IMAGELOCK AUTHENTICATION

Group 12

Jeremy Abel, Colin Childers, Danny Higgins,
Keegan Merck, Joe Onghena

1.0 Project Overview

Our Project is about how we will use multiple user authentication methods such as facial recognition to create a unique authentication method. This authentication method is based off of US patented US8655027B1 which is classified as a G06F18/00 pattern recognition patent. The goal of this patent is to create an authentication method that is safe and easy to remember for the end user by combining multiple authentication methods to make it harder for a potential threat to get into an account while making the authentication process memorable for a user and making it easy for them to login overall. During the account creation process the user types in their name, email, image category, an image offset pattern, locations on their face, and a rotation value. The login process starts with a user typing in their username, then they go through facial recognition. If their face matches their username they are directed to an image grid. On this grid are images from multiple categories. They will find 2 images from the category they originally picked out, then use the offset pattern to find the image they have to click. For example, if their offset pattern was 1 down and 1 to the right and their image category was cars, the user would find the picture of the car in the grid and then go 1 image down and 1 to the right to find the image they have to select. If the user selects 2 of the correct images, they are then brought to a page with an image over their face. They will then drag the image over the face location that they picked out and then rotate the image to correspond with the rotational value that they picked out during the signup process. If they get these parts correct they will be logged into the system. The potential stakeholders for this project are businesses that want to implement a multi-factor secure login process. We are working with the NSA on this project to get guidance and help create a functional model of this patent.

2.0 Architectural Overview

The most appropriate architectural design for the method of an image-based user authentication system is the "Model-View-Controller (MVC) Style." This choice is well-justified because the system has a multi-step process, using functions such as image capture, user data management, and facial recognition. The MVC style allows for these processes by enabling the division of these functions into separate components that can operate separately and undergo individual testing.

More specifically, the Model component within MVC is responsible for managing the backend aspects, including user data management, the execution of facial recognition, and the handling of image capture. It performs the bulk of the functionality in our system. On the other hand, the View component focuses on presenting information to the user, including tasks like image and grid display for authentication steps. The Controller is a connector, managing user interactions, ensuring data flow between the Model and View, and executing the authentication logic.

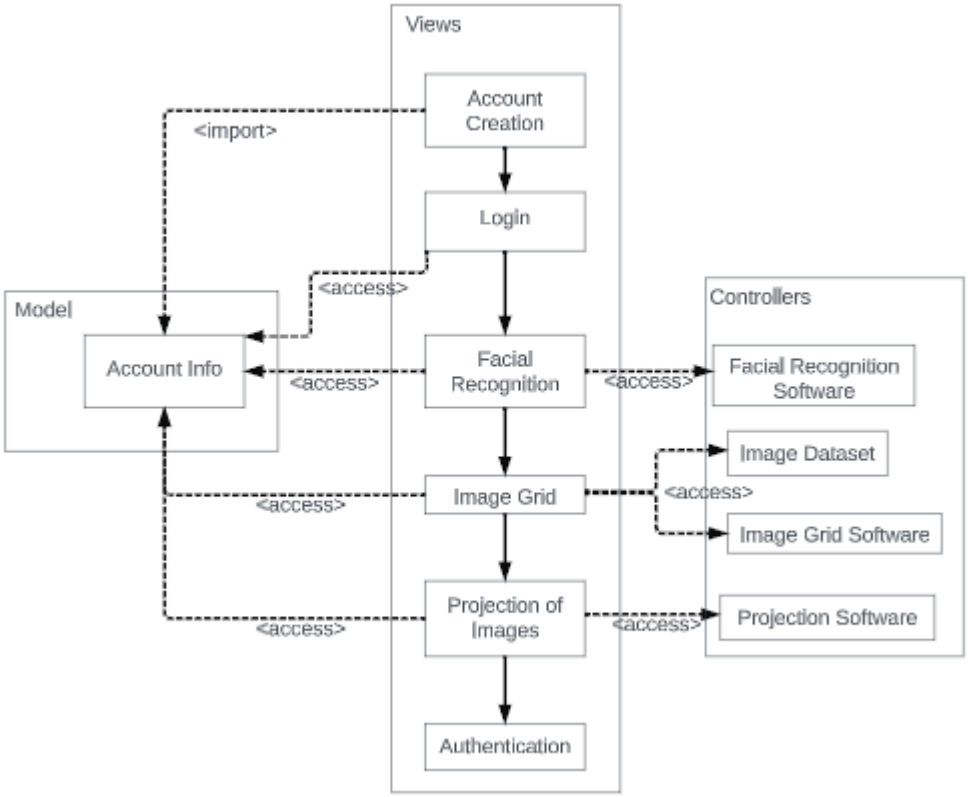
Other architectures were looked into for this project, however, most did not fit the application perfectly. For example, the "Layered Architectural Style" and "Shared Data Style" lack the requisite separation of concerns and modularity necessary for managing the multi-step process and fulfilling the user interaction requirements of the system. The "Pipe-and-Filter" style is more suited for data processing pipelines and may not effectively address user interactions and authentication logic. Similarly, the "Event-Driven" and "Brokers" styles could introduce unnecessary complexity, which is less than ideal for a system primarily centered on user authentication. In summary, the MVC Style emerges as the most fitting choice, owing to its ability to effectively manage complexity, separate concerns, facilitate user interactions, and promote flexibility.

2.1 Subsystem Architecture

In our system's subsystem dependency diagram, we have adopted the MVC architectural pattern, which provides a structured and efficient approach to organizing and managing our application. This pattern divides the system into three primary layers: Models, Views, and Controllers. The Models layer is responsible for managing account and user information, serving as the foundational layer of our system. It plays a crucial role as it stores and maintains essential user data, such as authentication details. Importantly, these models are accessible by both the Views and Controllers, making user data readily available for presentation and processing.

The Views layer encompasses the user interface components that allow users to interact

with the application. These views depend on the Models and Controllers to retrieve user-specific data and perform actions, ensuring the application runs as desired. Meanwhile, the Controllers form the middleware layer, housing most of the application's software. They act as intermediaries that facilitate the interaction between the Models and Views. They handle the business logic, user input, and authentication processes, effectively connecting the core data management and user interface layers.



2.2 Deployment Architecture

This software will run on a single processor.

2.3 Persistent Data Storage

In this project, we have opted to employ SQLAlchemy, a lightweight and versatile Object-Relational Mapping (ORM) library for Python, in conjunction with Flask-SQLAlchemy, to manage our database needs. This decision was driven by the relatively modest initial storage requirements of the application, which primarily involve the storage of user account information, including usernames, names, emails, image categories, and image paths for facial recognition. By utilizing SQLAlchemy, we could seamlessly interact with a relational database while keeping the development process agile and adaptable. The choice of SQLite as the underlying database system

aligns with the project's lightweight nature, ensuring that it suits our immediate needs. The usage of SQLAlchemy facilitates database operations and interactions, and it allows us to leverage the benefits of a relational database while keeping the system's resource demands well-suited to our project's scope.

The database schema for this project is designed to capture and organize the essential data related to user accounts and their associated images for facial recognition. The central component of this schema is the "User" table, where each row represents an individual user. The "id" column serves as a unique identifier for each user, ensuring data integrity and providing a means for efficient data retrieval. The "username" column stores the user's chosen username, which is both unique and mandatory, serving as the primary identifier for user authentication. The "name" and "email" columns hold the user's name and email address, respectively, providing additional user identification information. The "image category" column categorizes the user's associated image, helping organize data for facial recognition purposes. Finally, the last column is “image_path” which is responsible for holding the User's initial face capture when they create an account. This schema's design is streamlined, focusing on the specific data elements crucial for user management and facial recognition while maintaining simplicity and efficiency.

The "4155User.db" file serves as a self-contained database, enabling data to be conveniently stored, accessed, and managed within a single file, simplifying deployment and ensuring data persistence.

User
id (Integer, Primary Key): Unique identifier for each user.
username (String): Unique username for each user, not nullable.
name (String): Name of the user, not nullable.
email (String): Email address of the user, not nullable.
imagecategory (String): Category of the image associated with the user, not nullable.
image_path (String): Path to the user's image for facial recognition.

2.4 Global Control Flow

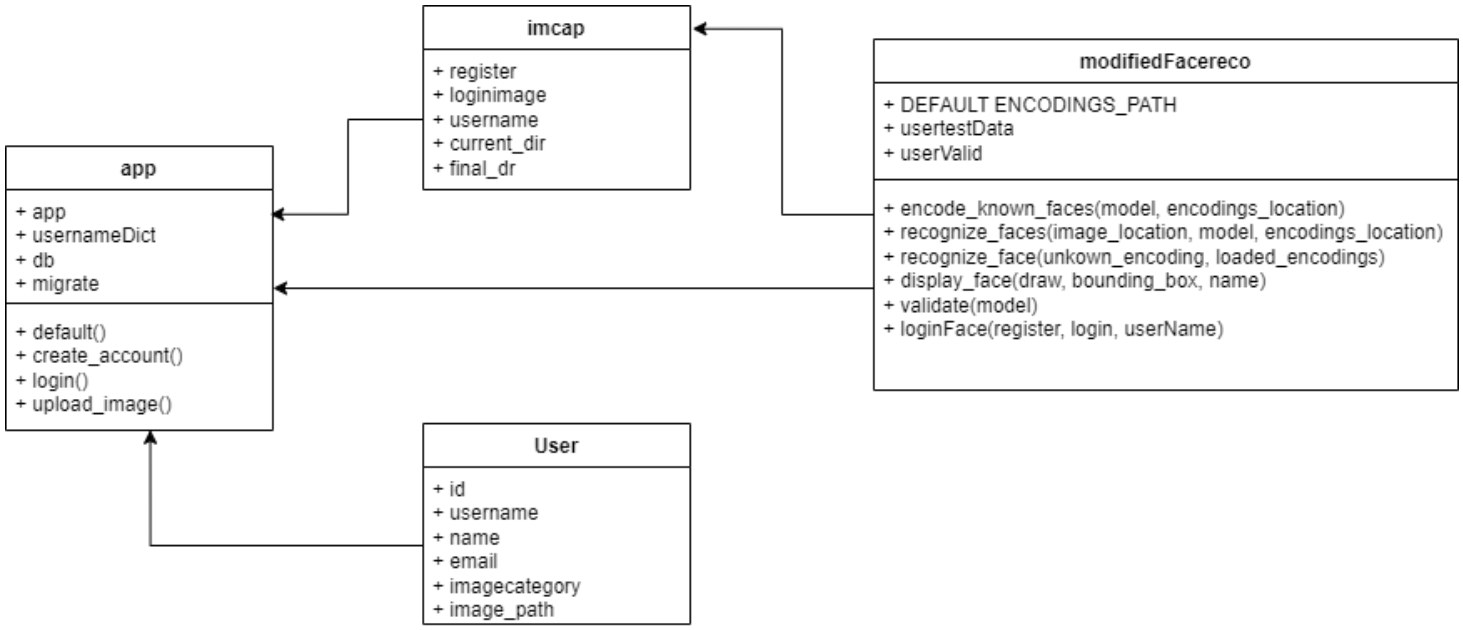
This project is procedure-based. When the user opens the project they always start on the login page. From that page, they can go to the signup page to sign up and upload their image. Once they are done uploading their images they are sent back to the login page. When they log in, they first type their username. Then they are brought to the face recognition page. If their face is

recognized they are brought to the image grid page. If they select the correct images, they are then brought to the image projection page. If they move the image to the correct part of their face and rotate it the correct amount of degrees, they are then logged in and the program just shows them a simple page saying their name. If they get any step wrong during the login process, they are sent back to the login page. There are no time-controlled actions in our system and there is no concern for real-time events. Our system does not utilize multithreading either.

3.0 Detailed System Design

For our project model, we have decided to create a Flask-SQLAlchemy application written in Python that implements a myriad of libraries for facial recognition as well as image projection. More specifically, we are using Flask as our backend framework to facilitate efficient communication between the user interface and the various image processing components, while SQLAlchemy serves as the bridge to interact with our relational database, ensuring seamless storage and retrieval of user data, including images and facial recognition results. Furthermore, our front end consists of HTML templates that the user can interact with, providing them with detailed instructions on what to do within the application. On top of this, we are styling the application using external CSS to make everything look more professional and approachable. We are also implementing Jinja and JS to add functionality and responsiveness to our application for things such as image selection as well as image captures.

3.1 Static view



As our application currently stands, app.py functions as a controller with all of the routes and database configurations being done within the class. The User class is currently contained within the app which is a database model that we use to store a user's information per person. The imcap.py and modifiedFacereco.py classes function as our current attempt to capture a user's face as well as recognize them once they are in the system. This being said, our future plans have us removing the imcap.py class and replacing it with a new method in app.py called upload. Overall,

when it is put all together we get an application that is capable of user creation that implements face scan and recognition libraries to give the user access to their account. In the future, we will implement image projection in tandem with facial recognition to add an additional layer of security which will likely include additional classes and libraries to be successful.

3.2 Dynamic view

The progression of the user experience within our system follows a straightforward and sequential path. Initially, users encounter a login screen, and since they likely don't have an existing account, they will navigate to the account creation page. Here, users can input their essential information, which will be stored in our SQLAlchemy database. Furthermore, they provide key data required for the image verification process, including the selection of an image category from a predefined set of options. Another pivotal step in the account creation process involves the user capturing a self-portrait as prompted by the website. After successfully creating their account, users are redirected back to the login page, initiating the verification process. In cases of incorrect data entry, the site prompts users to retry. Once data entry is successful, users are asked to capture their face again, and their login information and photo undergo verification. A successful verification allows users to proceed, while an unsuccessful attempt prompts them to repeat the login process. The final step involves users selecting two images related to their chosen category from an image grid featuring twenty different options. Correct selections lead users to the landing page, but if incorrect image types are chosen, a prompt informs them of the error. This sequential flow ensures a smooth and secure user experience.