August 20, 2015

Face Tag

HONOURS PROJECT FOR CARLETON UNIVERSITY

CHRISTOPHER DUFOUR - 100854402

Supervised by: Dr. Dwight Deugo

School of Computer Science

Carleton University

Ottawa, Canada

Abstract

Face Tag is a social network that integrates facial recognition to demonstrate meaningful relationships between users. It features a system to detect and recognize users in a social network, a search feature to search for images including specific users and a mobile app client that provides easy access to the social network features.

Acknowledgements

I would like to thank Dr. Dwight Deugo for his help in supervising and providing input towards creating a quality project. I would also like to thank my family at home for being understanding and helping me test and complete this project.

Contents

Introduction

Problem

Images are an important part of modern social networks, especially images of people. People like to express themselves and create memories by taking pictures of themselves and their friends. While humans are be able to easily extract useful information from a picture a computer has a much tougher time as all it sees is a mass of pixels. The problem I will be trying to solve is how a computer is able to generate useful information about the people in pictures, and what this information can tell us about those people, particularly when those pictures contain multiple people.

Motivation

My motivation for this project is to create a product that combines and demonstrates what I have learned over the past 4 years of my Computer Science degree. The intent of this project is to show what I have learned and how it can be applied. Additionally it will highlight which areas of computer science I personally find interesting and how they can be applied to real life situations.

This project was inspired by similar technologies such as Facebook's tagging and the facial recognition software in the OpenCV programming library. The ability to apply facial recognition to a social network setting seemed to be a novel way to represent relationships between people. If a user often takes pictures together with another user it can represent how close those people are better than simply if they are friends or not.

Goals

The initial goal is to prove that, using facial recognition, we can identify people in an image automatically. To do this a facial recognition system will have to be created. It must be able to lean to identify a person based on their face and tell who is in an image. As the images are not simply a picture of a person's face it will first have to find their face in the image, then run the recognizer to determine who they are.

The second goal is to be able to save the images of people it identified and be able to request galleries of specific people. For example I should be able to enter “Alex” as a search term and get a gallery of all the images I took where Alex is present. More complicated searches such as “Alex AND Ethan” can return images with both Alex and Ethan present.

The third goal would be to migrate the facial recognition and the storage to a server. This allows for better processing ability, removes the requirement for the app to be platform specific, and allow for more secure and controlled access to the images. Users are at this point required to create an account and log in to the server to access their images.

The fourth and final goal will be to evolve the server into a social network where the people in the images are the users and they can tag each other in their images using the facial recognition software. Limitations such as only being able to tag friends can guarantee privacy and will allow for the formation of groups, an important measure in gathering information on users. As it is a social network a client UI will be required to display and upload information to the server. For this a mobile app will be created to control the social network.

Objectives

1. Create a facial recognition system to learn people’s faces. After that it should be able to take an image, find all of the faces and correctly identify the people in the image.
2. Allow for searching by people in the images, returning all images with the specified people in them. This is the first step in identifying links between people through images. Advanced queries featuring keywords such as ‘AND’, ‘NOT’ and ‘ONLY’ can be used to narrow the focus of the query.

3. Store images and possibly perform facial recognition on a server, allowing for more storage available for the user, better and faster facial recognition and reduces the requirements of any client accessing the application. Users can be required to register an account on the server and will be required to log in to access the service. User data and images will be stored in a database running on the server. This will allow individual users to identify themselves and other in the images on the server.
4. Evolve the server to a social network. Users can also become friends with other users and tag those users in the images they upload. Users should be able to run queries like in #2 to be able to find images of various users and groups of users. Create a mobile application to serve as a client for the social network.

Background

Facial Detection

Object detection can be a powerful tool for getting information out of an image. Being able to determine the location and type of things in images is extremely useful, especially in a social context. This works by having the program 'learn' an object type by providing a large number of sample images and allowing the program to learn the various features of that kind of object. It can then later search an image for objects similar to the ones it has learned. In this case I plan to use it to find human faces in an image.

Knowing about the content of user's images can prove to be a good measure in determining various things about the uploader, such as demographic information, interests and values. Facial detection specifically can be used to determine what kind of image a user is uploading. We can often assume that images with people in them are images of the user themselves, or their friends and/or family, who

probably also have accounts on the social network. This can be combined with facial recognition to determine the identity of users for even more information.

Facial detection on its own does not have many applications as simply being able to know where in an image a face is isn't that important if we are not able to determine who that person is. When combined with facial recognition it becomes a powerful combination with many practical implications.

Facial Recognition

Facial recognition allows us to determine who a person is in a given image if we have learned from previous images. It is similar to facial detection in that it requires sample images to 'learn' individual faces, but as for a recognizer it is specifically designed to pick up on the small details in individual faces and tell you which person it believes the image is of, rather than telling you where in the image the face occurs.

When combined with facial detection this technology allows us to identify people in an image, which can be very useful as in a social network setting any data which tells us a personal connection between two users is valuable information.

Real life applications of facial recognition are generally based in biometrics as facial recognition is best used as a means to identify individuals. As said in the paper Face Recognition and its Applications: "Face recognition has the advantage of ubiquity and of being universal over other major biometrics, in that everyone has a face and everyone readily displays the face."[1] While it lacks the accuracy of stronger biometrics it does not require the same level of equipment to process a face in an image as it does a rental scan or a fingerprint. This principal of determining unique people is applied in this project as a means to determine the identity of people in images.

Facial Recognition in Social Networks

Facial recognition in social networks is, while not a common feature, a useful metric in determining user relationships with others. It is also a good feature for the users as it will let them search for images of a specific person or perform more complicated searches such as wanting images of where multiple people are present.

While not a common feature the ability to tag people in images has recently become a popular feature in social networks. Facebook has been a leader in this respect as it has implemented “tagging” images as a core part of interaction between friends. This has generated some controversy as many people believe that Facebook having the ability to identify you in images that you have not personally been tagged in is a breach of privacy [2]. Despite this it can be an extremely useful measure in determining relationships between users as people being found together in an image represents a real life relationship between people, or at the very least a shared interest.

RESTful Web Services

REST (Representational State Transfer) [3] is an architectural style in web service design which allows the application’s clients to retrieve resources from the service through URI requests. For example, if a client wanted the user account for the user “bob” they could send a HTTP GET request to the address “www.example.com/user/bob” and the service would return various account information for the user specified on the /user/ path.

RESTful design is very useful as it provides a mechanism to create a service without the need for a specific client implementation as all requests can be represented as requests to the server for data in a generic form before it is processed. This allows for different clients to be created for the same service without changing anything on the service itself.

Another feature of RESTful design is that it is stateless, in that requests are not required to come in any order, and as long as the request is valid the service can handle any request at any time for any

resource. This works well for social networks as there is often no need for users to perform operations in any specific order is they are requesting, creating or modifying resources on the service.

Android Mobile Application

Android is an operating system designed for mobile devices such as mobile phones and tablets. Android is a good platform for creating mobile applications on as it contains many features, both hardware and software, that developers can take advantage of. This project will be taking advantage of its strong ability to work with internet services and its inclusion of a camera and other image services.

Approach

For this project I have decided to divide the project into two sections: the server and the mobile application. The server will be responsible for controlling all the social network information, storing images and data as well as authenticating users and performing the image recognition. The mobile application will be a user friendly interface for the social network, providing a clean way to control the server though underlying requests to the server. They will interact with each other using HTTP requests that the server will reply to using JSON.

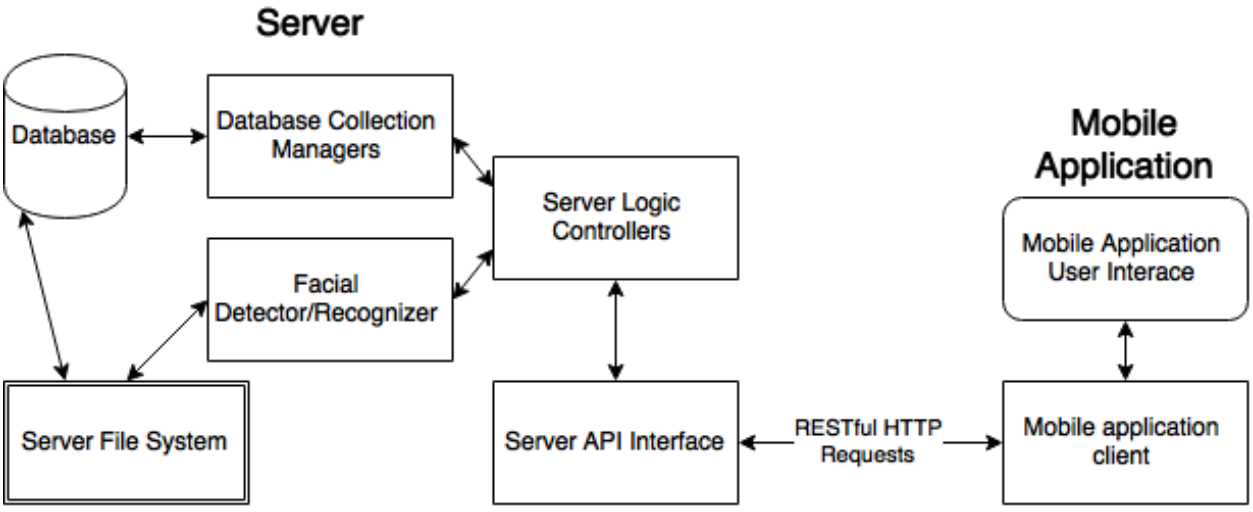

Figure 1 - Program Component Diagram

Libraries and Other Programs Used

Server:

MongoDB:

MongoDB is a document based database. Instead of storing data in tables it treats each entry as a document. Data is stored as kay value pairs similar to JSON objects. Each document has a unique ObjectId, a unique identifier created by the database automatically. Documents stored in MongoDB can be mapped to JSON objects easily and therefore it provides a simple way to take an object from a database and change it into a JSON string. [4]

Documents are stored in collections, similar to tables. Collections can be assigned an object class, allowing us to query the database and have Java objects returned instead of Maps or proprietary object types.

This project uses Collection Manager Classes as a wrapper around the MongoDB function calls to contain the query and library specific code so that there are no queries or database specific calls in the server logic. This project also takes advantage of the ObjectIDs that are automatically generated for all of its classes, using them as unique identifiers for images, users and rectangles.

Eclipse Dynamic Web Project and Jersey:

The server for this project is based off of the Eclipse Dynamic Web Project template and uses Jersey to create a RESTful architecture. An Eclipse Dynamic Web Project is a project template included in Eclipse EE Developer Tools that allows a user to create a webserver and have advanced control over how data is accessed and more resources to work with than with a static web project.

Jersey is an API framework that makes it easier to build RESTful web services by providing a simple API around the JAX-RS API [5]. For the purposes of this project it is used to provide a means of creating a RESTful web service by allowing us to control what we do with data that is received on various URLs.

Using Jersey we can have the server run logic on the input it receives on certain addresses and return specific output. For example if we receive a GET request on www.example.com/user/bob then Jersey will allow us to run code to retrieve the Bob user from storage and return it to the client in some form it can understand. In the case of this project all data received will be GET requests on specific URL paths with query parameters or POST requests on specific URL paths with parameters in the request body.

OpenCV:

OpenCV is a computer vision library that provides access to many computer vision algorithms and techniques. This project takes advantage of its facial detection and recognition algorithms. [6]

The facial detection functionality works based off of a trained Harr Cascade Classifier. OpenCV provides a pre trained classifier that can be used to perform facial detection on an image. All that is required it to load the training data and perform detection on an image and the result it a set of rectangles that contain what the algorithm believes to be human faces. This project uses this to detect faces in the images that users upload to create rectangle entities that are then sent to a facial recognizer to be recognized.

The facial recognizer is an algorithm that uses various techniques to 'learn' individual faces and then be able to guess which person out of the faces that it has learned is in a given image. This algorithm also returns a confidence value that shows how confident it is that a user is the same as the face in a given image. In this project we use it to recognize faces defined by the rectangles found by the face detector to identify which of the image uploader's friends are in the image. The results of the recognition are both presented to the user and also used to search for images based on the recognized users.

Spring:

Spring is a framework that allows developers to easily integrate a RESTful web service client into their apps. It allows us to build HTTP requests to send to the webserver and handle the output. [7]

In this project Swing is used to handle http requests between the server and the mobile application, allowing us to simply build the request, send it to the server, receive the response and then handle it. Responses from the server are received as strings, but we also receive other information such as the response code.

By overriding the build in error handler we are able to control behaviour in the error handler as well as for normal output. In the case of this app we process 400 and 404 error requests as if they were normal requests, allowing manual control over different returns from the server.

Jackson:

Jackson is a general purpose library for marshalling JSON data to java classes. While JSON conversion is built into MongoDB on the server side on android we use Jackson to convert JSON responses to classes parallel to those on the server. This allows us to send out entities between the server and the client easily. [8]

In this project we use Jackson to parse our entities to and from JSON data so that they can be sent to and from the server. For example: if we request a user profile from the server we can get the database entry for it, parse it to a JSON string and return that to the client. The client will then take the string and turn it back into a user profile and display that to the user.

Entities

The service uses several entities to store and represent different kinds of data. Most of these are sent back and forth between the server and the client, for example if a client wanted the profile for a user the profile information would be sent to the client as a User profile entity with their name, user Id and other information. This goes both ways as well, so if a client wants to upload an image they would send an image entity to the server with the image data, the image title and other information.

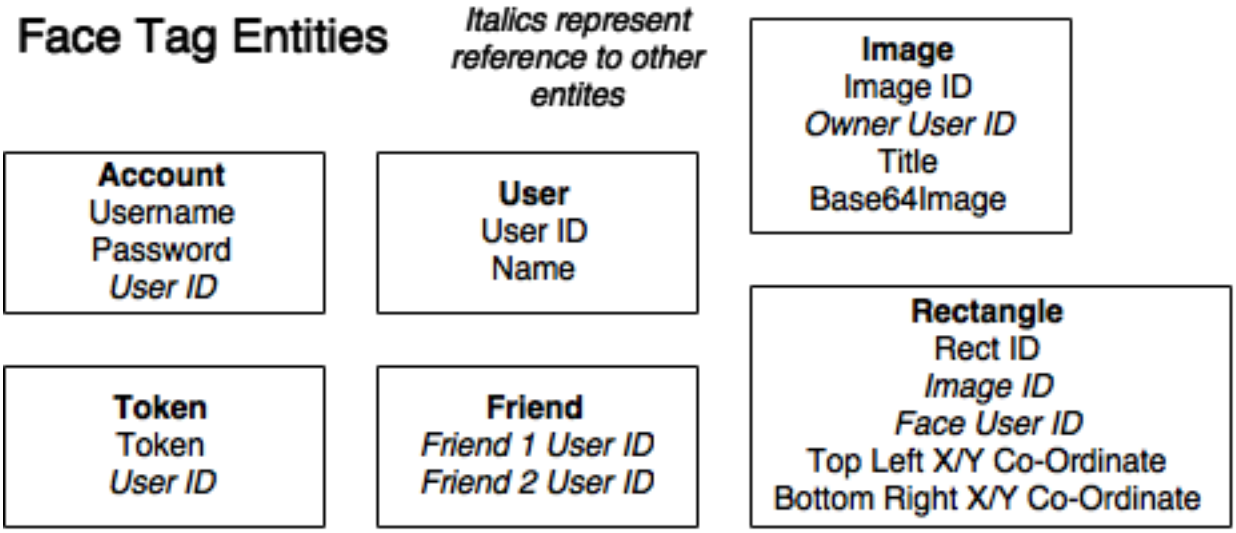

Figure 2 - Face Tag Entities

Account

Each user has an account which is a combination of their password and username. This account object will not be accessible to users and can only be created upon signing up and checked by the server when a user is logging in. It also contains a reference to that user's profile.

User Profile

Each user has a profile that contains their unique ID, known as a user ID, their name and any other information that would fit into a user profile. This is a user's public representation, as apart from their account which manages any information that should be only known to that user for security purposes.

Image

Images contain a unique ID, the image ID, the user ID of the user that uploaded the image and other information such as the title of the image.

When an image is being sent to the client or an image is being uploaded to the server it is stored as a Base64 string. This allows the data to be encoded, sent as a HTTP parameter or JSON string and then decoded at the other end. While this does increase the bandwidth of sending the image and adds a (potentially large) memory constraint on both the server and client it standardizes how data is sent and received in that data sent to the server will always be HTTP parameters and data sent to the clients will always be JSON.

The Base64Image string is never stored on the server as it is decoded and processed once it is proved to be properly formatted and should not be kept in string form on the client for very long as it can take up a lot of memory.

Friends

Friend entities are pairs of user ID to show that those two users are friends. Being friends allows users to find each other in their images using the recognizer. This means that if two users are friends and one uploads an image to be recognized then both users' recognizers will attempt to find those users in the image, but if they are not friends only the user that uploaded the image will be recognized.

Rectangles

Rectangles are the area where a face in an image was found and the user ID of the user in the image. They contain the x-y co-ordinates of the top left and bottom right corners of the rectangle were a face was detected, the ID of the user detected in the image, if any, and the image ID of the image it was found in. They also contain a unique rect ID that is used to identify them.

Tokens

Tokens are randomly generated keys that clients are given when logging in to show that they are acting as the valid user of that account and can make posts, add friends, etc… as the user. They are given to the user upon logging in and deleted when logging out. They are simply proof of identity from the server's perspective and must be sent along with the account's user ID for any action besides logging in and signing up.

These tokens can also act as a sort of access control, as users that are not friends with another user can be blocked from accessing another user's account information, images and other information depending on that user's settings.

API

The API is how clients send and receive data to and from the server. It is an interface based on HTTP requests that will perform commands on the server and return data to the client. This includes adding new accounts, performing facial recognition on images and any other interaction that a client must be able to perform.

If a request containing errors or a request that can't be processed is sent the server will return an error message explaining why that request couldn't be fulfilled.

All requests will send data to the server by specifying a path in the request showing what operation they want to perform (or to have performed) and they will receive a response. The data being sent can either be part of the path, showing which resource they want to access. For example, to get a user with a user ID of 333 we could send a GET request to the address www.example.com/user/333. To include more parameters to a request query parameters are used. For example, if we wanted to login user bob with password pass to a server we would send the following GET request to the address

www.example.com/login?username=bob&password=pass. This would return a token and the user ID for bob's account.

The table below illustrates all the different paths and request type combinations that can be performed, which are all mapped to different operations on the server. Returned data are all examples of data returned by successful requests. For data returned by unsuccessful requests see the following tables for error messages.

All paths are extensions of the server's address (for example: www.example.com/FaceTag/rest/). Values in parentheses i.e. {value} are values included in the path. Values with square brackets Object [] are an array of that object type. Any classes that are sent and received are sent as Strings of JSON data, including arrays. If no data is returned then the server simply sends a message confirming a positive outcome.

Action Name	*Request Method*	*Address Path*	*Sent Data*	*Returned Data*	*Purpose*
Sign Up	POST	/signup	String Username String Password String Name		Creates a new account if one for this user doesn't already exist.
Log In	GET	/login	String Username String Password	String userID Integer token	If the username and password combination are correct then send the user their userID and token, allowing

					them to act as that user.
Log Out	POST	/logout	String userID Integer token		Deletes the token/user ID combination, effectively logging that user out from that session.
Get all users	GET	/user	String _id Integer token	User[] allUsers	Returns user objects for all users on the server.
Get user	GET	/user/{targetUserID}	String userID Integer token {String targetUserID }	User user	Returns a user object for the user with the user ID equal to { targetUserID }.
Get user's friends	GET	/user/{targetUserID}/friend	String userID Integer token {String targetUserID }	User[] allFriends	Returns an array of User objects that represent the profiles of all of the users that the target user is friends with.
Add friend	POST	/user/{targetUserID}/friend	String userID Integer token {String targetUserID }	User newFriend	Add the target user as a friend. If a new friendship is created

					then return the user object of that user.
Delete friend	POST	`/user/{targetUserID}/friend/delete`	String userID Integer token {String targetUserID }	User deletedFriend	Removes friend. Returns the info of the friend you just removed.
Get user's images	GET	`/user/{targetUserID}/images`	String userID Integer token {String targetUserID }	Image[] userImages	Request an array of all the images a user has. The base 64 image data will not be sent.
Get image	GET	`/image/{imageID}`	String userID Integer token {String imageID }	Image image	Requests the image on the server with the image ID {imageID}. Returns that image with the base64 image data as an Image object.
Delete image	POST	`/image/{imageID}/delete`	String userID Integer token {String imageID }		Deletes the image with the corresponding imageID.
Post Image	POST	`/image`	String userID Integer token String base64Image	Image image	Posts a new image using the current account specified by userID. This new image

			String title		will have the encoded data from the base64Image and the title from the title variable. When the server is done uploading the image it will return the new image's Image object without the base64Image (as it assumes the client already has it).
Search	GET	`/image/search`	String userID Integer token String[] include String[] exclude	Image[] images	Searches for images that contain all the users (by user ID) in include and do not contain any of the users in (exclude). It will return an array of images that satisfy these conditions.

Get rectangles	GET	/recognize/{imageID}/rectangles	String userID Integer token {String imageID }	Rectangle[] rectangles	Returns all the rectangle objects for the requested image.
Set rectangles	POST	/recognize/{imageID}/rectangles	String userID Integer token {String imageID } Rectangle[] rectangles	Rectangle[] rectangles	Request that the following rectangles be updated to the values set. The new rectangles are sent contain the rect ID of the rectangle they are updating. The old rectangles will then take on the values of the new rectangles and the recognizers will learn the new rectangles for their respective user. The updated rectangles are returned to the user.
Generate rectangles	GET	/recognize/{imageID}/rectangles/new	String userID Integer token {String imageID }	Rectangle[] rectangles	Uses the recognizer to generate new rectangles for the image with the

		matching image ID. This will run the face detector to find where in the image people's faces are then run all of the current user's friend's recognizers on the faces. The user that most closely matches that face will be set in that rectangle and all of the rectangles with their estimations will be returned.

Table 1: Face Tag API specification

Server

For this project I have decided to program my own server instead of using an online service as it will allow me to specify the specific implementations of various elements in the program as well as have very precise control on what responses the server will return to the client when a request is made.

Database and Server File System

The server's database stores all the social network and recognizer information in tables. The recognizer information and the images are stored in the server's file system as they are not well suited to being

stored as database entries. They are accessible using the image's id as a file name and the recognizer information is named after the user it is representing.

The file system is organized as such:

Files_Root – The location on the drive where Face Tag's file system data is stored. The server should have sufficient privileges to access this location.

This location also contains any data that the face detector needs to function.

Files_Root/images – The location where all raw image data is stored. All images are stored as jpeg files without file extensions. The name of the image file should be the same image ID as in the database.

Files_Root/recognizer – The location on the dive where every user's recognizer data is stored. Each user has a recognizer and any data that the recognizer needs to save is stored as a file with that user's user ID as the file name.

Below is a table explaining every database collection and their purpose.

Unique (*) entries are entries that can't be repeated, such as a userID.

Unique pairs ([1]data1, [1]data2) are pairs of data where that specific pair can only occur once.

ObjectID is the MongoDB class for the automatically generated ID for entries. It automatically takes the field name _id. This table will not show them for collections where they are not relevant. References to IDs are ObjectIDs that have the same value as the _id value is the respective class.

Collection	*Purpose*	*Structure*

Account	Store all account data. Contains a userID field which is the userID of the corresponding user entity.	*	String username String password * ObjectID userID
User	Store all user profile data.	*	ObjectID _id String name
Image	Store all image meta data (not the image data). Contains an ownerID field which is the ID of the user who uploaded the image.	*	ObjectID _id ObjectID ownerID String title
Token	Store all currently active tokens.		ObjectID userID Integer token
Friend	Store all pairs of users that are friends.	[1]	ObjectID userID1 [1] ObjectID userID2
Rectangle	Store all rectangles that have been identified. userID is the ID of the user that was identified in the image. imageID is the ID of the image where the rectangles were found. x1 and y1 form the top left x-y co-ordinate of the rectangle and x2 and y2 are the respective bottom right corner.		Integer x1 Integer y1 Integer x2 Integer y2 ObjectID userID ObjectID imageID

Table 2: Database Collections, their purpose and a list of fields in each collection

Database Collection Managers

Database collection managers provide a programming wrapper around the raw database commands. These collection managers provide functionality to the server logic controllers so that the server logic doesn't have to manage data base connections, queries and other technicalities. This also provides

flexibility in how the database can be implemented as the server logic doesn't need to know what is going on at that level.

Collection managers are implemented following the singleton pattern as to maintain a single and constant connection to the database and its collections. As soon as the first request to access a database element the database initializes and connects to the database. After that any request to access database elements will go through that connection.

Facial Recognizer / Detector

This component is responsible for processing images and returning the location of the faces in those images and its guesses as to who is in those images. It is also responsible for managing and updating the recognizer data saved on the file system.

In order to recognize a face the recognizer system first detects the locations of all the faces in an image then tries to find out which of the uploading user's friends match which detection. Each user has a recognizer object that can take a face and return how close that face is to that user's face. The user with the best relation to the image is returned to the server logic as the person in the image as a rectangle entity.

The recognizer is also responsible for updating the recognizers when the user either confirms that the face was detected correctly or updates recognizers with corrected data.

Server Logic Controllers

The server logic controllers are responsible for the main functionality of the web application. The take arguments from the client's request, process them, and then may retrieve data from the database, perform recognitions on images using the recognizer, update the recognizer, and/or perform any other specific tasks. After that it will return data to the API manager to be sent back to the client.

Any error checking on arguments sent by the client is also performed here. In the event that a call with invalid information, such as a formatting error in the call, invalid user credentials, an invalid token or any other issue the logic controller will handle the response to the client, providing information on what was wrong with the request.

Server API Interface

The API interface is the component that contains the networking inputs that receive and respond to HTTP requests. Which requests and how they are responded defines the API that the client will use. This component will receive the data from clients, structure it as arguments and call upon the Server Logic Controllers to process it and generate a response. The interface will then return that data as a response to the client's request.

Server Handling of API Requests

Below is a series of charts demonstrating the process that the server uses when it receives an API request on the API interface and how it handles these requests and returns the expected data.

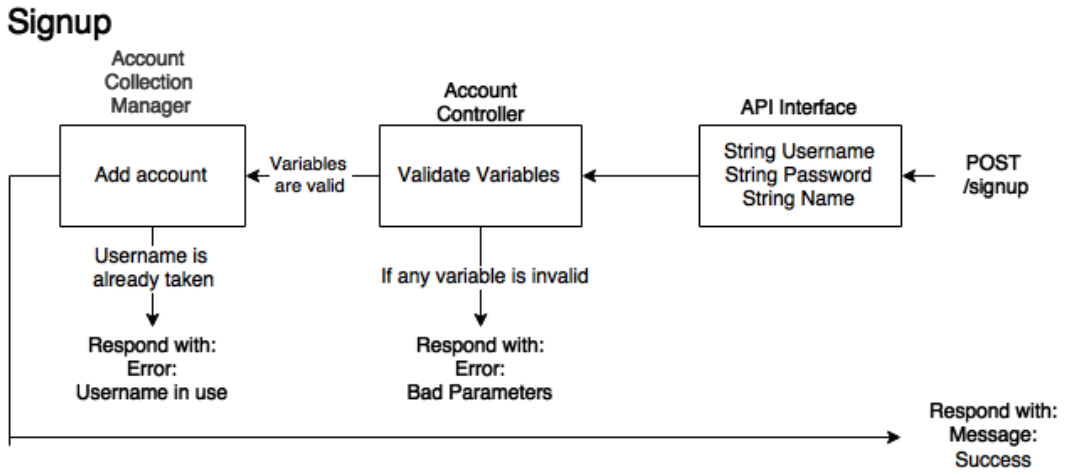

Figure 3: Sign up action – server logic

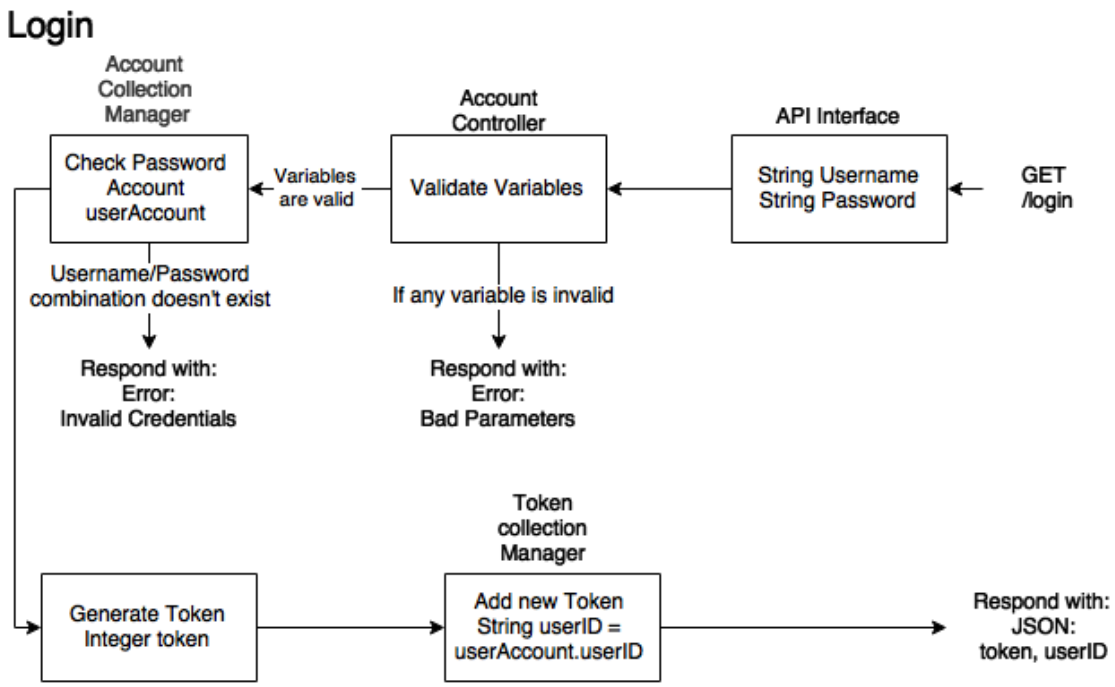

Figure 4: Log in action – server logic

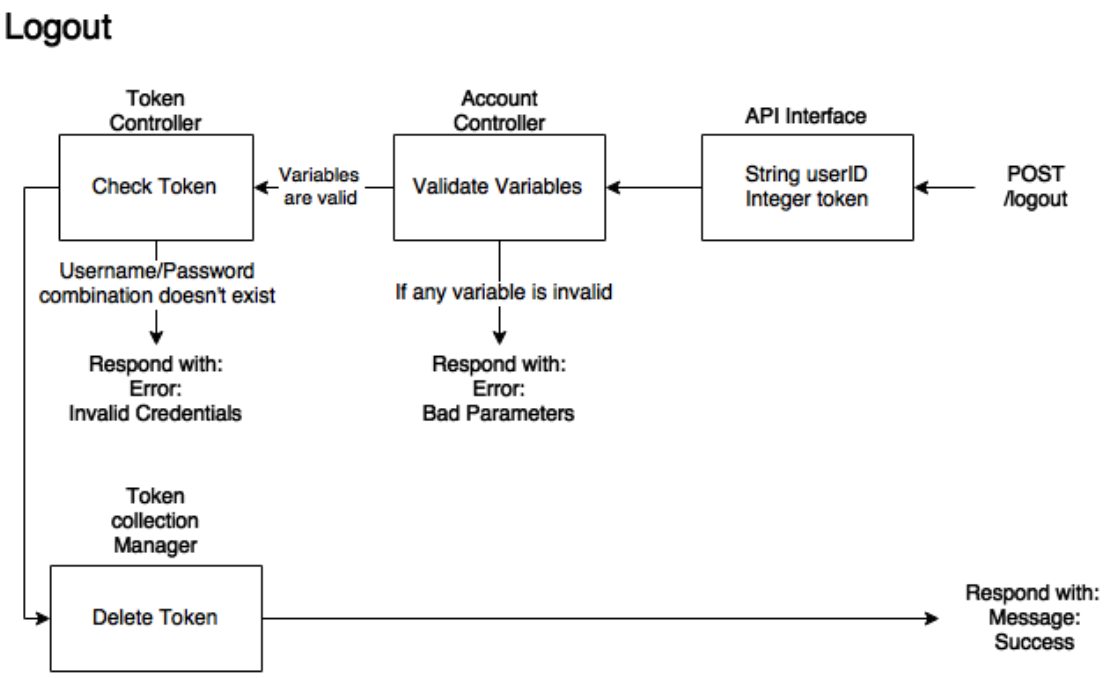

Figure 5: Log out action – server logic

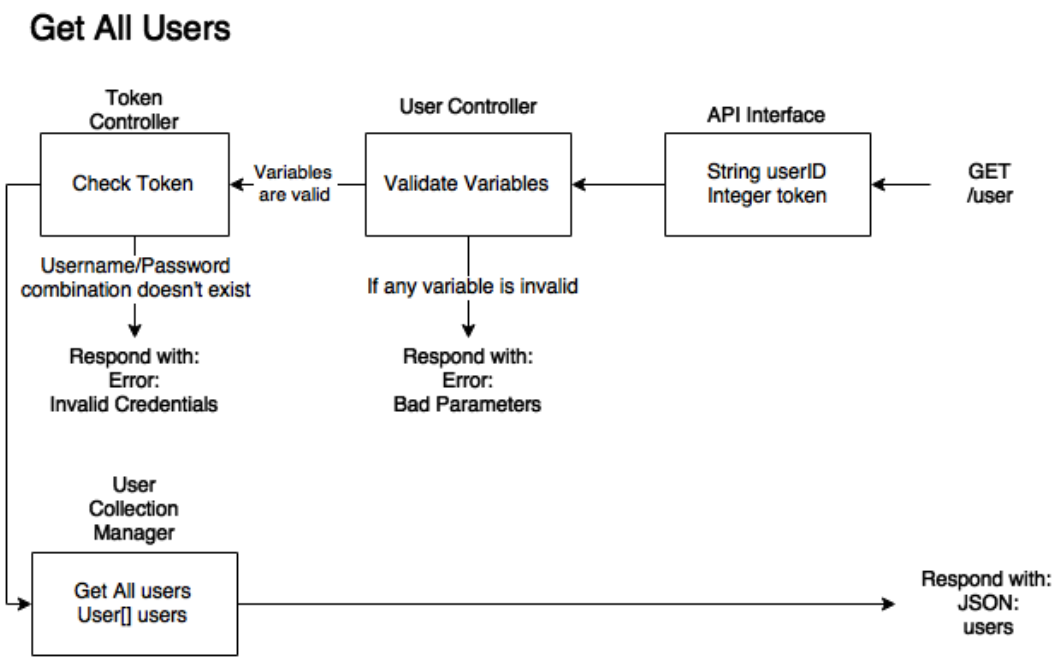

Figure 6: Get all users – server logic

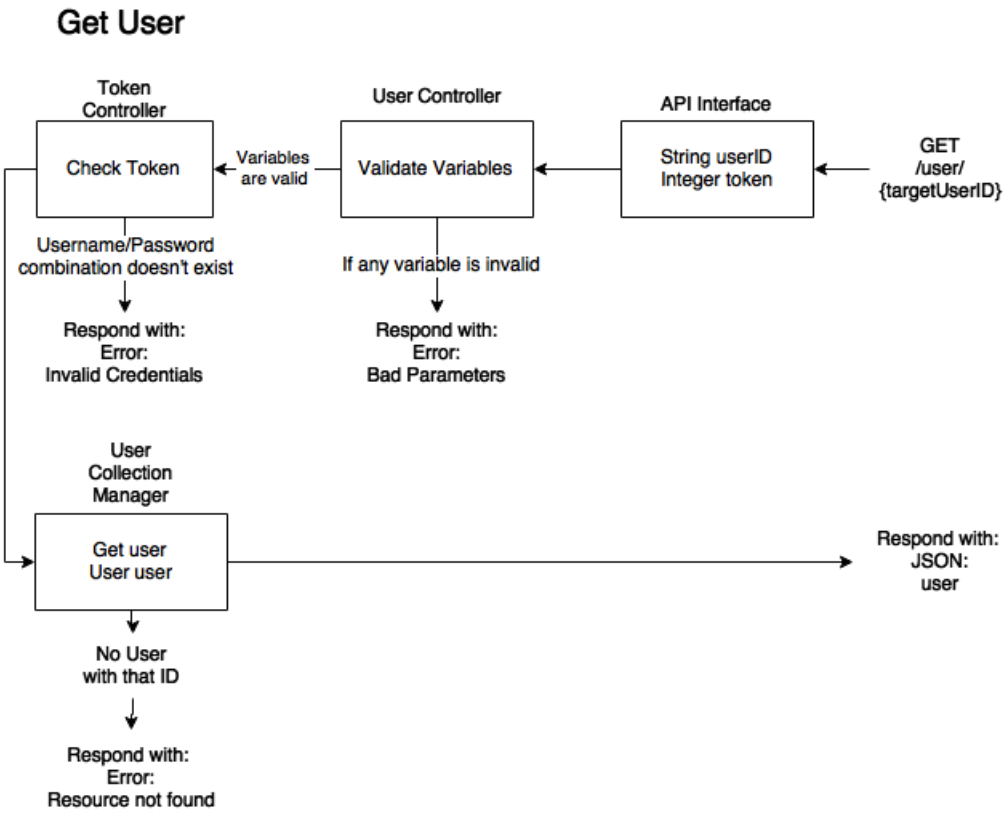

Figure 7: Get user – server logic

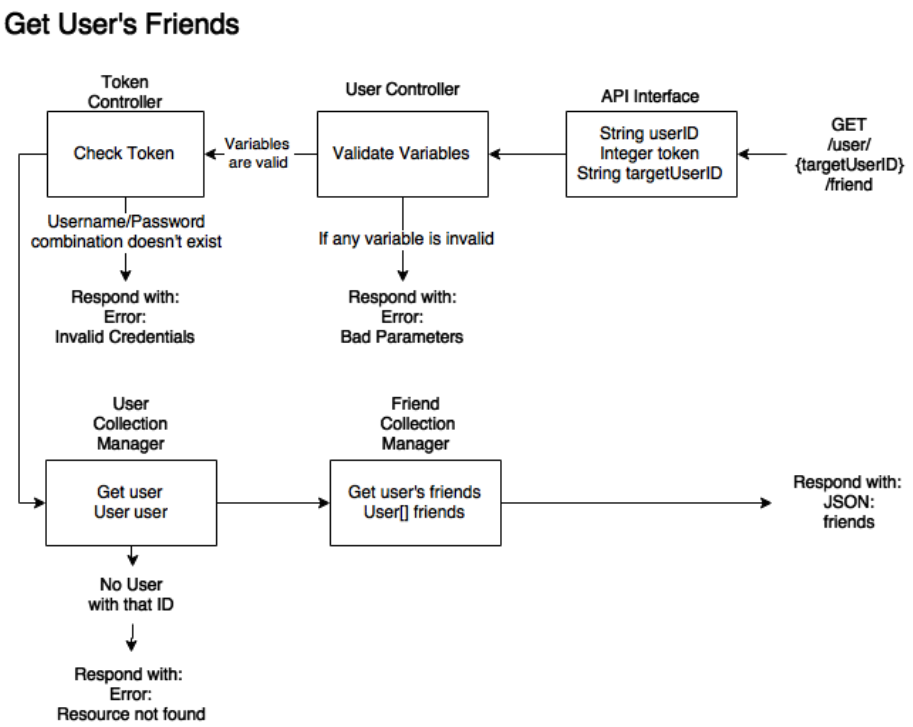

Figure 8: Get user's friends

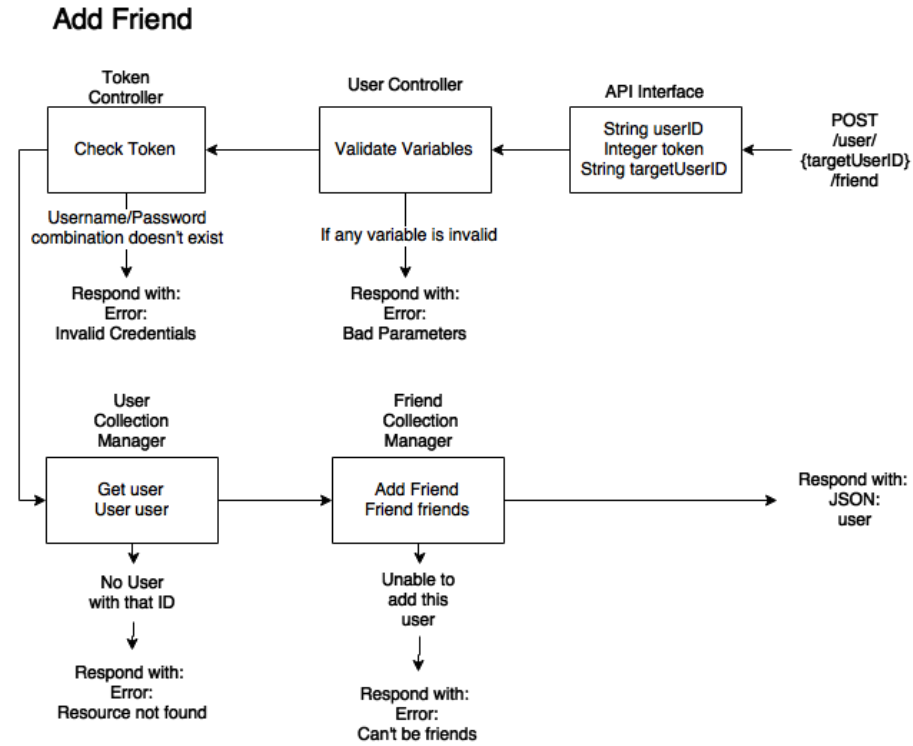

Figure 9: Add friend– server logic

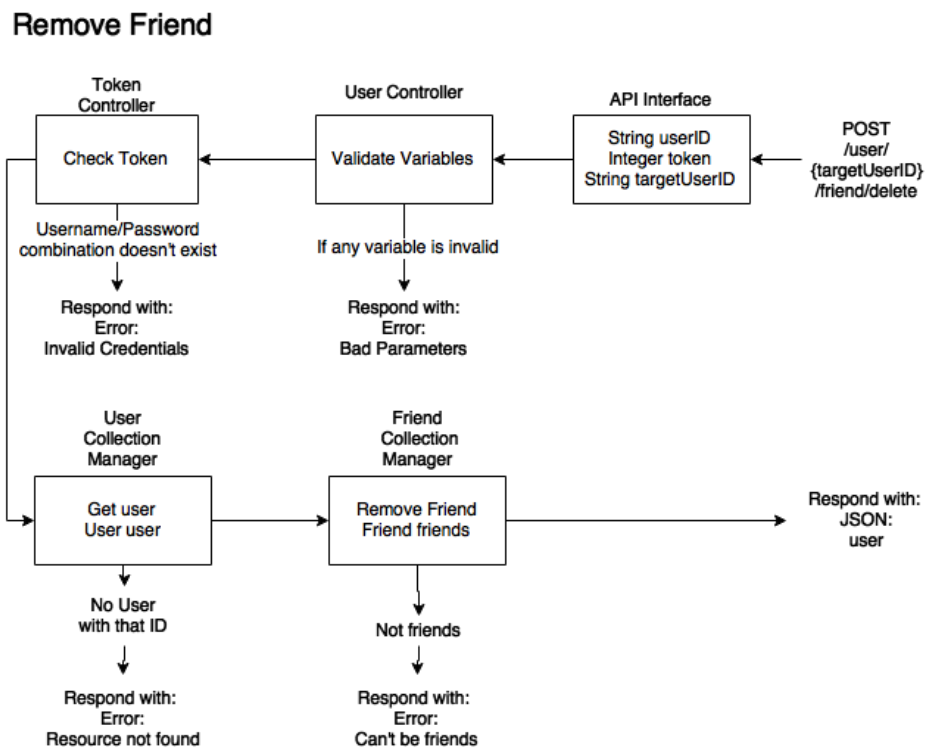

Figure 10: Remove friend– server logic

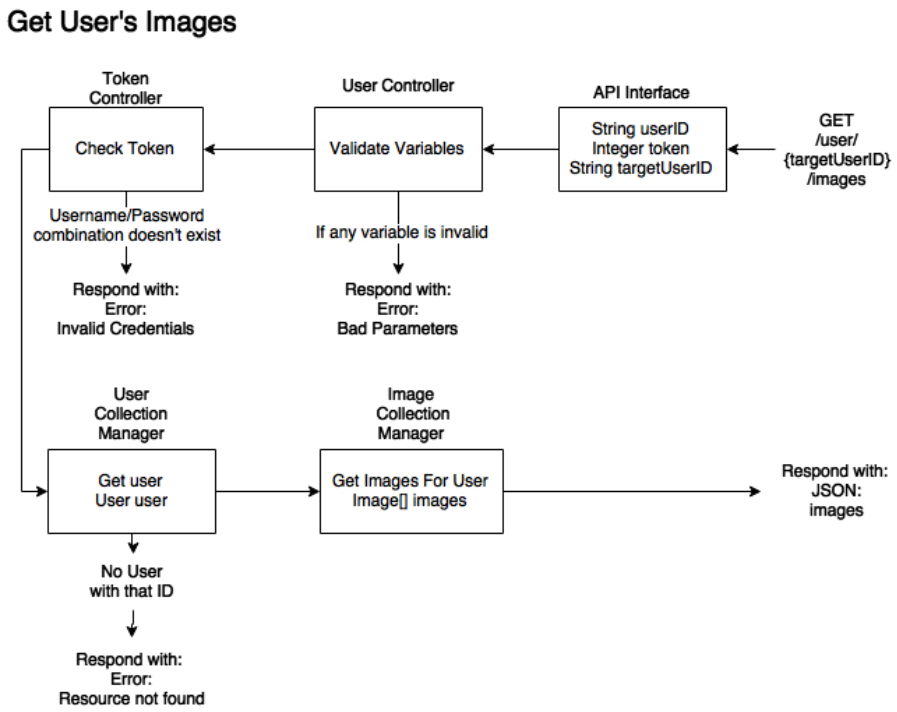

Figure 11: Get User's Images– server logic

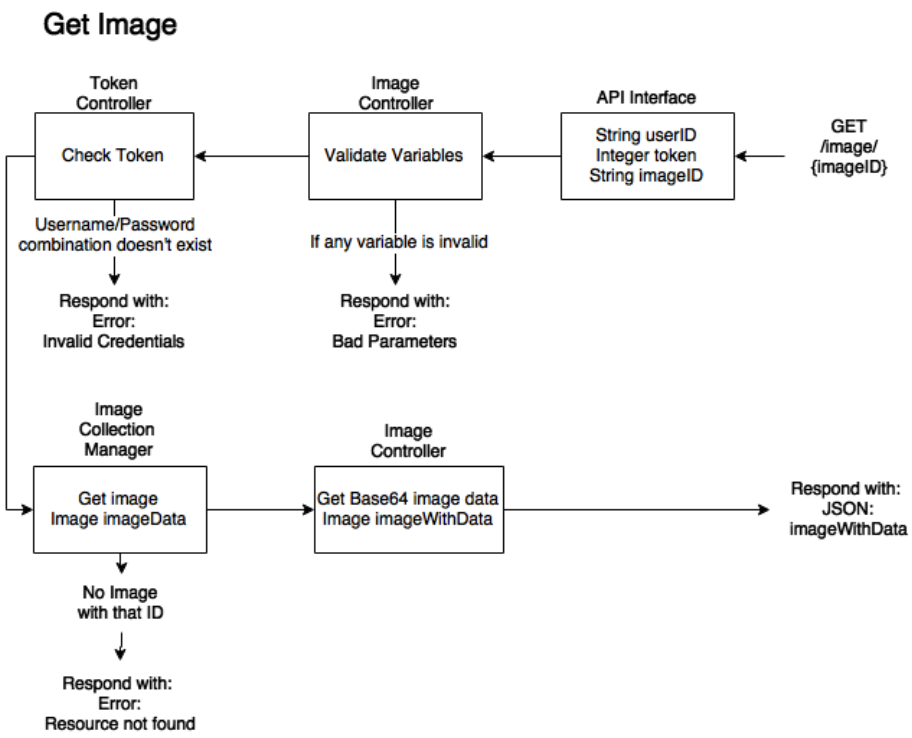

Figure 12: Get Image– server logic

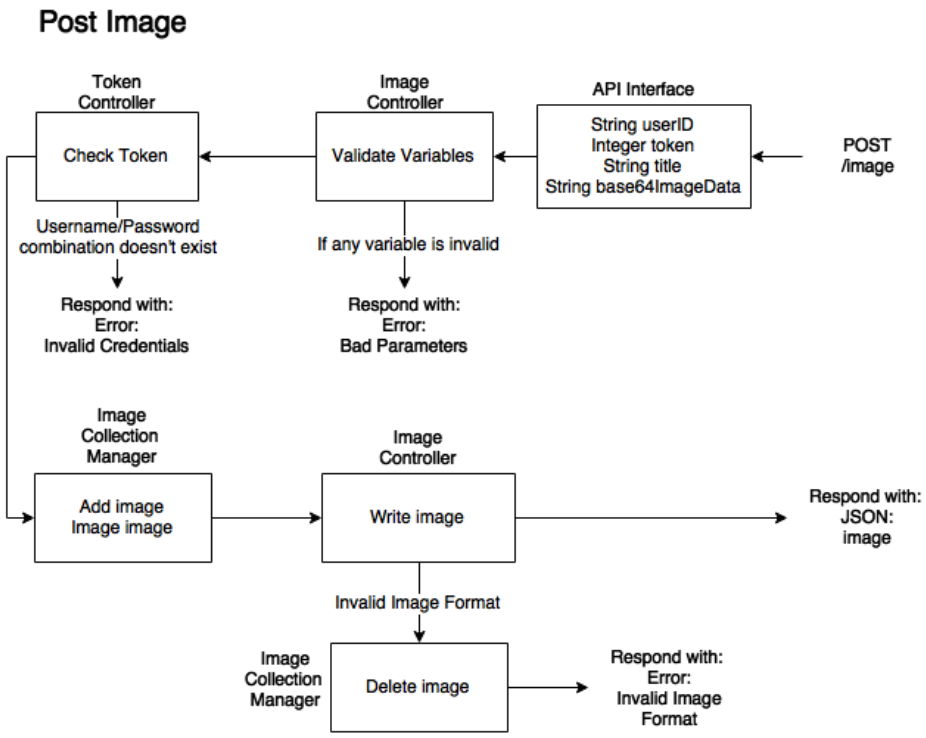

Figure 13: Post Image– server logic

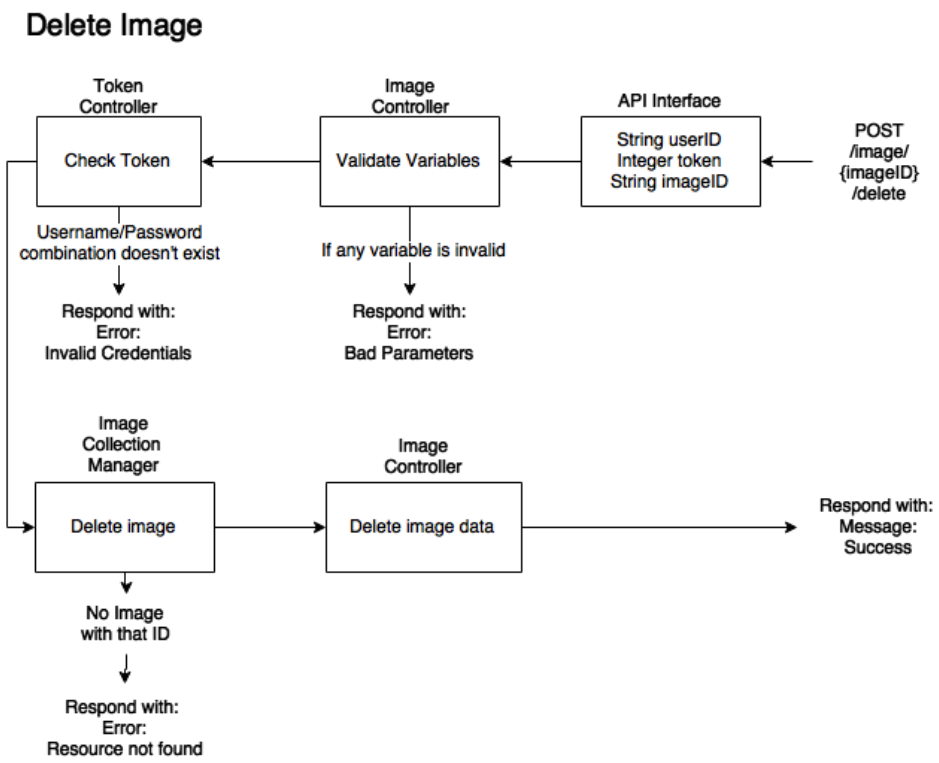

Figure 14: Delete Image– server logic

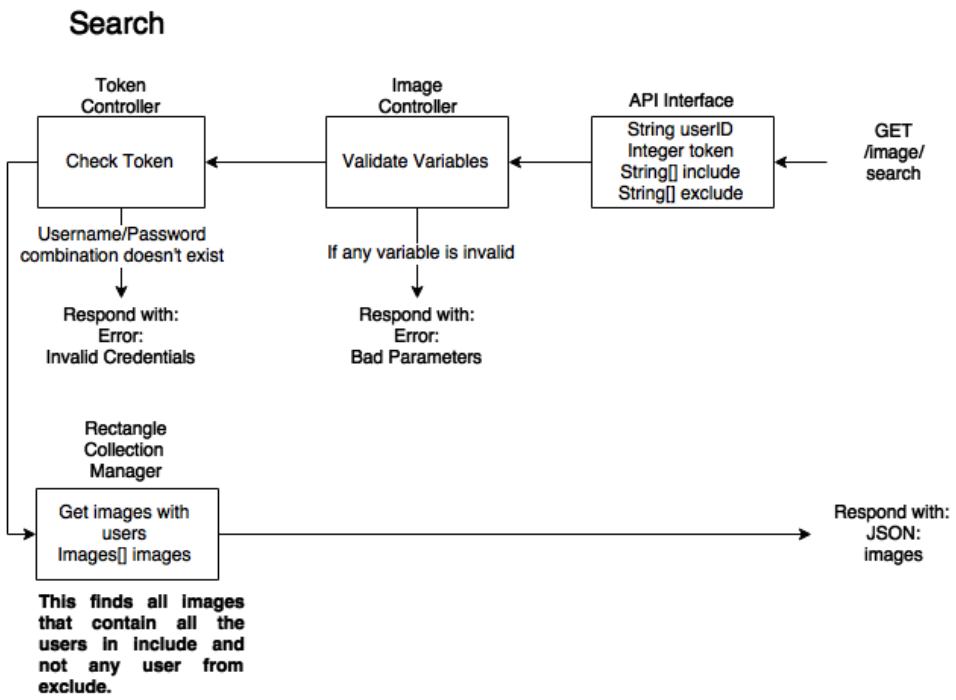

Figure 15: Search– server logic

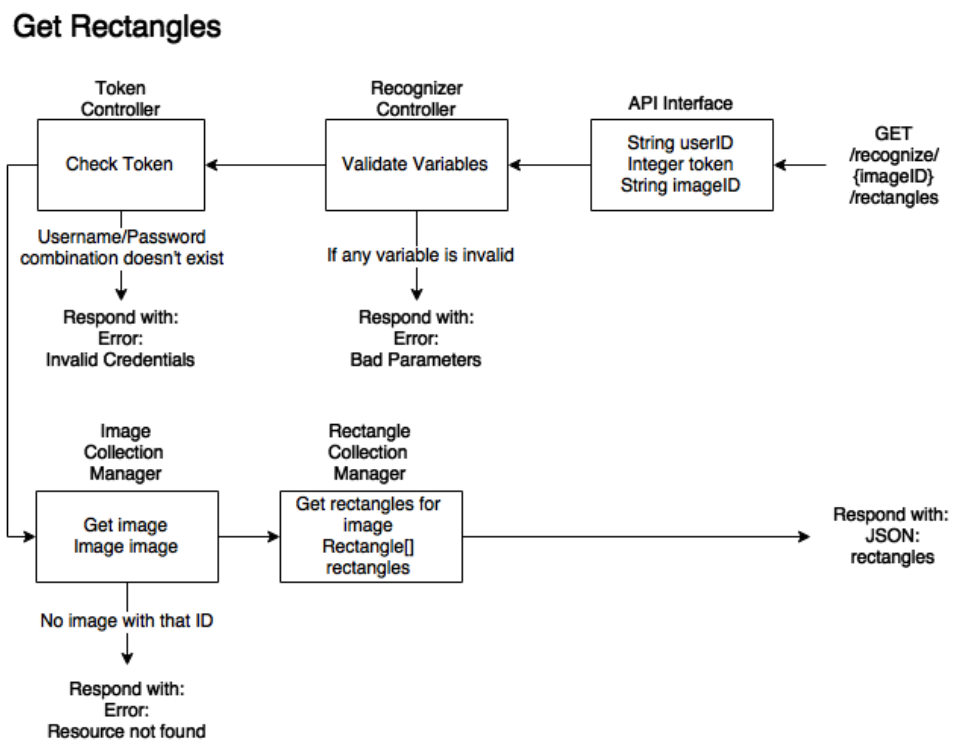

Figure 16: Get Rectangles– server logic

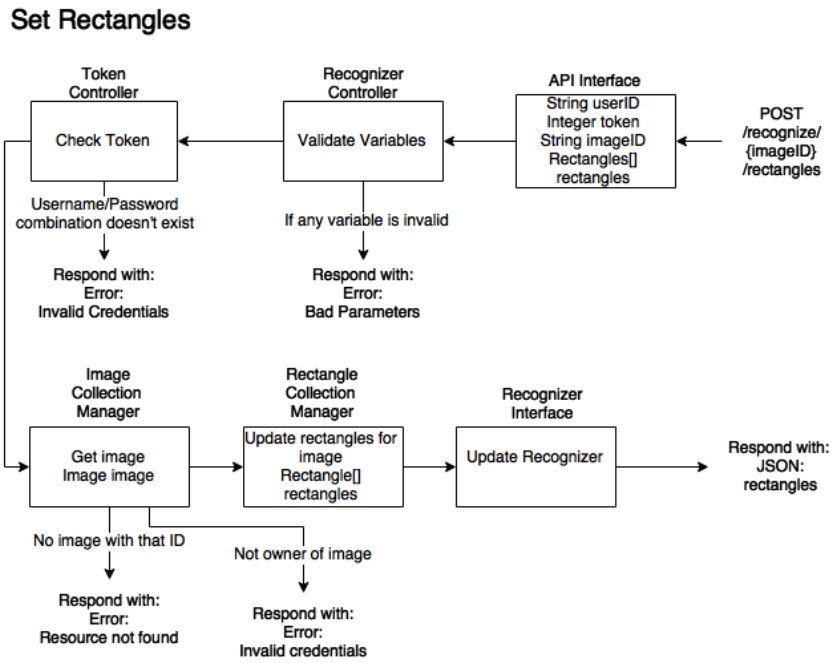

Figure 17: Set Rectangles– server logic

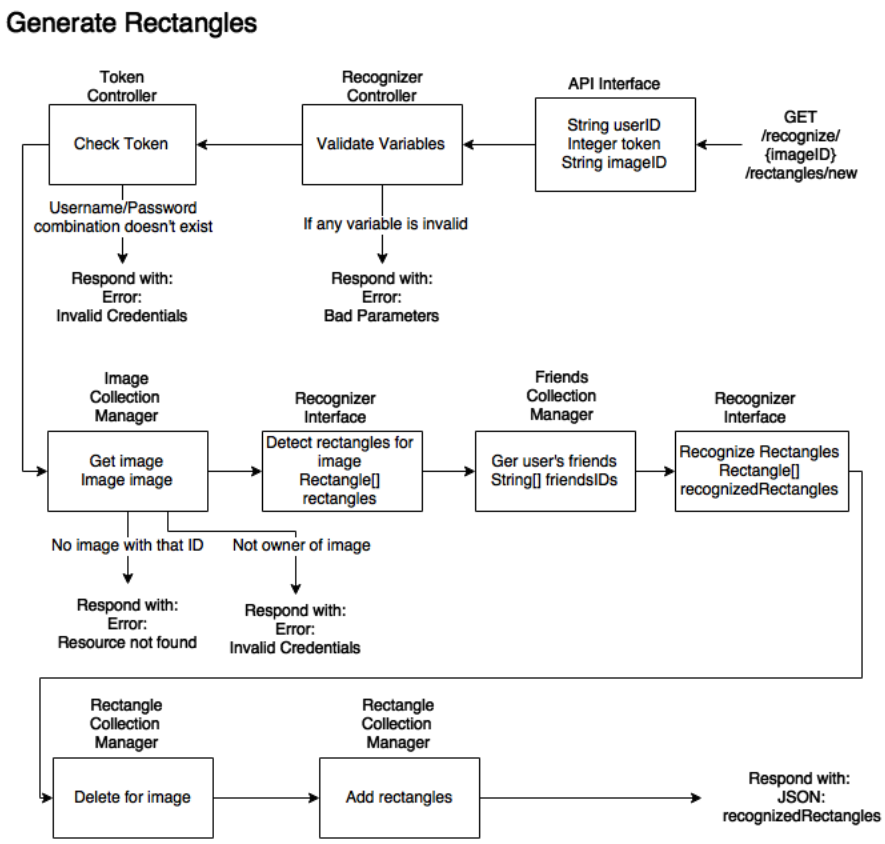

Figure 18: Generate Rectangles– server logic

Android Mobile Application

The Android Mobile Application is an instance of a client application using the API from the server. In this case using Android provides a good platform to both design the application and run it. Android devices usually have cameras accessible to the application and they also suited for both displaying images and performing internet operations due to their powerful programming libraries giving developers a lot of control on how tasks are performed.

API Interaction

The mobile application will interact and control the web service through HTTP requests to the API. As it is a client implementation it will need to be programmed with the API in mind in order to understand and properly control what data it receives. For example it will need to be able to receive and manage a

group of users if, for example, a user requests a list of their friends profiles. As such it will need to have some way to represent the program Entities and also be able to create and send them to the server in a way the server can process.

Additionally the mobile application must be responsible for managing any tokens the user receives to perform various operations on the application.

User Interaction

A user interface provides the user with a way to both visualize the data and to create new data to send to the server. This user interface must be intuitive and not show how any of the interaction takes place, as the user should not be able to see how any of the interactions take place. This means that if a user wants a profile for another user they simply select it from a list and the application will take care of the API behind the scenes and present the user's information.

Design

The mobile app is designed in a way that each view is able to function without knowledge of which view called it. This means that, for example, a view showing a list of users may have been called from the main menu, wanting a list of all the users on the server, or from another user's profile in order to list their friends. This statelessness gives the huge advantage as the API is designed the same way, and while the app must have some form of structure in how it traverses the data (has to log in before uploading images for example) where it can go after accessing that data is not restricted based on how it reached that data. Below are some graphs that show how the views are organized and which functions are called at various points. Also included is a short description of the views and their purpose.

Dotted lines indicate that the view is disposed after this operation is completed as the current view is destroyed. This can happen when the view was created to return data to the view that called it or if being able to return would cause errors (such as being on the main menu after logging out).

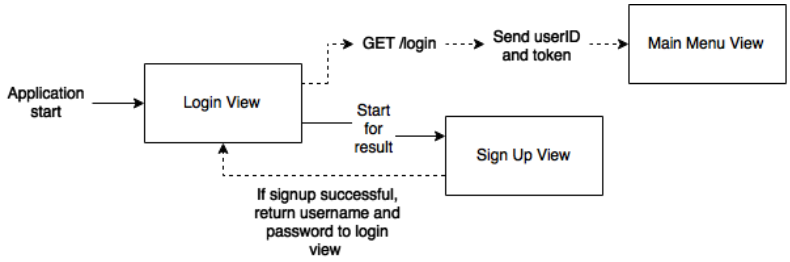

Figure 19: Login View – Mobile Application

The login view contains two text boxes, one for a username and one for a password, and two buttons, one to log in and another to sign up.

Pressing signup will start the Sign Up View, which will, if successful return a username and password of a newly created account which will automatically be put in the text boxes.

Pressing the login view will send a login request to the server using the username and password from the text boxes. If the request is successful then we receive a token entity (password and userID combination). We then start the main menu view, sending the token entity to it, and end this view.

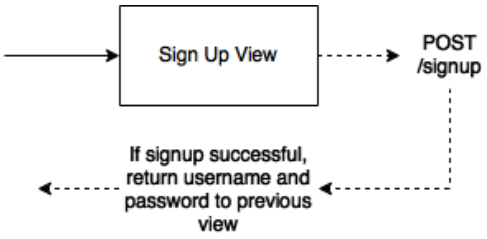

Figure 20: Sign Up View – Mobile Application

The Sign Up view contains 3 text boxes, one for a username, one for a password and one for a name, and a button for signing up.

Pressing the signup button will send a sign up request to the server using the username, password and name from the text boxes. If the request is successful then we return the username and password to the previous view, where it will fill in its username and password text boxes, and end this view.

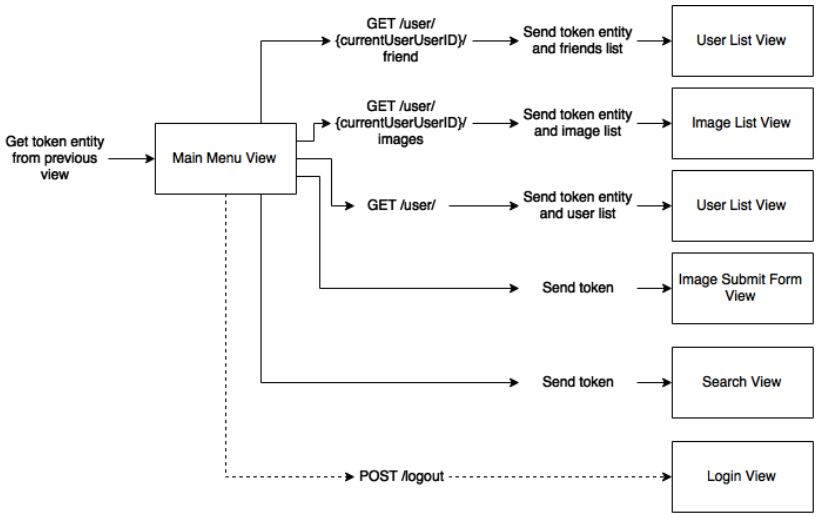

Figure 21: Main Menu View – Mobile Application

The main menu view contains 6 buttons: My friends, My images, All users, Submit image, Face Tag Search and Logout.

Pressing My friends will request from the server a list of the current user's friends and then pass them and the token to a user list view and start it.

Pressing My images will request from the server a list of the current user's images and then pass them and the token to an image list view and start it.

Pressing All users will request from the server a list of all the users on the server and then pass them and the token to a user list view and start it.

Pressing Submit will pass the token to an Image Submit Form View and start it.

Pressing Search will pass the token to a Search and start it.

Pressing Logout will send a request to the server to delete the token, go to a login view and end this view.

Figure 22: User List View – Mobile Application

This view contains a list of all the users passed to it by the previous view.

It displays all the users it received into a list view, showing all the users' name and userID in a cell.

Pressing a cell will request that user's profile and a list of the current user's friends and then open a User View, sending the token entity, the friend list and the received user entity.

Figure 23: Image List View– Mobile Application

This view contains a list of all the images passed to it by the previous view.

It displays all the images it received into a list view, showing all the images' title and imageID in a cell.

Pressing a cell will open an Image Entity View, sending the token entity and the user entity for that image.

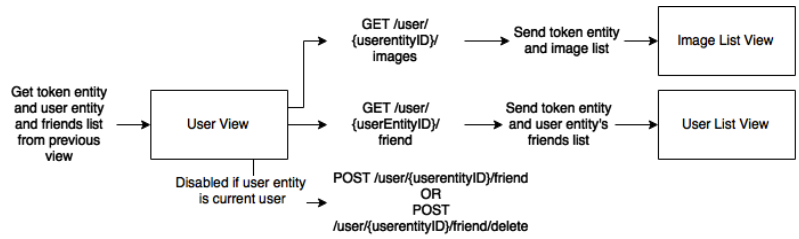

Figure 24: User View– Mobile Application

This view contains 3 buttons: View Images, View Friends and Add/Remove friend.

If View Images is pressed then it will request a list of all of the user entity's images and then pass them and the token to an Image List view and start it.

If View Friends is pressed then it will request a list of all of the user entity's friends and then pass them and the token to a User List view and start it.

If the friends list contains the user entity then the add/remove friend button will display Remove friend. Pressing the button will send a request to the server, and when it responds then the user entity will be removed from the friends list and the button will change to Add friend.

If the friends list does not contain the user entity then the add/remove friend button will display Add friend. Pressing the button will send a request to the server, and when it responds then the user entity will be added from the friends list and the button will change to Remove friend.

If the current user userID (from the Token entity) is the same as the user entity userID then the Add/Remove friend button will be disabled and invisible.

Figure 25: Image Entity View– Mobile Application

The image entity view contains an image view and a menu button, Show Rectangles.

Upon loading the view will download and display the image from the imageID sent by the previous view. The results will be decoded from base64 displayed in the image view.

Pressing the menu option, Show Rectangles, will store the image data in a temporary file, send the file location, the image entity and the token entity a Rectangles View, then start it.

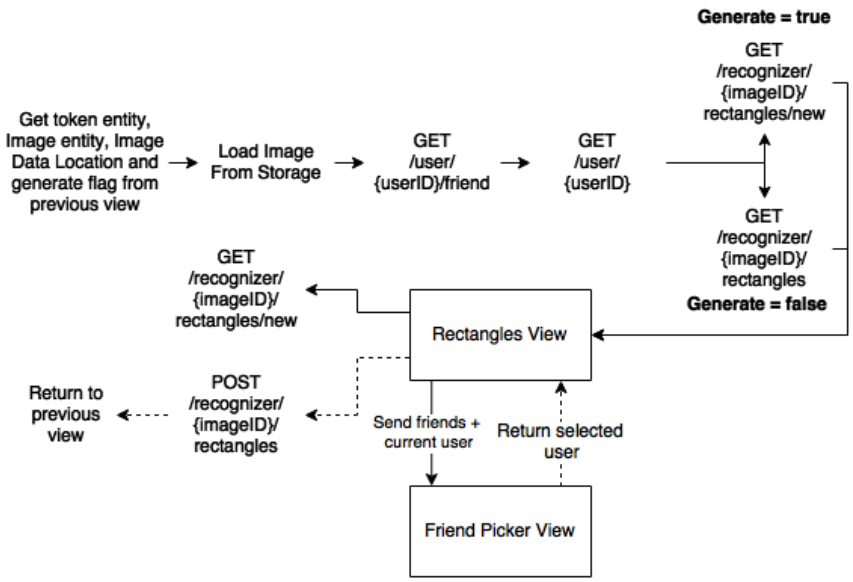

Figure 26: Rectangles View– Mobile Application

Upon starting this view it will, in order, load the image from storage from the Image Data Location and display it, request the current user's friends list from the server, the current user's profile from the server and then, if the generate flag is set it will request the server send a newly generated set of rectangles, and if not or if the flag was never set, send the rectangles for this image from the server. Once the rectangles are received they are overlaid on the image, displaying the name of the user that the rectangle represents if that user is friends with the person being shown.

This view has an Image View, a Submit button and a Generate New Rectangles menu button.

If the current user is the owner of the image then the menu button, Generate New Rectangles and the button Submit are available. The owner may also press inside of the displayed rectangles to Set Rectangle User.

Pressing Generate New Rectangles will send a request to the server to generate new rectangles for the rectangles that have been found. When the server returns the results the image view is refreshed with the new rectangles.

Pressing on the image view within a rectangle will pass the friends list, including the current user and a blank entry, to a Friend Picker View. There the user can select a user or blank. Upon selecting one of those the data will be returned to the current view and whichever rectangle had been selected will have their userID set to the userID of whatever the user selected, with blank resulting in no user having been selected.

Pressing submit will send the current rectangles to the server to update both the recognizer and save in storage. This view will then close and return to the previous view.

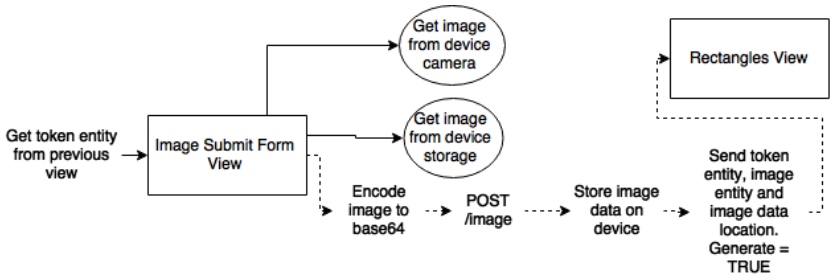

Figure 27: Image Submit Form View– Mobile Application

This view contains an image view, a text box for a title to be entered and 3 buttons: Camera, Device Storage and Submit.

Pressing Camera or Device Storage will call on either the system storage or the device camera to produce an image file. When an image is found the image view is set to display it. It will automatically adjust for any rotation in the image by using the EXIF data for the image.

If the image is set and a title is entered then pressing submit will upload the image to the server and save it to the current user's account. It will then open a rectangles view, providing it all of its required parameters and setting generate to true. This view will then close.

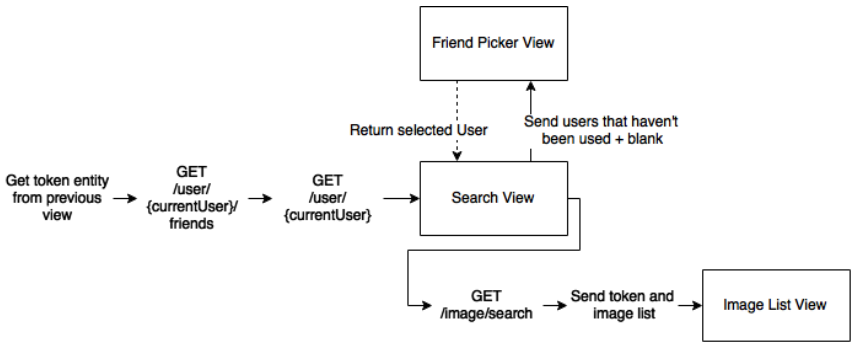

Figure 28: Search View– Mobile Application

This view contains 3 buttons: Search, Add Include and Add Exclude. It also contains 2 list views for users: Include List and Exclude List.

The search feature functions by sending two lists, a list of users to be included in the image and another list of users that are not in the image. You can add users to this list by pressing either Add Include or Add Exclude. Pressing either of them will open a Friend Picker View that will allow you to pick a user that is not yet in either list from a list of the current user's friends and that user. You may also edit a user that is already selected by selecting their name on either list view. Pressing a name will open a Friend picker with all of the people who haven't been selected yet, the user that was selected and a blank option. Selecting the blank option will remove the selected user from the list, and pressing any other will change the selected user to the new user.

Pressing Search will request images from the server using the users in the Include and Exclude lists.

When the server returns the images they will be passed, along with the token, to an Image List View.

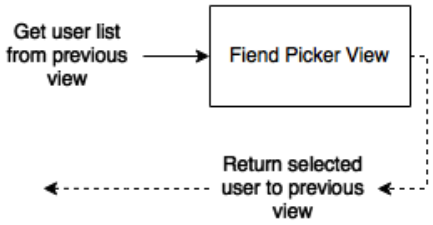

Figure 29: Friend Picker View– Mobile Application

The Friend Picker View contains a list of users. These users are received from the previous view.

Selecting a user from the list will return that user to the previous view and close this view.

Results / Validation

Facial Detection and Recognition System

The first step of this project was to develop the facial detection and recognition systems. The OpenCV library has functions that make implementing this simple, but unfortunately they are not fully implemented in for Java. The C++ version, being the main language for the project contains the functionality that I need, specifically the face recognizer systems, but I was unable to find a way to find a suitable platform to program a server in that language. In order to resolve this issue I used Java Native Interface (JNI). JNI allows Java programs to call upon C++ and other language functionality within Java code by compiling the C++ code into a library and then having the Java program load in the C++ implementation of various functions. This way I was able to create the C++ facial recognizer and detector system and run it as if it was a Java method.

The face detector uses a cascade recognizer to find the location of faces in an image. It first has to load in some pre-trained data to indicate what a face is. Once it has that we can run it on an image so that it can return the location of anything it believes to be a face. We can then send that face to the recognizer to be recognized.

The expected way that an OpenCV face recognizer is supposed to be run is that it is trained to be able to identify all the different people's faces. It is then able to take an image of a face and return what is its best guess to who that person is and a value showing how confident it was in that answer. This did not work with how I intended to use the facial recognizer as I wanted to only receive a user's friends as results, not any one on the network. To solve this issue I created a series of recognizers, one for each user, which will give back a confidence value telling me how closely the person in the image resembles that user. The user that most closely resembles the image is determined to be the correct user and therefore that image is most likely an image of that user.

Training the recognizer is also an issue as the recognized user may not be the correct user, especially in cases where that user has never been seen before in the network. Because of this any recognition that the recognizer performs is first sent to the user for confirmation before being learned by the recognizer. Once confirmed the image is sent back and the recognizer for the identified user is updated with that image. This, unfortunately, can mean that if an image is added incorrectly there is no way of removing it from the recognizer, resulting in less accurate detection in the future.

Searching

In order to search for specific people or specific combination of people a system where we can know who is in a picture is required. To do this the identified people in an image are saved along with references to that image (called rectangles, as they also have the dimensions of the sub image with the user's face), allowing the implementation of a system that searches for images that contain certain people and not others. I implemented this by taking two lists of users, users included in the image and

users not included, and searched for images that contained all of the included users and none of the excluded users. The search would return a list of images that fit the given parameters. This could potentially be useful in a variety of places, such as archiving images of people so that if later you need to find images of a certain person or groups of people you can search for them.

Server

Implementing the recognizer and searching services on a server allows for both remote access to the files and persistent storage of the images and other data. To do this I used a data base to store references to the images and the users that were identified, a location on the file system to store the images and recognizer data and an API as a means of running various queries on the server, such as uploading an image, recognizing people in the image and running searches.

Social Network

Changing the server to a social network allowed for the creation of users and friends, which worked well with the recognizer function. Users are able to set up accounts, add friends, upload images to be recognized and tag their friends in them. The ability to tag their friends in images is a useful indicator of how well people know each other and can provide context in how they know each other. With the modifier system I created being able to tag your friends and not people you don't know works well if the network were to get very large because new people would have difficulty to find their friends in images because they may often find images of people they don't know.

Mobile Application

The mobile application provided a good implementation of a client to the social network. Mobile devices often have internal cameras that can take pictures, so integrating that functionality to the mobile app provides a direct way to get an image to pass to the server. Mobile devices have the ability to create new images of people, unlike a computer or more web based implementation. A mobile device also has

access to any images that a user has previously taken, allowing the mobile app to upload images that the user had previously taken.

It also has the advantage of being able to present data in a visual manner, such as presenting images as a list. By keeping all of the server API calls behind a user interface it simplifies the process to perform actions. An example of this is when uploading an image all the user has to provide is a title and pick an image. Behind the scenes the image is converted to base 64 and the token must be sent along with the image data and title. After that a recognizer call is called, along with multiple calls to get user data to be used in verifying the rectangles, but the user never sees that. This abstraction between the UI and the server interaction provides a much cleaner way of interacting with the server.

Conclusion

In conclusion the facial recognizer and detector systems were able to be implemented into a social network system that is able to differentiate between users and accurately identify people in images. This knowledge of who is in an image could be potentially very useful when trying to determine relationships between users. Users can change these images if they are incorrect, allowing for the images to be corrected and then returned to update the recognizers with new information. This in turn allows the recognizers to function dynamically and not having to train a large number of images before it is able to identify users in an image.

Users are also able to perform advanced searches for images, finding images that have certain users and not others. This uses the results of the recognizer and detector to determine which users are in the images.

Future Work

Here is a list of potential improvements that can be made to the system.

- Improve the facial detector to provide multiple guesses for which user is in which image to allow for the recognizer to better determine who is in an image while the recognizer is still training and is not entirely confident in who it is identifying.
- Improve the social network to allow for more social features such as commenting and including a timeline to show what other users are doing.
- Investigate how the metric of users in the same image and the frequency of who is in a user's images can be used to represent relationships.
- Improve the security and functionality of the mobile app as well as write a Terms of Service and privacy policy to prepare it for a release onto the Google Play Store or equivalent.
- Add more image features, such as being able to manually tag users that the detector may not have picked up and creating galleries of images.
- Integrate sentiment analysis to attempt to determine the context of when the picture is taken as a means to specify what the relationships between users are.

References

[1] Senior, A., & Bolle, R. (2002). FACE RECOGNITION AND ITS APPLICATIONS. In *The Kluwer international series in engineering and computer science* (Vol. 2SECS 697, pp. 101-115). Boston, Massachusetts: Kluwer Academic.

[2] MCHUGH, M. (2015, June 24). Facebook Can Recognize You Even if You Don't Show Your Face. Retrieved August 21, 2015.

[3] Rodriguez, A. (2015, February 9). RESTful Web services: The basics. Retrieved August 21, 2015, from http://www.ibm.com/developerworks/library/ws-restful/

[4] Introduction to MongoDB. (n.d.). Retrieved August 21, 2015, from http://docs.mongodb.org/manual/core/introduction/

[5] Jersey. (n.d.). Retrieved August 21, 2015, from https://jersey.java.net/

[6] About OpenCV. (n.d.). Retrieved August 21, 2015, from http://opencv.org/about.html

[7] Spring for Android. (2015). Retrieved August 21, 2015, from http://projects.spring.io/spring-android/

[8] Jackson JSON Processor. (n.d.). Retrieved August 21, 2015, from http://wiki.fasterxml.com/JacksonHome