Home Security

Computer and Systems Department
Alexandria University - Faculty Of Engineering

Final Milestone Report for Human-Computer Interaction Project

Prepared By

Ahmed Mustafa Elmorsy Amer	21010189
Ahmed Youssef Sobhy Elgoerany	21010217
Moustafa Esam El-Sayed Amer	21011364
Ebrahim Alaa Eldin Ebrahim	21010017
Ahmed Ayman Ahmed Abdallah	21010048
Ali Hassan Ali Mohamed	21010837

Contents

1 Introduction

1.1 Overview and purpose of the application

Observa is a cutting-edge application designed to provide users with real-time surveillance and monitoring of the doorway to their homes. The primary purpose of the app is to enhance security and convenience for homeowners by leveraging advanced technology such as computer vision and facial recognition. Here are the key features and aspects of **Observa**:

- **Live Stream Monitoring: Observa** offers a live stream of the doorway, allowing users to monitor activities in real time. This can be particularly useful for keeping track of visitors, deliveries, or any unexpected events.
- **Facial Recognition and Visitor Labeling:** The app incorporates a sophisticated facial recognition system. Users have the ability to add known visitors to a database. When a recognized person approaches the door, **Observa** labels them, providing instant information about their identity.
- **Historical Visitor Log: Observa** maintains a comprehensive log of all past visitors, complete with timestamps and identified individuals. This feature allows users to review who has visited their home over time, providing an added layer of security and awareness.
- **Photo Upload for Visitor Database:** Users can upload photos of their known visitors, creating a personalized contact list. This enhances the accuracy of the facial recognition system and ensures that users receive accurate and detailed information about who is at their doorstep.

1.2 Goals And Objectives

The objective of **Observa** is to maintain home-security by developing a smart doorbell system. The system leverages facial recognition feature that notifies the clients whether they are familiar with this person or he is not one of their usual visitors. Streaming and observing the door-way to ensure clients' in-home safety from burglars or undesired visitors. What stimulates the project is the need for more secure and convenient home security solutions. The expected outcome is the creation of a user-friendly, efficient, and effective smart doorbell system that enhances residential security. Existing systems are often expensive, difficult to install and lack advanced features such as face recognition. Besides, such systems are not applicable or available in the middle east. Our system's utility lies in providing homeowners' with real-time access to their doorstep.

1.3 Target audience

- **Homeowners and Families:** Observa is ideal for homeowners and families who want to enhance the security of their homes and keep track of who comes and goes. It provides peace of mind by offering real-time monitoring and historical data on visitors.

- **Security-Conscious Individuals:** People who prioritize security and want a reliable system to monitor their home entrance will find Observa valuable. This includes individuals living in various types of residential settings, from houses to apartments.
- **Busy Professionals and Parents:** Observa caters to individuals with busy lifestyles, such as professionals and parents, who may not always be physically present at home. The app allows them to stay connected and informed about activities at their doorstep, even when they are away.
- **Tech Enthusiasts:** Those who appreciate the integration of advanced technologies like facial recognition and real-time streaming in their daily lives may be drawn to Observa. The app combines innovation with practical utility.

2 Key Functionalities

In out project, the main goal is to provide users with interactive websites that enables them access all the features they demand, based on our previous market research and interviews. All the features included in the following subsections are ones that either facilitate interaction with our environment or add functionalities to our whole program development.

2.1 Home Page

We display a home page that acts as a representative of our services to new users who would like to check the application and still not yet decided to subscribe and give it a try. It redirects users to signup or login if they are interested.

2.2 Sign-up

Users must have an account with specific minimum data in order to subscribe to our service, this data include basic information like email, name, address for later installments, date of birth and phone number. This data is essential in order to provide users credentials to use our web-application.

2.3 Login

The main feature of our project is for each user to have unique database with significant data-structures and data transfer objects, that are unified amongst all users, these transfer objects provide all users whole access to the program and the back-end implementing it.

2.4 Authentication

Since we allow multiple users login, we generate a token that differentiates a user from another, this is necessary for the access of database and all other features of the application. For instance, a user needs to be identified to the system first in order to view his stream, visitors and unique data.

2.5 Authorization

The authentication token is passed to all the routes and the back-end handles the token to verify the user, thus he is authorized to access our features, those are two key features that are established for users' safety and data security.

2.6 Password Hashing

Every user has a password, this password is not stored in the database to ensure users' security. This one-way hashing method is essential to prevent hackers and malicious users from illegal access to the database, and even if they accessed it they won't be able to either get the password, and for the reset password with the user name they will have to access the user's email and get his password in order to know his randomly generated password, before he changes it, which we encourage users to change directly after it being sent.

2.7 Data Change

All users are allowed to change their data, only ones that are changeable, to facilitate their use of the program, in case a user forgets his password or changes his phone number or demands to change his email.

2.8 Reset Password

Additionally, users are allowed to reset their password in case they have forgotten it, in order to log in to the network, if they lost their password the are not prohibited from our service.

2.9 Interaction With Users

We handle sending emails to users in order to reset their password, or to notify them when a visitor is at their doorstep. This keeps users updated to our servers and have easy access to the Observa support team.

2.10 Video Stream

We provide the users with a real time update, as much as tech holds, to their doorstep view, to see who is visiting them and watch their doorway whenever they want to or feel unsafe or warned.

2.11 Face Detection

The stream is provided with models to detect people's faces as long as they are in the range of the camera. This enhances other features that are essential for the main course of the application.

2.12 Visitors Recognition

If visitors are labelled in the user's data base they are recognized, otherwise they are saved in the database for security matters.

2.13 Managing Contacts

Contacts are the visitors labelled in the user data, each visitor saved has a saved image in order to identify him later if he is to visit us again. Users may also label unknown visitors, add other contacts to the database with labelled attributes i.e name, image and relationship to the user. They can also delete an existing contact or even update his saved data.

2.14 Visitors History

Users can see the history of their recent visitors to know, who visited them and when, in case the user was not ready to check his email or the video stream along the day. This will be very useful to hardworking people who would like to check the list of their visitors throughout the day.

2.15 Remember Me

Once users are logged in to our network, they are remembered for a while, i.e if the user refreshes the page he is still logged in and he is still authorized to his endpoints and requests.

2.16 Log out

Users are allowed to logout even if they are logged into their profiles and their session time has not ended yer, since a user might have multiple accounts if he lives in a mansion, or he does have multiple homes and wants to check for his own home safety.

3 Installation Guide

3.1 Database Installation (PostgreSQL)

1. Install PostgreSQL:

- Download and install PostgreSQL from the official website: `https://www.postgresql.org/download/`

2. Create Database:

- Open a PostgreSQL client (e.g., pgAdmin) and create a new database for Observa.

3.2 Backend Installation (Flask)

1. Clone the Backend Repository:

- Open your terminal and run the following command to clone the backend repository:

```
git clone https://github.com/HCI26/Observa-API.git
```

2. Change Directory:

- Navigate to the cloned repository:

```
cd Observa-API
```

3. Create Python Virtual Environment:

- Create a virtual environment using the following command:

```
python3 -m venv venv
```

4. Activate the Virtual Environment:

- Activate the virtual environment:

```
source venv/bin/activate
```

5. Install Requirements:

- Install the required Python packages using the following command:

```
pip install -r requirements.txt
```

6. Open VSCode:

- Open Visual Studio Code (VSCode) in the current folder:

```
code .
```

7. Run the Server:

- Run the server using the launch.json configuration file in VSCode:
  - Press F5 or click on the "Run" icon in the Activity Bar on the side and then click "Start Debugging."

3.3 Frontend Installation (Vue.js)

1. Clone the Frontend Repository:

- Open your terminal and run the following command to clone the frontend repository:

```
git clone https://github.com/HCI26/Front-end.git
```

2. Change Directory:

- Navigate to the cloned repository:

```
cd Front-end
```

3. Install Node.js and npm:

- Ensure you have Node.js and npm installed. You can download them from `https://nodejs.org/`

4. Install Dependencies:

- Install project dependencies using the following command:

```
npm install
```

5. Run the Frontend:

- Start the Vue.js development server:

```
npm run serve
```

- Open your web browser and go to the provided URL (usually `http://localhost:8080/`) to view the Observa frontend.

Congratulations! You have successfully installed Observa. If you encounter any issues, refer to the documentation or seek assistance from the Observa community.

4 User Manual

4.1 Home Page

The home page provides information about the application and its features.

4.2 Sign Up

1. If you are not registered, click on the "Sign Up" button.
2. Fill in the required user details, including username, password, and any additional information.
3. Wait for the system to authenticate and authorize your registration.

4.3 Login

1. If you are already registered, click on the "Login" button.
2. Enter your username and password.
3. Wait for the server to authenticate your login request.

4.4 Dashboard

1. Upon successful login, you will be directed to the dashboard.
2. The dashboard provides a live stream of your doorway with real-time face recognition, detection, and labeling of visitors.
3. You can log out at any time using the logout option.

4.5 Visitor Pages

Access the "Visitor" section to manage your list of visitors.

4.5.1 Add a New Visitor

1. Specify the visitor's details and relationship.
2. Add a photo for recognition, which will be saved in the visitors' database.

4.5.2 View All Visitors

1. Check the list of all your recognized visitors with their last visit date.
2. Edit or delete visitor details as needed.

4.6 Visitor History

1. Navigate to the "Visitor History" page.
2. Access a detailed history of all days and their corresponding visitors.

4.7 Settings

Explore the "Settings" page.

4.7.1 Edit User Details

1. Modify your personal information as needed.

5 Architecture Overview

1. **High-Level Architecture Diagram:**

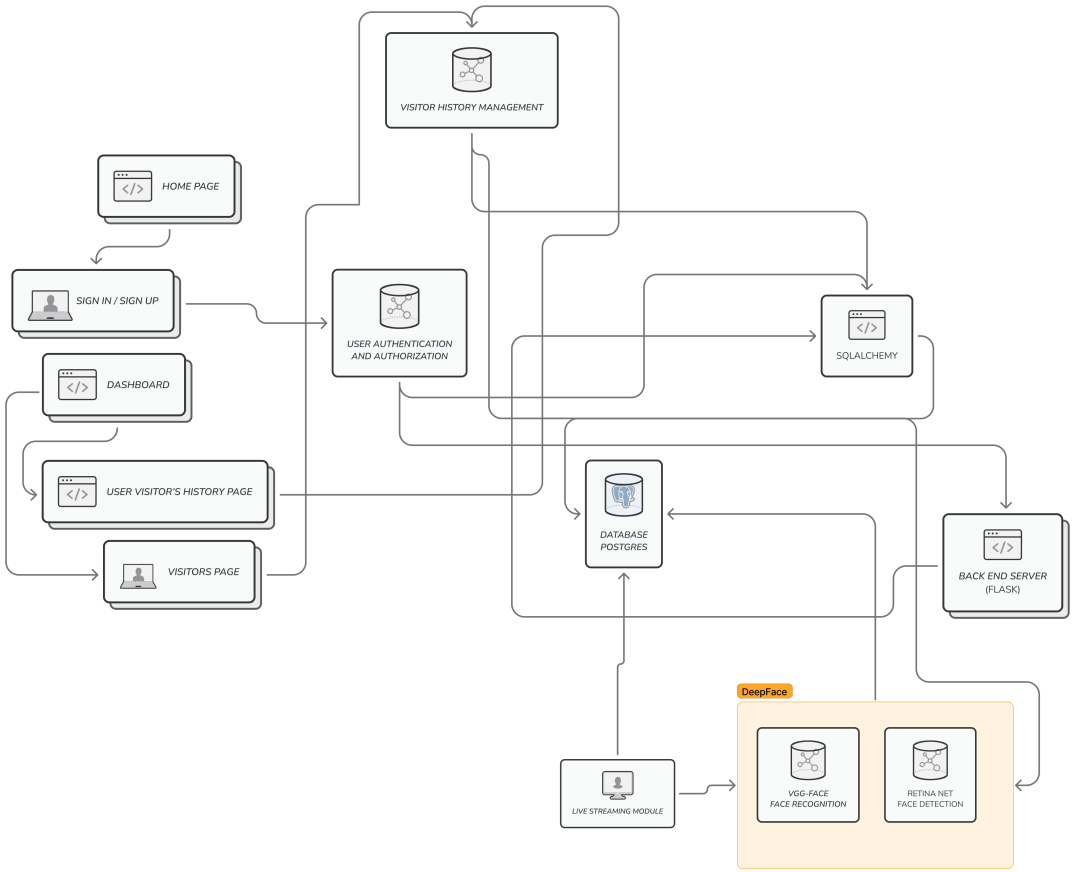

Figure 1: High-Level Architecture Diagram

2. **Layers:**

   - **Presentation Layer:**
     - User Interface (UI): The front-end interface where users interact with the application.
     - Live Stream Display: Displays the live stream from the doorway.
     - Visitor History Interface: Provides access to historical visitor logs.
     - Authentication Module: Handles user login/sign-in and ensures secure access.
   - **Application Layer:**
     - Live Stream Module:
       * Interfaces with the device camera for real-time streaming.
       * Sends the live stream to the facial recognition module.
     - Facial Recognition Module:

        * Utilizes the Deep Face package model for facial recognition.
        * Matches detected faces with the database of known visitors.
        * Labels and identifies recognized visitors.
    - Visitor Database:
        * Stores information about known visitors, including photos and labels.
        * Supports CRUD operations for managing visitor data.
        * SQLAlchemy provides an abstraction layer that enables you to interact with the PostgreSQL database using Pythonic syntax.
    - User Database:
        * Stores user information, including login credentials.
        * Manages user authentication and authorization.
        * SQLAlchemy facilitates CRUD operations for managing user data.
    - SQLAlchemy Integration:
        * The use of SQLAlchemy in this layer allows you to define database models as Python classes and perform operations like querying, inserting, updating, and deleting records using high-level Python functions.
- **Data Access Layer:**
    - Database Management System (DBMS):
        * Manages and stores data securely.
        * Supports efficient retrieval of historical visitor logs and user information.
        * PostgreSQL is the chosen DBMS, responsible for managing and storing data securely.
    - Database migrations using alembic: Easily make alterations in the class structure and apply them to the db with ability to revert changes, too.
- **Security Layer:**
    - Authentication and Authorization:
        * Validates user credentials during login.
        * Ensures that only authorized users can access certain features.
- **Integration Layer:**
    - Integration with Deep Face Package:
        * Connects the facial recognition module with the Deep Face package.
        * Facilitates seamless integration of advanced facial recognition capabilities.
- **Communication Layer:**
    - APIs (Application Programming Interfaces):
        * Facilitates communication between different modules and layers.
        * Enables data exchange between the front-end and back-end components.

3. **Interactions:**

- **User Authentication:**

- The Authentication Module verifies user credentials against the User Database.
- Authenticated users gain access to the Live Stream Display and Visitor History Interface.

- **Live Stream Monitoring:**
  - Live Stream Module captures the real-time video feed.
  - The facial recognition module analyzes the stream for recognized faces.
- **Facial Recognition and Face Detection:**
  - Facial Recognition Module utilizes the Deep Face package to recognize faces.
  - Matches detected faces with known visitors in the Visitor Database.
  - Labels and identifies recognized visitors.
  - The face detection module (RetinaFace) identifies faces in the live stream. RetinaFace is mainly based on an academic study: RetinaFace: Single-stage Dense Face Localisation in the Wild.
  - Detected faces are passed to the face recognition module (VGG-Face) for identification against the visitor database.
    The structure of the VGG-Face model is demonstrated below.

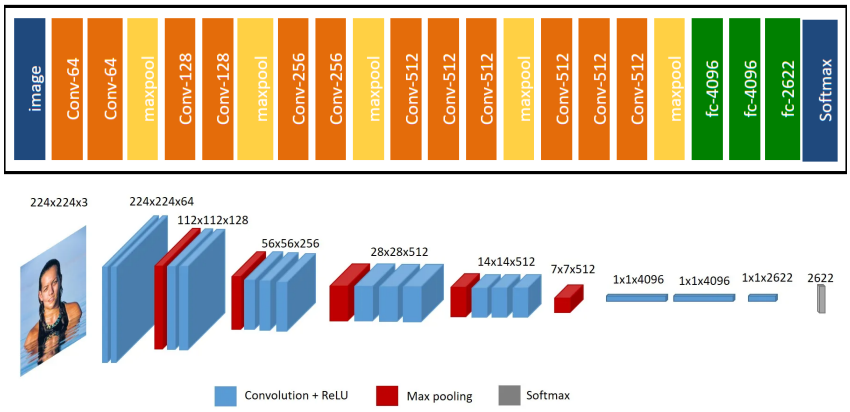

Table 1: Face Recognition Model Performance

Model	**LFW Score (%)**	**YTF Score (%)**
Facenet512	99.65	-
SFace	99.60	-
ArcFace	99.41	-
Dlib	99.38	-
Facenet	99.20	-
VGG-Face	98.78	97.40
Human-beings	97.53	-
OpenFace	93.80	-
DeepID	-	97.05

4. **Data Storage and Retrieval:**

- Visitor and user data are stored securely in their respective databases.
- Historical visitor logs are retrieved from the Visitor Database for display in the Visitor History Interface.

5. **Authentication and Authorization:**
    - The Security Layer ensures that only authenticated and authorized users can access sensitive features.

6 API Documentation and Endpoints Details

6.1 List User's Visitor

1. **API Description:** This API endpoint provides a list of visitors for the current logged-in user. It supports the HTTP GET method and returns a list of `Visitor DTOs` representing the visitors.
2. **Endpoint Details:**
    - **URL:** `/api/user/visitors/get`
    - **Method:** `GET`
3. **Authentication:**
    - `Authorization` header required with valid user credentials.
4. **Request Body:** The request for this endpoint is expected to be a simple GET request without a request body.
5. **Successful Response (200 OK):**
    - **Description:** The request was successful, and the server returns a list of `Visitor DTOs`.
    - **Response Body Data Types:** Visitor ID is integer representing the ID of the visitor, Visitor name is String representing the name of the visitor, Last visit stores the epoch time of the last visit.
    - **Example:**

```
{
  "visitors": [
    {
      "visitor_id": 1,
      "visitor_name": "John Doe",
      "last_visit": 1419086327
    },
    {
      "visitor_id": 2,
      "visitor_name": "Jane Doe",
      "last_visit": 1330650156
    }
  ]
}
```

6. **Error Responses:**

   (a) **404 Not Found:**

   - **Description:** The user was not found or no visitors were found for the current logged-in user.
   - **Example:**

```
{
  "error": "Not Found",
  "message": "User not found or no visitors found for the current u
}
```

   (b) **401 Unauthorized:**

   - **Description:** The request is missing valid authentication credentials or the provided credentials are invalid.
   - **Example:**

```
{
  "error": "Unauthorized",
  "message": "Authentication credentials are missing or invalid."
}
```

   (c) **400 Bad Request:**

   - **Description:** The request is malformed or contains invalid parameters.
   - **Example:**

```
{
  "error": "Bad Request",
  "message": "Invalid parameters in the request."
}
```

7. **Examples:**

   (a) **Request Example:**

   - **URL:** `https://URL/api/user/visitors/get`
   - **Method:** `GET`

   (b) **Successful Response Example (200 OK):**

```
{
  "visitors": [
    {
      "visitor_id": 1,
      "visitor_name": "John Doe",
      "last_visit": 1330650156
    },
    {
      "visitor_id": 2,
      "visitor_name": "Jane Doe",
```

```
        "last_visit": 1419086327
      }
    ]
}
```


(c) **Error Response Example (404 Not Found):**


```
{
  "error": "Not Found",
  "message": "User not found or no visitors found for the current user."
}
```


(d) **Error Response Example (401 Unauthorized):**


```
{
  "error": "Unauthorized",
  "message": "Authentication credentials are missing or invalid."
}
```


(e) **Error Response Example (400 Bad Request):**


```
{
  "error": "Bad Request",
  "message": "Invalid parameters in the request."
}
```


6.2 Get User Information

1. **API Description:** This API endpoint retrieves the information of the current logged-in user. It supports the HTTP GET method and returns a list of User DTOs representing the user's information. If the user is not found, it returns a 404 error code.

2. **Endpoint Details:**

   - **URL:** `/api/user/info/get`
   - **Method:** `GET`

3. **Authentication:**

   - `Authorization` header required with valid user credentials.

4. **Request Body:** The request for this endpoint is expected to be a simple GET request without a request body.

5. **Successful Response (200 OK):**

   - **Description:** The request was successful, and the server returns a list of User DTOs containing the information of the current logged-in user.

- **Data Types:**
  - User DTO:
    * `user_id` (int): User ID
    * `username` (str): Username
    * `email` (str): Email address
    * `full_name` (str): Full name
- **Example:**

```
{
  "users": [
    {
      "user_id": 1,
      "username": "john_doe",
      "email": "john.doe@example.com",
      "full_name": "John Doe"
    }
  ]
}
```

6. **Error Responses:**

   (a) **404 Not Found:**
   - **Description:** The user was not found.
   - **Example:**

```
{
  "error": "Not Found",
  "message": "User not found."
}
```

   (b) **401 Unauthorized:**
   - **Description:** The request is missing valid authentication credentials or the provided credentials are invalid.
   - **Example:**

```
{
  "error": "Unauthorized",
  "message": "Authentication credentials are missing or invalid."
}
```

   (c) **400 Bad Request:**
   - **Description:** The request is malformed or contains invalid parameters.
   - **Example:**

```
{
  "error": "Bad Request",
  "message": "Invalid parameters in the request."
}
```

6.3 Add Visitor Endpoint

1. **API Description:** This API endpoint adds visitors. It supports the HTTP POST method and takes a request body containing a `VisitorDTO` and an image file of the person. The server returns a 400 error code if the deepface model does not detect any face; otherwise, it is executed successfully. An image file is saved in the "datasets" folder, and the path to it is returned as a string.

2. **Endpoint Details:**

   - **URL:** `/api/add_visitor`
   - **Method:** `POST`

3. **Authentication:**

   - `Authorization` header required with valid user credentials.

4. **Request Body:**

   - **Data Types:**
     - `VisitorDTO` (object): Visitor data transfer object.
       - `id` (int): Visitor ID.
       - `name` (str): Visitor's name.
       - `relation` (str): Relation to the user.
       - `date` (str): Date of the visit.
     - `image` (file): Image file of the person.

5. **Successful Response (200 OK):**

   - **Description:** The request was successful, and the server adds the visitor. An image file is saved in the "datasets" folder, and the path to it is returned as a string.
   - **Data Types:**
     - `image_path` (str): Path to the saved image file.
   - **Example:**

```
{
  "image_path": "/datasets/visitor_images/visitor123.jpg"
}
```

6. **Error Responses:**

   - **400 Bad Request:**
     - **Description:** The deepface model did not detect any face in the provided image.
     - **Example:**

```
{
  "error": "Bad Request",
  "message": "No face detected in the provided image."
}
```

6.4 Sign Up Endpoint

1. **API Description:** This API endpoint handles user sign-up. It supports the HTTP POST method and generates an authentication token. If the sign-up is successful, it returns the token for the frontend. If not, it returns a 400 error code.

2. **Endpoint Details:**

   - **URL:** `/api/users/signup`
   - **Method:** `POST`

3. **Request Body:**

   - **Data Types:**
     - `username` (str): User's username.
     - `password` (str): User's password.
     - `email` (str): User's email address.
     - `phone_number` (str): User's phone number.
     - `full_name` (str): User's full name.
     - `date_of_birth` (str): User's date of birth.
     - `city` (str): User's city.
     - `country` (str): User's country.
     - `gender` (str): User's gender.
     - `address` (str): User's address.

4. **Successful Response (200 OK):**

   - **Description:** The request was successful, and the server creates a new user account. It generates an authentication token and returns the token for the frontend.
   - **Data Types:**
     - `username` (str): User's username.
     - `token` (str): Authentication token for the signed-up user.
   - **Example:**

     ```
     {
       "username": "john_doe",
       "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxLCJleHA
     }
     ```

5. **Error Responses:**

   - **400 Bad Request:**
     - **Description:** The request is malformed, or the provided credentials (username or email) already exist.
     - **Example:**

```
{
  "error": "Bad Request",
  "message": "Invalid parameters in the request or user with the sa
}
```


6.5 Login Endpoint

1. **API Description:** This API endpoint handles user login. It supports the HTTP POST method and validates and verifies the user's entered password. It returns success with the username and token if the login is successful; otherwise, it returns a 401 Unauthorized status.

2. **Endpoint Details:**

   - **URL:** `/api/users/login`
   - **Method:** `POST`

3. **Request Body:**

   - **Data Types:**
     - `username` (str): User's username.
     - `password` (str): User's password.

4. **Successful Response (200 OK):**

   - **Description:** The request was successful, and the server validates and verifies the user's entered password. It returns success with the username and token if the login is successful.
   - **Data Types:**
     - `data` (str): Greeting message.
     - `token` (str): Authentication token for the logged-in user.
   - **Example:**


```
{
  "data": "Hello, john_doe!",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxLCJleHA
}
```


5. **Error Responses:**

   - **401 Unauthorized:**
     - **Description:** The provided credentials are invalid, or the user does not exist.
     - **Example:**


```
{
  "error": "Unauthorized",
  "message": "Invalid credentials."
}
```

6.6 Reset Password Endpoint

1. **API Description:** This API endpoint sends an email to the user for resetting their password. It supports the HTTP POST method and generates a random password, sends it to the user's email, and updates the password in the database. It returns a 403 error if the user does not exist, a 401 error if the user does not have an email associated, and a 201 success status otherwise.

2. **Endpoint Details:**

   - **URL:** `/api/users/reset_password`
   - **Method:** `POST`

3. **Request Body:**

   - **Data Types:**
     - `username` (str): User's username.

4. **Successful Response (201 Created):**

   - **Description:** The request was successful. An email with a generated password has been sent to the user, and the password in the database has been updated.
   - **Example:**

     ```
     "Message Sent Successfully"
     ```

5. **Error Responses:**

   - **403 Forbidden:**
     - **Description:** The user with the provided username does not exist.
     - **Example:**

       ```
       {
         "error": "Forbidden",
         "message": "User not found."
       }
       ```

   - **401 Unauthorized:**
     - **Description:** The user does not have an associated email address.
     - **Example:**

       ```
       {
         "error": "Unauthorized",
         "message": "User does not have an email address."
       }
       ```

6.7 Get Image Endpoint

1. **API Description:** This API endpoint helps in uploading a user's photo. It supports the HTTP GET method and retrieves the specified image file from the server's "uploads" directory. If the file is found, it is sent as an attachment; otherwise, a 404 error is returned.

2. **Endpoint Details:**

   - **URL:** `/uploads/<filename>`
   - **Method:** `GET`

3. **URL Parameters:**

   - **filename** (str): The name of the image file to retrieve.

4. **Successful Response (200 OK):**

   - **Description:** The request was successful, and the server retrieves and sends the specified image file as an attachment.
   - **Data Types:** Binary data (image file).

5. **Error Responses:**

   - **404 Not Found:**
     - **Description:** The specified image file was not found in the server's "uploads" directory.
     - **Example:**

```
{
  "error": "Not Found",
  "message": "Image file not found."
}
```

6.8 Edit Visitor Details Endpoint

1. **API Description:** This API endpoint handles the edit visitor feature. It supports the HTTP POST method and is protected by user authentication. The endpoint allows the user to edit the information of a visitor, including their name, relationship, and image. The edited information is stored in the database, and the server returns a 401 error if the image is not provided or if no face is detected in the image. On success, it returns a 201 status.

2. **Endpoint Details:**

   - **URL:** `/api/user/visitors/edit`
   - **Method:** `POST`

3. **Authentication:**

   - `Authorization` header required with valid user credentials.

4. **Request Body:**
   - **Form Data:**
     - `name` (str): Visitor's name.
     - `relation` (str): Relationship to the user.
     - `id` (str): Visitor's ID.
     - `image` (file): Image file of the visitor.
5. **Successful Response (201 Created):**
   - **Description:** The request was successful. The information of the visitor has been edited, and the changes are stored in the database.
   - **Example:**

     ```
     "Visitor edited successfully"
     ```

6. **Error Responses:**
   - **401 Unauthorized:**
     - **Description:** The user is not authenticated or authorized to access the endpoint.
     - **Example:**

       ```
       {
         "error": "Unauthorized",
         "message": "Authentication required."
       }
       ```

   - **400 Bad Request:**
     - **Description:** The image is not provided, or no face is detected in the image.
     - **Example:**

       ```
       {
         "error": "Bad Request",
         "message": "No face detected in the provided image."
       }
       ```

6.9 Delete Visitor Endpoint

1. **API Description:** This API endpoint handles the delete visitor feature. It supports the HTTP DELETE method and is protected by user authentication. The endpoint allows the user to delete a visitor by providing the visitor's ID. The visitor's information is removed from the database, and the server returns a 201 status on success.
2. **Endpoint Details:**

- **URL:** `/user/visitors/delete/<id>`
- **Method:** `DELETE`

3. **Authentication:**
   - `Authorization` header required with valid user credentials.
4. **URL Parameters:**
   - **id** (int): The ID of the visitor to be deleted.
5. **Successful Response (201 Created):**
   - **Description:** The request was successful. The information of the specified visitor has been deleted from the database.
   - **Example:**

```
{"message": "Visitor deleted successfully"}
```

6. **Error Responses:**
   - **401 Unauthorized:**
     - **Description:** The user is not authenticated or authorized to access the endpoint.
     - **Example:**

```
{
  "error": "Unauthorized",
  "message": "Authentication required."
}
```

6.10 Saving Settings Endpoint

1. **API Description:** This API endpoint handles the settings page where a user can save changes to their user details. It supports both the HTTP GET and POST methods and is protected by token authentication. The endpoint allows the user to update various user details, such as username, password hash, email, phone number, full name, city, and address. The changes are saved in the database, and the server returns a 201 status on success.
2. **Endpoint Details:**
   - **URL:** `/change_data`
   - **Methods:** `GET, POST`
3. **Authentication:**
   - `Authorization` header required with a valid token.
4. **Request Body (POST Method):**

- `username` (str): New username for the user.
- `passhash` (str): New password hash for the user.
- `email` (str): New email address for the user.
- `number` (str): New phone number for the user.
- `full_name` (str): New full name for the user.
- `city` (str): New city for the user.
- `address` (str): New address for the user.

5. **Successful Response (201 Created):**
   - **Description:** The request was successful. The user details have been updated and saved in the database.
   - **Example:**

     ```
     {"message": "User data updated successfully"}
     ```

6. **Error Responses:**
   - **401 Unauthorized:**
     - **Description:** The provided token is invalid or missing.
     - **Example:**

       ```
       {
         "error": "Unauthorized",
         "message": "Token authentication required."
       }
       ```

6.11 Retrieving User's details as Placeholder for Settings Endpoint

1. **API Description:** This API endpoint serves as a placeholder when a user first enters the settings page. It supports the HTTP GET method and is protected by token authentication. The endpoint retrieves and returns the user's already signed-up details as a JSON response.
2. **Endpoint Details:**
   - **URL:** `/settings`
   - **Method:** `GET`
3. **Authentication:**
   - `Authorization` header required with a valid token.
4. **Successful Response (200 OK):**
   - **Description:** The request was successful. The user's already signed-up details are retrieved and returned in a JSON format.

- **Data Types:** JSON object
- **Example:**

```
{
  "username": "john_doe",
  "passhash": "hashed_password",
  "email": "john.doe@example.com",
  "phone_number": "+1234567890",
  "full_name": "John Doe",
  "date_of_birth": "1990-01-01",
  "city": "City",
  "country": "Country",
  "gender": "Male"
}
```

5. **Error Responses:**

- **404 Not Found:**
  - **Description:** The user data could not be loaded.
  - **Example:**

```
{
  "error": "failed to load user data"
}
```

- **401 Unauthorized:**
  - **Description:** The provided token is invalid or missing.
  - **Example:**

```
{
  "error": "Unauthorized",
  "message": "Token authentication required."
}
```

6.12 Getting Visitors' History Endpoint

1. **API Description:** This API endpoint returns the user's visitor history. It supports the HTTP GET method and takes the user's ID as a parameter. The endpoint retrieves and returns the visitor history of the specified user as a JSON response.

2. **Endpoint Details:**

- **URL:** `/history/<user_id>`
- **Method:** `GET`

3. **URL Parameters:**

- **user_id** (int): The ID of the user whose visitor history is to be retrieved.

4. **Successful Response (200 OK):**

   - **Description:** The request was successful. The user's visitor history is retrieved and returned as a JSON array.
   - **Data Types:** JSON object
   - **Example:**

```
{
  "visitors": [
    {
      "id": 1,
      "name": "Visitor 1",
      "relationship": "Friend",
      "embedding": "embedding_data_1",
      "last_visited": "2023-01-01 12:00:00"
    },
    {
      "id": 2,
      "name": "Visitor 2",
      "relationship": "Family",
      "embedding": "embedding_data_2",
      "last_visited": "2023-01-02 10:30:00"
    }
  ]
}
```

5. **Error Responses:**

   - **404 Not Found:**
     - **Description:** The specified user is not found.
     - **Example:**

```
{
  "error": "User not found"
}
```

6.13 Livestream Video Feed Endpoint

1. **API Description:** This API endpoint serves the live stream video feed. It supports the HTTP GET method and takes the user's ID as a parameter. The endpoint returns a continuous stream of video frames as a multipart response.

2. **Endpoint Details:**

   - **URL:** `/video/<user_id>`
   - **Method:** `GET`

3. **URL Parameters:**

- **user_id** (int): The ID of the user for whom the live stream video feed is requested.

4. **Successful Response (200 OK):**

   - **Description:** The request was successful. The server returns a continuous stream of video frames in a multipart response.
   - **Data Types:** Multipart response
   - **Example:** (No example provided due to the nature of video stream response)