"THEN '1' ELSE '0' END;"
)

#Store 1/0 (true/false) if user should have access
user_is_allowed = db_query(access_query)

#Conver to integer
status = int(user_is_allowed[0])
```

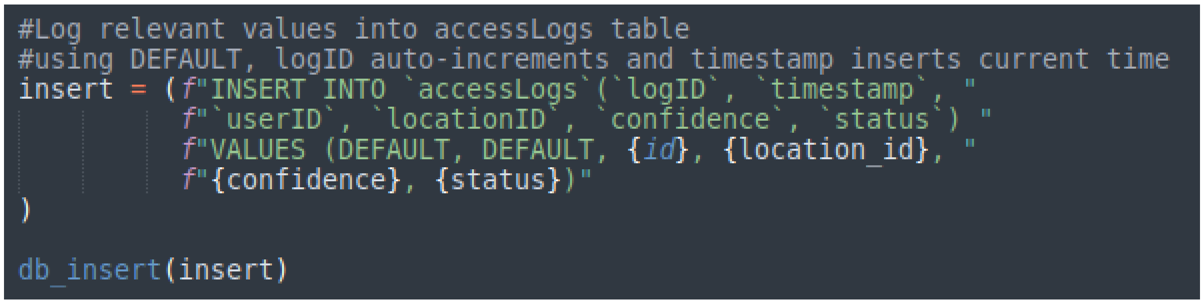
Fig. 15

Example status values given access levels are as follows:

User Access Level	Location Access Level	Status
3	3	1
3	1	1
1	1	1
1	2	0
2	3	0

Table 1

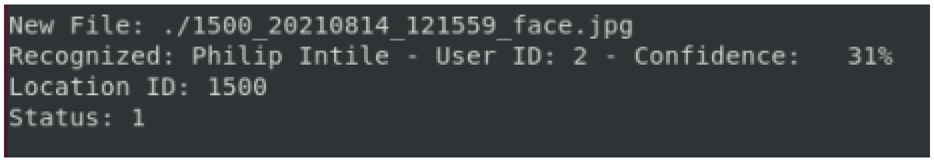
After the access level verification, an access log entry is created which contains the logID, timestamp (UTC), userID, locationID, confidence, and status (1 or 0 depending on if access was granted or not) to ensure that the system is auditable, and that the remote machines can approve access.



```
#Log relevant values into accessLogs table
#using DEFAULT, logID auto-increments and timestamp inserts current time
insert = (f"INSERT INTO `accessLogs`(`logID`, `timestamp`, "
          f"`userID`, `locationID`, `confidence`, `status`) "
          f"VALUES (DEFAULT, DEFAULT, {id}, {location_id}, "
          f"{confidence}, {status})"
)

db_insert(insert)
```

Fig. 16

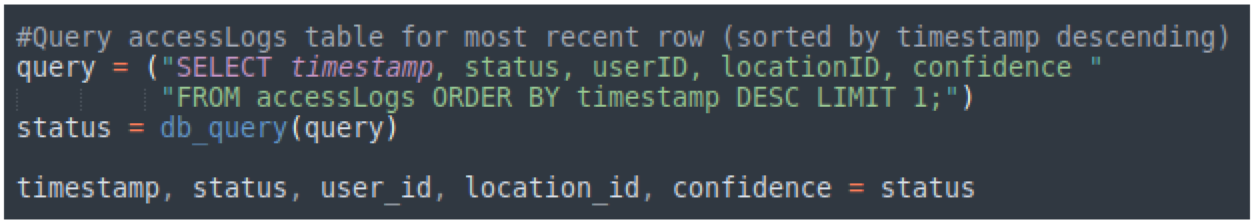
The system then displays a message that the user was recognized, and the image is deleted. Upon completion, the script will return to its listening state using the watchdog observer.



```
New File: ./1500_20210814_121559_face.jpg
Recognized: Philip Intile - User ID: 2 - Confidence:   31%
Location ID: 1500
Status: 1
```

Fig. 17

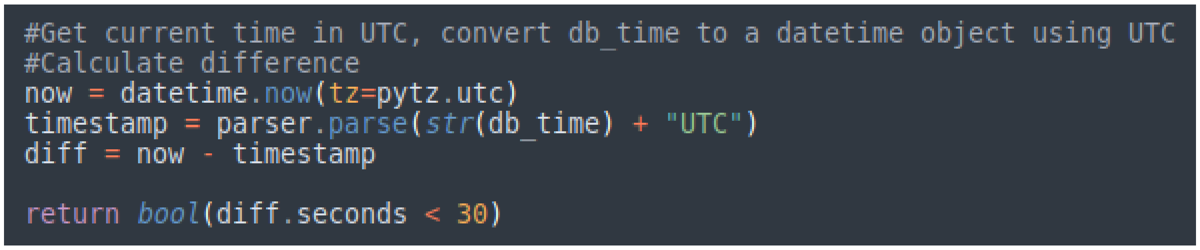
Returning to the virtual machine, the five second wait should now have occurred. The script can now continue on to send a database query on the accessLogs table. This query happens using a read-only account that only has access to SELECT queries to ensure database security on the remote machines. The query sent will return the most recent row from the accessLogs table, sorted descending by timestamp.



```python
#Query accessLogs table for most recent row (sorted by timestamp descending)
query = ("SELECT timestamp, status, userID, locationID, confidence "
         "FROM accessLogs ORDER BY timestamp DESC LIMIT 1;")
status = db_query(query)

timestamp, status, user_id, location_id, confidence = status
```

Fig. 18

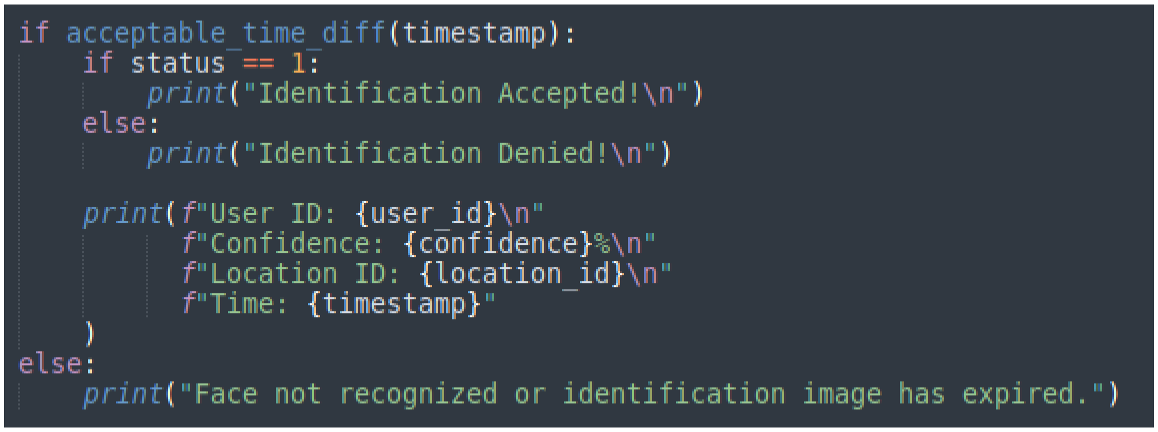
Using the timestamp from that row, a function is called to compare the current time, and the timestamp in the received row. If the difference between those times is greater than 30 seconds, the access request will be discarded. This is to prevent any potential database issues where old data could potentially be retrieved, and instead only allows current requests to be approved.



```python
#Get current time in UTC, convert db_time to a datetime object using UTC
#Calculate difference
now = datetime.now(tz=pytz.utc)
timestamp = parser.parse(str(db_time) + "UTC")
diff = now - timestamp

return bool(diff.seconds < 30)
```

Fig. 19

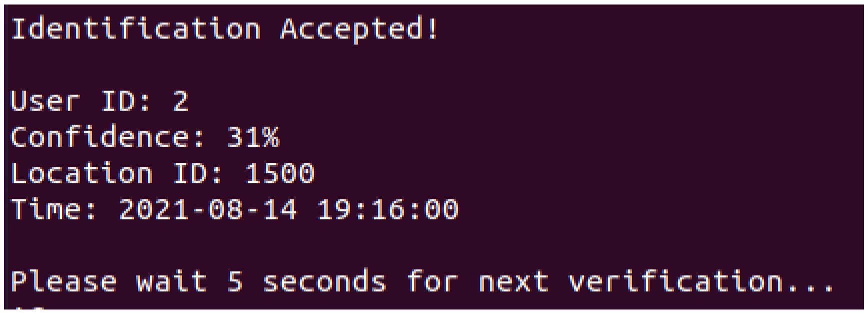
If the difference is below 30 seconds, the request continues on to check the status field. A result of 1 (true) means that access is granted! Otherwise, the access request will be denied.



```python
if acceptable_time_diff(timestamp):
    if status == 1:
        print("Identification Accepted!\n")
    else:
        print("Identification Denied!\n")

    print(f"User ID: {user_id}\n"
          f"Confidence: {confidence}%\n"
          f"Location ID: {location_id}\n"
          f"Time: {timestamp}"
    )
else:
    print("Face not recognized or identification image has expired.")
```

Fig. 20

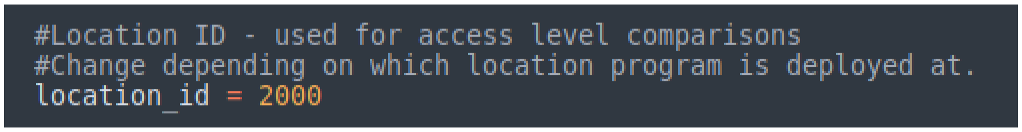
After either option, the system will print the user ID, confidence percentage, location ID, and timestamp to the terminal. This information can be used to easily ensure the system is working on the user side.



```
Identification Accepted!

User ID: 2
Confidence: 31%
Location ID: 1500
Time: 2021-08-14 19:16:00

Please wait 5 seconds for next verification...
```

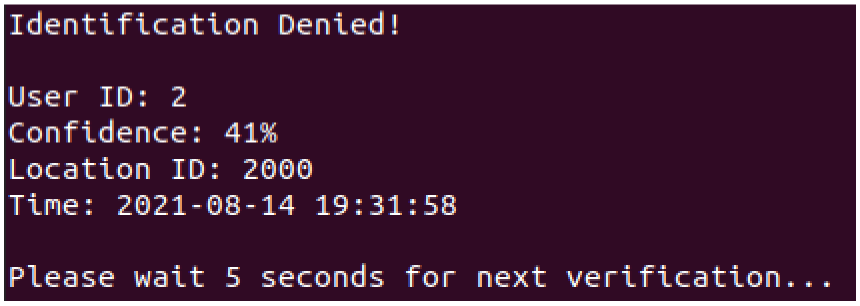
Fig. 21

Looking at Figure 7 in the Database Design section we can see that Philip is a developer, therefore he should have access to the room as seen in Figure 21. If the user's access level is insufficient for the given location ID set in the frss_capture program, access will be denied.



```
#Location ID - used for access level comparisons
#Change depending on which location program is deployed at.
location_id = 2000
```

Fig. 22



```
Identification Denied!

User ID: 2
Confidence: 41%
Location ID: 2000
Time: 2021-08-14 19:31:58

Please wait 5 seconds for next verification...
```

Fig. 23

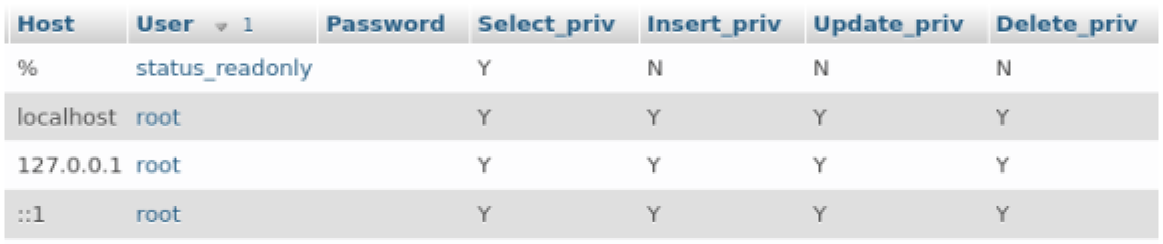
In this case, The locationID is 2000, the Server Room. In Figure 5, it can be seen that the accessLevel for this room is 3. Philip only has an accessLevel of 2, so he does not have access to the Server Room.

Finally, a short five second wait takes place to give the user time to move away from the camera, and for the next user to make their way in front of it. From there, the system loops back to the beginning of the facial detection process.

System Security

A number of important system security features were implemented throughout the development of the FRSS. One of the main features is that the SQL database is password protected for all access. Even more important is that remote logon using the 'root' user is not possible.

The following table displays users, and a selection of available permissions:

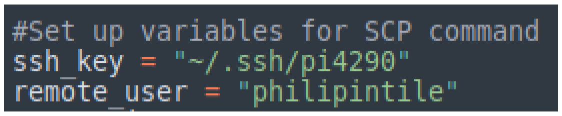


Host	User ▾ 1	Password	Select_priv	Insert_priv	Update_priv	Delete_priv
%	status_readonly		Y	N	N	N
localhost	root		Y	Y	Y	Y
127.0.0.1	root		Y	Y	Y	Y
::1	root		Y	Y	Y	Y

Fig. 24

As it can be seen, the root user is only available on local interfaces such as localhost, 127.0.0.1, and ::1 (for IPv6). The other main account for this database is the 'status_readonly' account. This account is used for the image stations, and is only ever required for reading the accessLogs table. As such, only the SELECT privilege was required. This account does not have the capability to insert, update, or delete any information within the FRSS database.

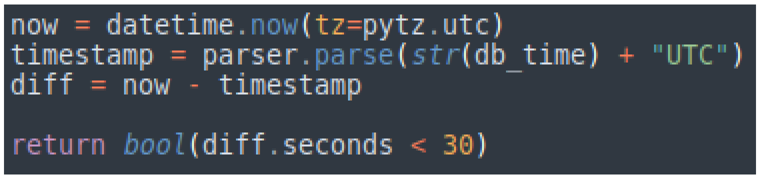
Within the frss_capture script, a function is in place to use the SCP (secure copy) command to copy an image from the virtual machine to the remote server. To enhance the security within the script, SSH keys were used. The SSH keys allowed for the script to carry out this command without a need to put a password to the account on the cloud instance in plain text within the script.



```
#Set up variables for SCP command
ssh_key = "~/.ssh/pi4290"
remote_user = "philipintile"
```

Fig. 25

Later on in the same script, the acceptable_time_diff function was used to guarantee only the most recent requests were being approved. This function calculated the time difference between the timestamp from the database, and the time of execution within the script. Any difference above 30 seconds leads to the request being discarded. This means that in the event that the wrong row is somehow received from the database, it is much less likely that it would be falsely accepted.

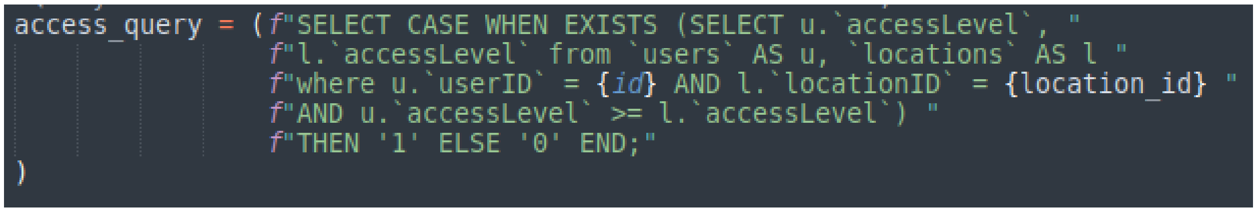


```
now = datetime.now(tz=pytz.utc)
timestamp = parser.parse(str(db_time) + "UTC")
diff = now - timestamp

return bool(diff.seconds < 30)
```

Fig. 26

Returning to the server, some useful security features were also implemented within the frss_recognition script. One of note is relying on the confidence value reported from the OpenCV recognizer. Below a certain threshold of confidence, a face image will be rejected. Given enough training data, the recognizer can confidently identify a user from an image of their face, so if only the most confident of recognitions allows for access to the rest of the system, the system is more secure.

If the system is confident that it recognizes a user, additional security checks are in place. In the FRSS, each user has a specified access level. The design of the access level is that the higher the value, the more access they have. For example, an access level of 3 would have more access than 1, while also being able to access anything that the levels below them are able to access. The following access query is used to determine if the recognized user should have access to the location that the supplied image is coming from.

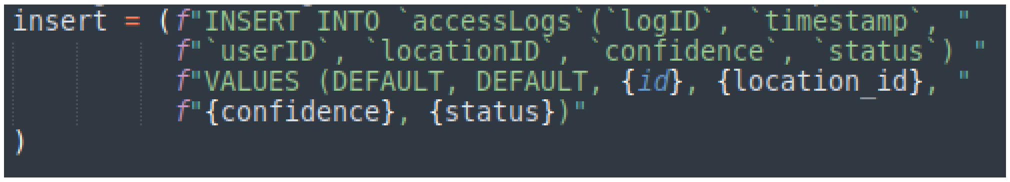


```
access_query = (f"SELECT CASE WHEN EXISTS (SELECT u.`accessLevel`, "
                f"l.`accessLevel` from `users` AS u, `locations` AS l "
                f"where u.`userID` = {id} AND l.`locationID` = {location_id} "
                f"AND u.`accessLevel` >= l.`accessLevel`) "
                f"THEN '1' ELSE '0' END;"
)
```

Fig. 27

As it can be seen, the access levels of the user and location are requested. If the user's access level is equal to or greater than the access level of the location, they are granted access (1). If not, their access is denied (0).

Once the access query is accepted or denied, an additional security feature is executed - logging. Given the information gathered from previous code in the function, a row is inserted into the accessLogs table.



```
insert = (f"INSERT INTO `accessLogs`(`logID`, `timestamp`, "
          f"`userID`, `locationID`, `confidence`, `status`) "
          f"VALUES (DEFAULT, DEFAULT, {id}, {location_id}, "
          f"{confidence}, {status})"
)
```

Fig. 28

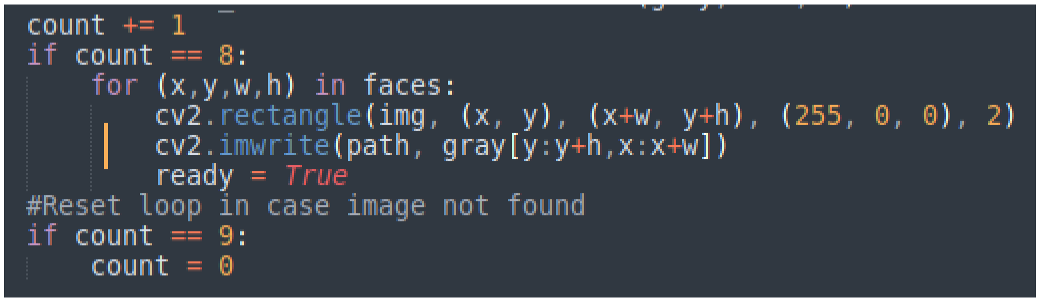
This insert statement uses an automatically incrementing ID for the logID, and an automatically generated current timestamp for the timestamp column. The requesting user's ID number is inserted, alongside the requesting location, and confidence value. Finally, the status (1 or 0) is inserted. These values allow for efficient, informative logging that allow an administrator to easily view when, where, and by whom the building was accessed.

As a final step in the recognizer function, the received image is removed. This means that the user's face is only stored on the FRSS server for a matter of seconds at most, allowing for privacy concerns to be easily avoided.

Testing & Quality Assurance Measures

As facial recognition is still an emerging technology, and few applications are completely error-proof, significant testing and quality assurance measures took place over the duration of development.

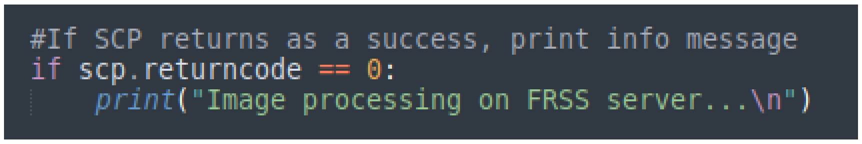
While initially testing the camera used to capture images for the frss_capture script, it was revealed that once the camera is enabled, it takes a short period of time to expose itself properly. This means that the first few frames captured by the camera were significantly darker than the frames after the camera had properly exposed itself to the surrounding lighting conditions. As such, the accuracy of the FRSS was reduced using these images as input. To mitigate this issue, a small loop was written that waits 8 frames after the camera is initialized before capturing and writing an image. If no face is detected, the process resets and waits an additional 8 frames.



```
count += 1
if count == 8:
    for (x,y,w,h) in faces:
        cv2.rectangle(img, (x, y), (x+w, y+h), (255, 0, 0), 2)
        cv2.imwrite(path, gray[y:y+h,x:x+w])
        ready = True
#Reset loop in case image not found
if count == 9:
    count = 0
```

Fig. 29

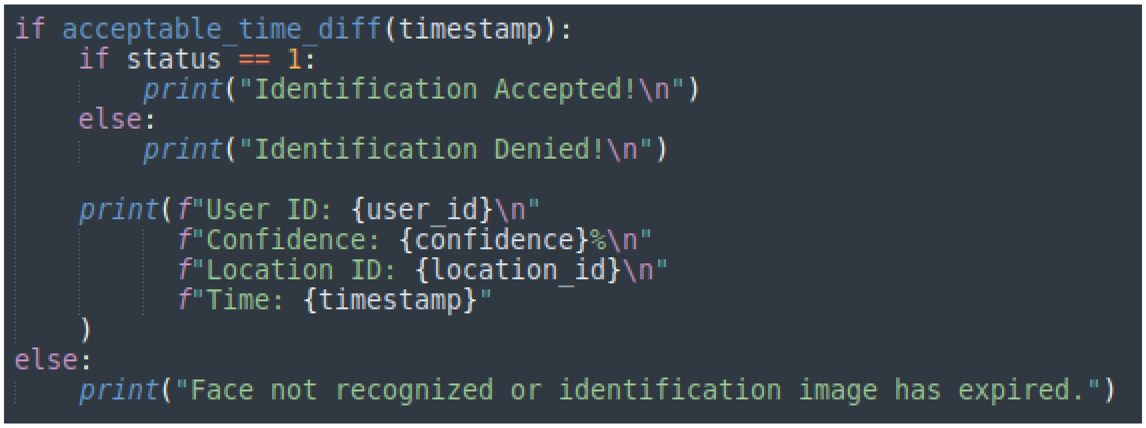
When sending an image from the virtual machine to the server, the scp (secure copy) command is used. As this is a Linux command being executed via the subprocess module in Python, a return code is received that stores whether or not execution was successful. The FRSS code takes advantage of this in order to determine if a notification message should be printed.



```
#If SCP returns as a success, print info message
if scp.returncode == 0:
    print("Image processing on FRSS server...\n")
```

Fig. 30

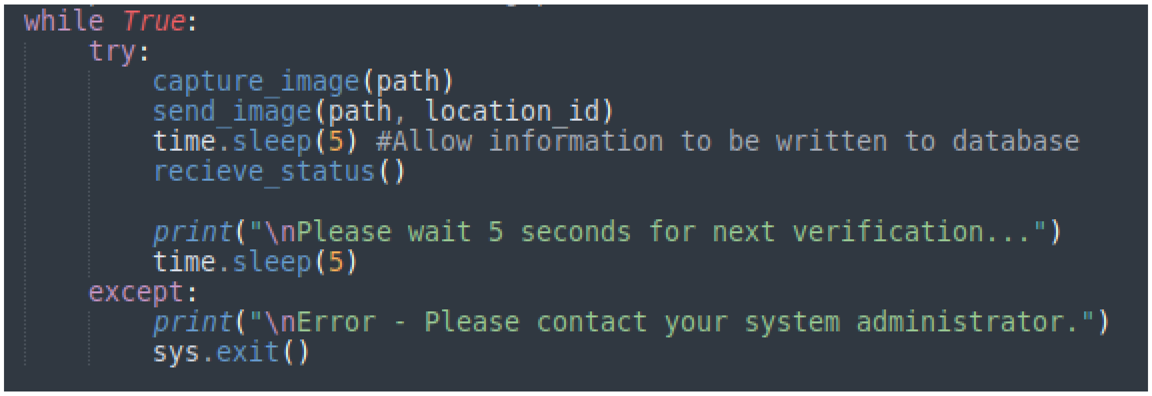
When the FRSS reviews the access log update, error checking is in place in the event that a face is not recognized, or the time difference between execution and logging is too high.



```
if acceptable_time_diff(timestamp):
    if status == 1:
        print("Identification Accepted!\n")
    else:
        print("Identification Denied!\n")

    print(f"User ID: {user_id}\n"
          f"Confidence: {confidence}%\n"
          f"Location ID: {location_id}\n"
          f"Time: {timestamp}"
    )
else:
    print("Face not recognized or identification image has expired.")
```

Fig. 31

Within the main frss_capture loop, exception handling was implemented in order to stop the program and print a notification to the terminal. This handling will take over in the event that any part of the application throws an error. This will avoid long blocks of error messages from printing out on a user-accessible interface.



```
while True:
    try:
        capture_image(path)
        send_image(path, location_id)
        time.sleep(5) #Allow information to be written to database
        recieve_status()

        print("\nPlease wait 5 seconds for next verification...")
        time.sleep(5)
    except:
        print("\nError - Please contact your system administrator.")
        sys.exit()
```

Fig. 32

In order to determine which location the image arrives from on the server, a location ID is inserted into the file name of the image. An example remote path would look like the following:

```
/home/user/frss/1500_20210810_112345_face.jpg
```

To ensure this ID could be extracted no matter which path the file appears in, a small regular expression was written in the FRSS code:

```
location_id = re.findall(r'(?<=\/)[\d]{4}(?=\_)', path)[0]
```

This regular expression can be split into three distinct sections:

1. Positive lookbehind that matches with a forward slash: "`(?<=\/)`"
2. Pattern matching to find four consecutive digits (0-9): "`[\d]{4}`"
3. Positive lookahead that matches with an underscore: "`(?=\_)`"

Once combined, it will match any piece of text that follows the format: "`/####_`". As the location ID is set at the beginning of the file name, it will always match this regular expression. For additional ease, due to the positive lookbehind and lookahead, the forward slash and underscore are removed from the pattern matching, and only the required four-digit location ID is returned.

Recommendations

While overall the FRSS works almost exactly as originally intended, there are a few potential areas for future improvement. In the short to medium term, a few new features could greatly improve the user experience of the FRSS.

One short-term feature would be improving portability. In its current state, the FRSS requires a number of prerequisite libraries. A major improvement could be to package the separate parts of the system into containers using an application such as Docker. Containers would allow for full portability of the FRSS on any system that supports them, and would allow for incredibly quick deployment of new capture locations. For example, if a new camera station were to be installed at a new door in a building, a Docker container could portably move the code and required modules to the new system without the need to manually install everything and risk potential user-error.

In regards to the server implementation of the FRSS, some additional security measures could be easily implemented on a short-term timeline. A simple, but substantial target would be to create service accounts for the database with heavily restricted permissions similar to the status_readonly account currently implemented in the FRSS. A service account could also be used for executing the code on both the client and server side, to further reduce the scope of required permissions on an account used to execute code. Even further, using Docker or virtualization, the scope of implementation could be greatly reduced down to a standalone virtual machine to reduce chances of security breaches.

Another short to medium term recommendation would be to improve the user experience with a user interface at the capture stations. This could be done with a portable computer such as a Raspberry Pi. Hardware devices such as a small touchscreen or button to quickly initiate the photo could be installed, and a display could be used to visualize the detected face and approval information. This would lead to slightly less mystery on the user side while interacting with the FRSS.

Longer-term, an administration user interface could be created. The interim solution for an administration user interface was implementing phpMyAdmin. This system allows for visual access to the SQL database, but the interface can be cumbersome and complex at times. A dedicated website that interfaces with the SQL database could be used by administrators to easily add, remove, or update users and locations without having to enter SQL statements manually. The UI should be clean and easy to use with a modern aesthetic design.

Further long-term recommendations could be to put significantly more research into the technical aspects of OpenCV facial detection and recognition. This would allow for more optimization to be put into the photo gathering and training to ensure the security and ease of use of the system, while reducing potential system errors.

Another long-term goal would be to enhance our understanding of privacy, data retention, and data governance laws in Canada. Due to the gathering of employee photos and facial recognition training data, it is very likely that rules, regulations, laws, standards, and frameworks are already designed and developed by the government that apply to these types of systems. The goal is to carry out the implementation of a system like the FRSS in a fully legal and ethical manner with respect to the privacy of the users of the system.

Project Management and Quality Assurance

Project Effectiveness & Lessons Learned

As a whole, the process of creating the FRSS was an efficient and valuable experience. Choosing to approach this project in an iterative way allowed for the greatest amount of value to be extracted from building this system, as it gave significant freedom to adapt to changing information as the project progressed.

The development of this project allowed for broad exposure to different technologies that have real-world use and value in the information technology industry, such as the following:

- Cloud computing platforms (Google Cloud)
- Virtual Machines (VMware)
- Linux (Ubuntu)
- OpenCV (computer vision, facial recognition)
- XAMPP (phpMyAdmin, MariaDB, MySQL)
- Python
- Watchdog

Having exposure to each of the above technologies and integrating them as one cohesive system was a very good way to gain a deeper understanding of each individual system. While each technology has power on its own, finding ways to piece them together allowed for a fairly seamless process of creating a security system using facial recognition.

Project Management Practices

Throughout the duration of the project, weekly meetings were held to discuss progress, concerns, and future plans. Through following an iterative development approach, these meetings allowed for progression through each phase, while staying on top of tasks and evaluating the necessities of specific features and designs.

Meetings were typically held over the course of one to two hours using Discord, and progress would be shared via screen capture. This way, specific feature implementations could be easily demonstrated remotely to each team member, and suggestions could be tested or implemented in real-time.

Discord also proved to be useful for asynchronous communication, and allowed for our group to frequently communicate and share ideas through the duration of the project via text without needing to meet as a group. Typically, text-based discussions would go over new ideas or errors we ran into between meetings, and allowed for a more rapid process of feedback during development.

A required task for this project was to write and submit several status reports. These reports included what was done since the last report, what was currently being worked on, and what was currently planned to work on for the next status report. Beyond group meetings, several meetings were held with our Instructor to ensure that progress was on-track, and was meeting the required deliverables for the project. Meetings were typically 10 to 20 minutes long, and included some review of the submitted status reports, and some demonstration of early code snippets during development.

Risk Review

With the development of the FRSS completed, the initial risks presented at the beginning of the project can be evaluated and reviewed.

An initial risk mentioned was the potential for bias, inaccuracy, and a lack of transparency in the facial recognition software. OpenCV is open source software, and is well researched, so there is a high level of transparency on the facial recognition software itself for the FRSS. While bias was not an issue for the development of the FRSS, the potential for it to exist in the software is absolutely a possibility given a large enough sample size. Inaccuracy was an issue in the FRSS, but it is one that can be solved with an increased dataset size, and further training of the recognizer.

Additional risks mentioned included false positives and false negatives, which were occasionally seen through development and testing of the FRSS. Once again, these risks can be mitigated through further training and an increased sample size. Due to the constraints of group size, the FRSS was only trained on four faces, so the chance of these errors can be quite high simply due to lack of information.

Transparency on how facial recognition data is used was another potential risk. In the case of the FRSS, the only retained photos are the ones used to train the recognizer. All other photos exist only for a matter of seconds in transit before they are deleted by the FRSS. In a real-world implementation, this could be explicitly laid out in something like a privacy policy.

The final risk originally mentioned was cost. In this implementation of the FRSS, all additional costs were avoided. The virtual machine was hosted on a personal computer, and the server was hosted using free trial credits on the Google Cloud Platform. This allowed for the full development and testing of the FRSS to take place entirely for free. Extending past this free trial however would incur additional costs for the Google Cloud server instance.

Conclusion

The FRSS was researched, developed, tested, and completed over the last two semesters. During this period, significant work was completed to ensure the FRSS would be a valuable resource beyond just a proof of concept. Ultimately, the goal was to create a system that can detect and recognize a face, state the user associated with the face, and act upon that relationship as a security system. This means that the system should allow for administration and logging, and be able to give specific users access to specific areas.

The following list is a selection of project objectives set for the FRSS prior to development:

- The FRSS should take into account both physical and digital security
- The FRSS should be capable of recognizing a face in under 5 seconds
- The FRSS should be integrated with a password-protected database
- The secure database should store access requirements and logs
- The FRSS should have a secure front-end for administrators for the provision of access to specific areas
- The FRSS should be able to receive images from remote machines and return access information in under 5 seconds
- The FRSS should have complete project documentation to aid learning and future development

At the end of this project, it can confidently be said that each and every one of the above objectives have been met or exceeded by the implementation of the FRSS.

The FRSS includes features to detect and recognize faces of multiple users, is secured by a password protected database, ssh keys, and secure copy, and is capable of carrying out the process of security via facial recognition in well under five seconds. Using Python, a database, and open-source libraries, the objectives were accomplished and a functional facial recognition security system was created.

The hypothesis of this project was that a FRSS may be a more convenient and secure solution than traditional badge-based access control systems. After implementing the FRSS, it can be said with some confidence that with enough data and training, a FRSS could easily rival or improve upon the convenience of a badge-based system. The FRSS is very fast, modular, scalable, and monitorable. These features make it a potentially desirable alternative for companies who might be looking for a modern alternative to traditional area security measures.

In conclusion, this project was a great success and incredibly valuable learning experience in a wide variety of technologies.

References

Bruegge, B., & Dutoit, A. (2010). Object-Oriented Software Engineering Using UML, Patterns, and Java^{TM} (Third Edition). Prentice Hall.

Buzdar, K. (2021, March 26). *How to install XAMPP on your Ubuntu 20.04 LTS SYSTEM*. VITUX. https://vitux.com/ubuntu-xampp/.

Christensson, P. (2006). RUP Definition. Retrieved 2021, Aug 15, from https://techterms.com

Crumpler, W., & Lewis, J. A. (2021, June). How Does Facial Recognition Work? https://csis-website-prod.s3.amazonaws.com/s3fs-public/publication/210610_Crumpler_Lewis_FacialRecognition.pdf.

Google. (n.d.). *Managing ssh keys in metadata | compute engine documentation*. Google. https://cloud.google.com/compute/docs/instances/adding-removing-ssh-keys#createsshkeys.

Google. (n.d.). Transferring files to Linux vms | Compute Engine documentation. Google. https://cloud.google.com/compute/docs/instances/transfer-files#scp.

Mangalapilly, Y. (n.d.). *watchdog*. PyPI. https://pypi.org/project/watchdog/.

Mastromatteo, D. (2019, January 13). *How to create a watchdog in python to look for filesystem changes*. The Python Corner. http://thepythoncorner.com/dev/how-to-create-a-watchdog-in-python-to-look-for-filesystem-changes/.

MySQL. (n.d.). *MySQL 8.0 reference Manual :: 13.2.11.6 subqueries with exists or not exists*. MySQL. https://dev.mysql.com/doc/refman/8.0/en/exists-and-not-exists-subqueries.html.

MySQl. (n.d.). *MySQL Connector/Python developer GUIDE :: 5.1 connecting to mysql Using Connector/Python*. MySQL. https://dev.mysql.com/doc/connector-python/en/connector-python-example-connecting.html.

MySQL. (n.d.). *MySQL Connector/Python developer GUIDE :: 5.3 inserting data Using Connector/Python*. MySQL. https://dev.mysql.com/doc/connector-python/en/connector-python-example-cursor-transaction.html.

Oracle. (n.d.). *MySQL Connector/Python developer GUIDE :: 5.4 querying data Using Connector/Python*. MySQL. https://dev.mysql.com/doc/connector-python/en/connector-python-example-cursor-select.html.

Pandey, S. (2021). How to Install openCV on Ubuntu 18.04 or above - Studytonight. Retrieved 16 August 2021, from https://www.studytonight.com/post/how-to-install-opencv-on-ubuntu-18-or-above

Rovai, M. (2019, September 20). *Real-time face recognition: An end-to-end project*. Medium. https://towardsdatascience.com/real-time-face-recognition-an-end-to-end-project-b738bb0f7348.

Appendix

The following three Python scripts make up the FRSS.

face_training.py

Sourced from - https://towardsdatascience.com/real-time-face-recognition-an-end-to-end-project-b738bb0f7348

```
import cv2
import numpy as np
from PIL import Image
import os
# Path for face image database
path = 'dataset'
recognizer = cv2.face.LBPHFaceRecognizer_create()
detector =
cv2.CascadeClassifier("/usr/local/share/OpenCV/haarcascades/haarcascade_frontalface_default.xml");
# function to get the images and label data
def getImagesAndLabels(path):
    imagePaths = [os.path.join(path,f) for f in os.listdir(path)]
    faceSamples=[]
    ids = []
    for imagePath in imagePaths:
        PIL_img = Image.open(imagePath).convert('L') # grayscale
        img_numpy = np.array(PIL_img,'uint8')
        id = int(os.path.split(imagePath)[-1].split(".")[1])
        faces = detector.detectMultiScale(img_numpy)
        for (x,y,w,h) in faces:
            faceSamples.append(img_numpy[y:y+h,x:x+w])
            ids.append(id)
    return faceSamples,ids
print ("\n [INFO] Training faces. It will take a few seconds. Wait ...")
faces,ids = getImagesAndLabels(path)
recognizer.train(faces, np.array(ids))
# Save the model into trainer/trainer.yml
recognizer.write('trainer/trainer.yml')
# Print the number of faces trained and end program

print("\n [INFO] {0} faces trained. Exiting Program".format(len(np.unique(ids))))
```

frss_capture.py

```
#!/usr/bin/env python3

"""
frss_capture.py

Facial Recognition Security System
INFO 4290 - KPU

Philip Intile - 100335192
Sunveer (Sunny) Sandhu - 100308494
Davis Fulton - 100228199
Stephen Dereniowski - 100319110

This program captures an image, detects a face, and then sends it to a remote server.
Once processed, this program checks a MySQL database for access log information,
and shares whether the face was recognized or not alongside some other useful
information.
"""

import subprocess
import time
import sys
from datetime import datetime
import pytz
import cv2
from dateutil import parser
import mysql.connector

def capture_image(path):
    """
    Capture an image using OpenCV. Detects a face, and saves an image after 8 loops
to
    ensure that the camera is adequately exposed.

    @type path: string
    """

    ready = False
    count = 0

    #Initialize camera
    cam = cv2.VideoCapture(0)
    cam.set(3, 640) # set video width
    cam.set(4, 480) # set video height
    cam.set(cv2.CAP_PROP_FOURCC, cv2.VideoWriter_fourcc(*'MJPG'))
    face_detector =
cv2.CascadeClassifier('/usr/local/share/OpenCV/haarcascades/haarcascade_frontalface_d
efault.xml')
```

```
    print("Capturing Image - Please stand still...")

    while not ready:
        ret, img = cam.read()
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        faces = face_detector.detectMultiScale(gray, 1.3, 5)
        count += 1
        if count == 8:
            for (x,y,w,h) in faces:
                cv2.rectangle(img, (x, y), (x+w, y+h), (255, 0, 0), 2)
                cv2.imwrite(path, gray[y:y+h,x:x+w])
                ready = True
        #Reset loop in case image not found
        if count == 9:
            count = 0

    cam.release()

def send_image(path, location_id):
    """
    Sends an image to a remote server using SCP.

    @type path: string
    @type location_id: string
    """

    #Create timestamp in filename using the Pacific timezone (UTC-8)
    #Formats timestamp as 'YYYYMMDD_HHMMSS' ie '20210810_112345'
    van_tz = pytz.timezone("America/Vancouver")
    timestamp = datetime.now(tz=van_tz).strftime("%Y%m%d_%H%M%S")

    #Set up variables for SCP command
    ssh_key = "~/.ssh/pi4290"
    remote_user = "philipintile"
    remote_ip = "**.**.**.**"
    remote_path = f"/home/{remote_user}/frss/{location_id}_{timestamp}_face.jpg"
    target = f"{remote_user}@{remote_ip}:{remote_path}"

    #Example remote path: "/home/user/frss/1500_20210810_112345_face.jpg"

    call = [
    "scp",
    "-i",
    ssh_key,
    path,
    target
    ]

    #Execute SCP command, hide output from terminal
    scp = subprocess.run(call, capture_output=True)
    #If SCP returns as a success, print info message
    if scp.returncode == 0:
        print("Image processing on FRSS server...\n")


def acceptable_time_diff(db_time):
    """
    Calculate time difference between current and database timestamps.

    @type db_time: datetime
    @rtype: bool
    @return: true/false if the difference is below 30 seconds
    """

    #Get current time in UTC, convert db_time to a datetime object using UTC
    #Calculate difference
    now = datetime.now(tz=pytz.utc)
    timestamp = parser.parse(str(db_time) + "UTC")
    diff = now - timestamp

    return bool(diff.seconds < 30)

def recieve_status():
    """
    Query database for identification status.
    """

    #Query accessLogs table for most recent row (sorted by timestamp descending)
    query = ("SELECT timestamp, status, userID, locationID, confidence FROM
accessLogs ORDER BY timestamp DESC LIMIT 1;")
    status = db_query(query)

    timestamp, status, user_id, location_id, confidence = status

    if acceptable_time_diff(timestamp):
        if status == 1:
            print("Identification Accepted!\n")
        else:
            print("Identification Denied!\n")

        print(f"User ID: {user_id}\n"
              f"Confidence: {confidence}%\n"
              f"Location ID: {location_id}\n"
              f"Time: {timestamp}"
        )
    else:
        print("Face not recognized or identification image has expired.")

def db_connection():
    """
    Connect to the database.

    @rtype cnx: mysql connection object
    @return: connection to the database using read-only account (only SELECT
permission)
    """

    cnx = mysql.connector.connect(
        user='status_readonly',
        password='***********',
        host='**.**.**.**',
        database='frss'
    )

    return cnx

def db_query(query):
    """
    Connects to MySQL database, runs, and returns a one-row result from a specific
query

    @type query: string
    @rtype output: tuple
    @return: values received from database query
    """

    cnx = db_connection()
    cursor = cnx.cursor()

    if query:
        cursor.execute(query)

    for result in cursor:
        output = result

    cursor.close()
    cnx.close()

    return output

def main():
    """
    Main Function
    """

    #Location to write captured image to
    path = "/home/philip/faceimg.jpg"

    #Location ID - used for access level comparisons
    #Change depending on which location program is deployed at.
    location_id = 1001
    #Loop forever with small time gap for next user.
    while True:
        try:
            capture_image(path)
            send_image(path, location_id)
            time.sleep(5) #Allow information to be written to database
            recieve_status()

            print("\nPlease wait 5 seconds for next verification...")
            time.sleep(5)
        except:
            print("\nError - Please contact your system administrator.")
            sys.exit()


if __name__ == "__main__":
    main()
```

frss_recognition.py

```
#!/usr/bin/env python3

"""
frss_recognition.py

Facial Recognition Security System
INFO 4290 - KPU

Philip Intile - 100335192
Sunveer (Sunny) Sandhu - 100308494
Davis Fulton - 100228199
Stephen Dereniowski - 100319110

This program receives images, and uses OpenCV to recognize faces within the images.
Interacts with a SQL database for user and access information, and logging.
"""

import time
import os
import re
import mysql.connector
import cv2
from watchdog.observers import Observer
from watchdog.events import PatternMatchingEventHandler

def wd_handler():
    """
    Initialize watchdog handler

    @rtype: watchdog event handler
    @return: watchdog event handler
    """

    patterns = ["*"]
    ignore_patterns = None
    ignore_directories = False
    case_sensitive = True

    return PatternMatchingEventHandler(
        patterns,
        ignore_patterns,
        ignore_directories,
        case_sensitive
    )

def on_created(event):
    """
    Watchdog - code to execute when a file is created in the observed directory
    @type event: event
    """

    path = event.src_path
    print(f"New File: {path}")

    recognize(path)

def wd_observer():
    """
    Watchdog Observer - watches directory specified in path for new files
    """

    #Initialize handler and creation event
    frss_wd_handler = wd_handler()
    frss_wd_handler.on_created = on_created

    path = "."
    go_recursively = True
    observer = Observer()
    observer.schedule(frss_wd_handler, path, recursive = go_recursively)

    observer.start()
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        observer.stop()
        observer.join()

def recognize(path):
    """
    Uses OpenCV to recognize faces and MySQL Connector to interact with the database.

    @type path: str
    """

    #Initialize recognizer, trainer, and face cascade
    recognizer = cv2.face.LBPHFaceRecognizer_create()
    recognizer.read('trainer/trainer.yml')
    cascade_path =
"/usr/local/share/opencv4/haarcascades/haarcascade_frontalface_default.xml"
    face_cascade = cv2.CascadeClassifier(cascade_path)

    min_w = 0.1 * 640
    min_h = 0.1 * 480

    #Read image, convert to grayscale
    img = cv2.imread(path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    #Face detection initial setup
    faces = face_cascade.detectMultiScale(
        gray,
        scaleFactor = 1.2,
        minNeighbors = 3,
        minSize = (int(min_w), int(min_h))
    )

    for (x, y, w, h) in faces:
        cv2.rectangle(img, (x, y), (x+w, y+h), (0, 255, 0), 2)
        id, confidence = recognizer.predict(gray[y:y+h,x:x+w])

        #Confidence value of 0 is a perfect match
        if confidence < 80:
            query = ("SELECT userID, fullName FROM users WHERE userID={}".format(id))
            id, name = db_query(query)

            #Convert confidence value to percentage (ie c=30 => c=70%)
            confidence = round(100 - confidence)
            confidence_percent = "  {0}%".format(confidence)

            #Find location ID in path - matches 4 digits in between '/' and '_'
            #ie. '/1500_' matches to '1500'
            location_id = re.findall(r'(?<=\/)[\d]{4}(?=\_)', path)[0]

            #Query user & location access levels. 1 if u.al >= l.al, else 0
            access_query = (f"SELECT CASE WHEN EXISTS (SELECT u.`accessLevel`, "
                            f"l.`accessLevel` from `users` AS u, `locations` AS l "
                            f"where u.`userID` = {id} AND l.`locationID` =
{location_id} "
                            f"AND u.`accessLevel` >= l.`accessLevel`) "
                            f"THEN '1' ELSE '0' END;"
            )

            #Store 1/0 (true/false) if user should have access
            user_is_allowed = db_query(access_query)

            #Convert to integer
            status = int(user_is_allowed[0])

            #Log relevant values into accessLogs table
            #using DEFAULT, logID auto-increments and timestamp inserts current time
            insert = (f"INSERT INTO `accessLogs`(`logID`, `timestamp`, "
                      f"`userID`, `locationID`, `confidence`, `status`) "
                      f"VALUES (DEFAULT, DEFAULT, {id}, {location_id}, "
                      f"{confidence}, {status})"
            )

            db_insert(insert)
            #Terminal output for feedback while program is running
            print(f"Recognized: {name} - User ID: {id} - Confidence:
{confidence_percent}\n"
                  f"Location ID: {location_id}\n"
                  f"Status: {status}\n")

    #Remove the received image, no longer needed
    os.remove(path)


def db_connection():
    """
    Connect to the database.

    @rtype cnx: mysql connection object
    @return: connection to the database using root credentials
    """

    cnx = mysql.connector.connect(
        user='root',
        password=**********,
        host='localhost',
        database='frss'
    )

    return cnx


def db_query(query):
    """
    Connects to MySQL database, runs, and returns a one-row result from a specific
query

    @type query: string
    @rtype output: tuple
    @return: values received from database query
    """

    cnx = db_connection()
    cursor = cnx.cursor()

    if query:
        cursor.execute(query)

    output = cursor.fetchone()
    cursor.close()
    cnx.close()

    return output

def db_insert(insert):
    """
    Insert into the database, and commit.

    @type insert: string
    """

    cnx = db_connection()
    cursor = cnx.cursor()

    if insert:
        cursor.execute(insert)
        cnx.commit()

    cursor.close()
    cnx.close()

def main():
    """
    Main Function
    """

    print("FRSS Initiated.")
    print("Waiting for images... Press ctrl+c to quit.\n")

    wd_observer()

if __name__ == "__main__":
    main()
```