

# **(Dua-Khety: Hieroglyphics Detection and Classification)**

Software Design Document

Name (s): Malak Sadek, Mohamed Badreldin, Ahmed El Agha,  
Mohamed Ghoneim

Date: (15/05/2018)

<b>TABLE OF CONTENTS</b>	
<b>1.</b>	<b>INTRODUCTION</b>
1.1	Purpose
1.2	Scope
1.3	Overview
1.4	Reference Material
1.5	Definitions and Acronyms
<b>2.</b>	<b>SYSTEM OVERVIEW</b>
<b>3.</b>	<b>SYSTEM ARCHITECTURE</b>
3.1	Architectural Design
3.2	Decomposition Description
3.3	Design Rationale
<b>4.</b>	<b>DATA DESIGN</b>
4.1	Data Description
4.2	Data Dictionary
<b>5.</b>	<b>HUMAN INTERFACE DESIGN</b>
5.1	Overview of User Interface
5.2	Screen Images, Objects and Actions
<b>6.</b>	<b>Appendix A - Sequence Diagrams</b>
<b>7.</b>	<b>Appendix B – Results</b>

# **1. INTRODUCTION**

## **1.1 Purpose**

This software design document describes the architecture and system design of the Dua-Khety system for hieroglyphics detection and classification. It is intended for the developers that will be working on the mobile application, it describes the architecture of the system, the main classes, their subclasses, and all the attributes and functions associated with each one, as well as the relationships between them. It also details the user interfaces, and how they relate to each other.

## **1.2 Scope**

The system is meant to offer archaeologists and tourists an insight into the Gardiner codes and transliterations of hieroglyphics they might encounter while exploring ancient Egyptian monuments or coming face to face with ancient Egyptian artifacts. The system will allow users to take pictures of hieroglyphics and mark the region in the photograph where they would like the decoding to take place, they will then be presented with the results. The system also offers social interaction features in the form of a news feed for registered users, as well as the ability to submit content to the developers to improve future classifications. The service is offered for free but registering for an account will unlock the social network features as well as the ability to send the developers content. It will be connected to an online database of Gardiner codes, a neural network for classification, and an online database for social network features.

## **1.3 Overview**

This document will give a high-level system overview of the main classes and their relationships. It will then break down these main classes into subclasses and provide the attributes and functions of each one with descriptions and relationships between them. Finally, the user interface is explored.

## 2. SYSTEM OVERVIEW

The classes of the system are:

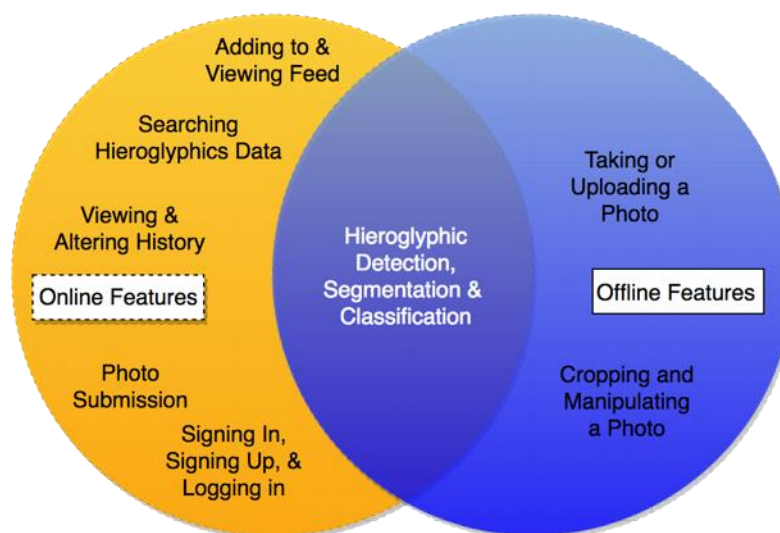
(Format: Main class -> sub-class)

The system can be split into easily distinguished online and offline features with the main feature (ultimate classification) being present in both modes of operation. If the application detects an active internet connection, then the user can initiate any of the online features, or else they will only be able to engage in the sole feature of obtaining a classification.

Online features include:

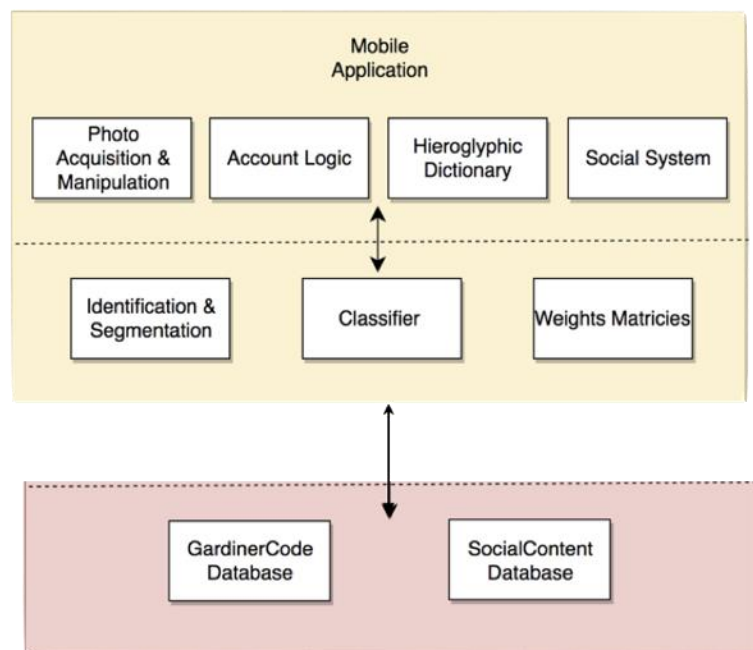
1. Having the results of images added to the feed that all online registered users are presented with
2. Being able to view said feed
3. Viewing a history of all previous classifications belonging to the user
4. Deleting some or all of said history
5. Submitting a photo with its Gardiner's code and optionally its description to aid better future classification
6. Signing up for an account, signing in to an existing account or logging out of a signed in account
7. Being able to search for information about a hieroglyph using its Gardiner code

If there is no internet connection, then users would only be able to take or upload a photo, crop/rotate/flip it as necessary, and obtain a classification for it without extra information.



Another main distinction that affects which features a user is able to access is whether they have created an account or signed into one, or whether they opted to use the application directly as a “one-time user”. One-time users would be able to only use the segmentation & classification systems (whether online or offline) and to search for Hieroglyphic data. Only registered users who are currently logged in would be able to access the rest of the features (viewing their history, viewing their feed, and photo submission).

The system is mainly split into three layers based on the number of steps needed to access a function. The top layer would include all features that are generally accessible in a few steps, grouped together under one of four umbrellas; photo acquisition & manipulation, account logic, the dictionary, or the social system. The second layer represents functions that are inaccessible to the user and that happen in the background to provide them with the intended output, mainly the segmentation and classification logic. This layer was to be duplicated both on the mobile application (primitive functions) and on the server (more advanced functions). However, there is no need to include it at the server level as it was fully integrated into the mobile application, supporting fully offline usage. And the final layer includes components that are not physically placed on the mobile device all together, namely the databases holding data used by the application.



### Photo Acquisition & Manipulation

Functions that fall under this category are all functions that are available offline and that handle the image. They will include opening the camera, taking a picture, opening the gallery, choosing a picture, cropping a picture, saving a picture, uploading a picture along with the user’s information to the SocialContent Database, downloading a picture from the SocialContentDatabase, and sending a picture to the classifier. It also includes displaying more in-depth information about the detected hieroglyph once a classification has been obtained.

## Account Logic

This accounts for any function that handles users' accounts and would include creating an account, storing the data on the device's shared preferences, interfacing to Firebase and storing the account information there, emailing the user with a verification email, checking that the account has been verified, signing the user in, logging the user out, removing the data from the device's shared preferences, and validating the user's information.

## Social System

These functions would handle both the Feed and the History features. They would consist of opening the feed, obtaining all posts from the SocialContent database, searching for posts that are under a certain owner, searching for posts that are for a certain identification code, opening the history, obtaining all posts belonging to the current user, deleting a post belonging to the current user, and deleting the user's entire history.

## Hieroglyphic Dictionary

This feature allows users to search for information about any hieroglyph using its Gardiner's code. It interfaces with the GardinerCode database and is able to provide information about the hieroglyphic's image, description, transliteration, and any useful links when provided with a Gardiner code. This information was collected from [egyptianhieroglyphics.net](http://egyptianhieroglyphics.net).

## Identification & Segmentation

This is accomplished using OpenCV by converting the bitmap taken by the user into a Mat object, turning it to black and white using `cvtColor` and `Imgproc.COLOR_BGR2GRAY` and then blurring the image using `blur` with a radius of 3. The average is then computed using `mean`. Afterwards, the image is thresholded using `threshold` and `Imgproc.THRESH_BINARY_INV` with a min-val of the computed average and a maxval of 255. Canny is then applied using `Canny` with thresholds of the average \* 0.66 and the average \* 1.33 and an aperture size of 3 which was found to be optimal. Afterwards, the components are extracted using `connectedComponentsWithStats` and are used to draw bounding boxes around individual hieroglyphics on the original image. These are then cropped around and placed in an array of smaller images. The cropped images (from the original image) are then turned to black and white and thresholded in the same manner before being fed to the classifier to maximize accuracies.

## Classifier

The initial classifier used consists of a 32 matrix convolutional layer with a kernel size of 3x3 and Relu as an activation function followed by an identical 64 matrix convolutional

layer. This is followed by max pooling with a pool size of 2x2 and a drop out of 0.25. The model is then flattened and this is followed by a dense layer 40 with a Relu activation function that is also flattened. Finally, a dense layer 22 representing the 22 classes that sir Gardiner has used to categorize hieroglyphics is used (will be replaced by 162 which is the number of classifiable hieroglyphs). This layer's activation function is Soft Max and it uses an RMP Prop optimizer. The resulting .h5 file is optimized and frozen to be converted into a .pb file to be used within the Tensorflow framework on Android devices, and is also fed into the CoreML conversion tools to fit within the CoreML framework on iOS devices. The classifier accepts images of dimensions 150x150 (any incoming images are resized to this).

Initially, classifier results were poor, however they swiftly improved after binarizing the taken images before feeding them to the classifier. While the following results were numerically satisfactory for any machine learning problem (around 66% accuracy), it was found that the trained model was overfitting on the both training and testing data. After looking at the root of the problem, which was the data set, it was found that the obstacle was in the fact that the dataset was extracted from one place which is the Pyramid of Unas at one time from one person using one device and was captured from a book, so the testing data was not as the aim of the system is to classify hieroglyphs in different temples and on different stones. From then on, hieroglyphic scriptures from the Louvre and the British Museum and the Museum of Alexandria were obtained as new testing data, as they proved more dispersed. In addition to this, the previous dataset was also augmented and cleaned to remove misleading symbols. Augmentation uses a shear range of 0.2, zoom range of 0.2, and rotation range of 0.2. With some research, it was found that the Siamese network would be more suitable to the problem at hand.

Such a network differs from normal ones by taking as an input pairs of images and a label representing whether they are from the same class or not (displayed as a 0 or 1). In other words, half of what is fed to the network are pairs of images of the same classes with their label as 1, and the other half, two different images from two different classes, with their label as 0. The images are chosen randomly.

The model consists of four separable convolutional layers with varying kernel size and 'relu' as an activation function with Max Pooling between each one. The model is flattened at the end to get the features at that stage. For training, two of these exact layers are merged and another dense layer is added at the end with one output to represent whether the pairs of images are from the same class or not, that uses 'sigmoid' as an activation function.

Below is a sample of the model code:

```

Input_shape = (75, 50, 1)
Left_input = Input(input_shape)
Right_input = Input(input_shape)
Convnet = Sequential()

Convnet.add(SeperatbleConv2D(32, (5, 5), activation='relu',
input_shape=input_shape))
Convnet.add(MaxPooling2D())

Convnet.add(SeperableConv2D(48, (4, 4), activation='relu'))
Convnet.add(MaxPooling2D())

Convnet.add(SeperableConv2D(64, (3, 3), activation='relu'))
Convnet.add(MaxPooling2D())

Convnet.add(SeperableConv2D(64, (3, 3), activation='relu'))
Convnet.add(Flatten())

Encoded_l = Convnet(left_input)
Encoded_r = Convnet(right_input)
L1_distance = lambda x: K.abs(x[0]-x[1])
Both = merge([encoded_l, encoded_r], mode = L1_distance,
output_shape = lambda x: x[0])

Prediction = Dense(1, activation='sigmoid',
bias_initializer=b_init)(both)

Siamese_net =
Model(input=[left_input, right_input], output=prediction)
Siamese_net.compile(loss='binary_crossentropy',
optimizer='adamax')

```


Recently, this Siamese concept has been proven to be helpful in problems using a large number of classes with a small number of images for each class, like the Humpback Whale Identification Challenge in Kaggle. As for classifying the actual test image, the training images are first fed through the mentioned network and a feature vector of 640 values is extracted. Afterwards, the average of all the vectors of the images in a class are computed and stored it in a Comma Separated Values (CSV) file. This occurs for all the classes (157). Coming down to predicting the class number for a new test image, it is fed to the same network and a feature vector of 640 values is extracted for it as well. The L1 distance between its feature vector and the previously extracted ones from each class is



calculated, and the smallest five distances are taken as the top five predictions.


### SocialContent Database

This database will be used for holding the content of the Feed and History features, it will consist of an ID acting as a primary key with the auto-increment setting activated. An email, name, Gardiner code, and the photo taken. Upon classification, if the user is registered and has internet access, their image and its classification will be added to the database and thus displayed in all users's feeds from that point forward. It will also be available in that user's history. They can remove it from the database (and thus from the feed and history) by deleting it from their history within the application. The feed will also allow users to search for content within that database based on either the poster's name (Owner) or the identification code of the image.

#	Name	Type
1	ID 	int(11)
2	Email	text
3	Owner	text
4	Photo	longblob
5	Code	text

### GardinerCode Database

This database will be used for holding all the classified Gardiner codes, a sample image of the hieroglyph they represent, the visual description of the hieroglyph, its transliteration, and if applicable, a Wikipedia link that provides more information about it. This is for the purposes of the dictionary feature, as well as providing information after a classification has taken place if the mobile device has an internet connection at the time. The user will be unable to access the database or alter it directly through the application. The Gardiner's code will be the database's primary key.

#	Name	Type
1	Code 	text
2	Image	longblob
3	Description	text
4	Meaning	text
5	Link	longtext

### External Interfaces

*CoreML* - For integrating the classifier with the iOS mobile application.

*CroppableImageViews* – To allow the user to crop images on iOS.

*EgyptianHieroglyphics.net* – For descriptions and transliterations of all hieroglyphs included in the dictionary feature.

*Firebase* – For account management (signing up, in, or out, and verification email logic).

*Gson* – For retrieving data from the databases in JSONArrays.

*Image Cropper* – For allowing images to be cropped, flipped, or rotated on Android.

*OpenCV* – For preprocessing and segmentation features.

*Picasso* – For more lighter and more efficient photo handling.

*Reachability* – In order to discover whether the iOS device is connected to the internet either through cellular data or WiFi to be able to warn the user that online features are not available.

*TableViewCache* – For caching obtained images from either the history or feed in order to speed up viewing them in the future and thus improve user experience.

*Tensorflow* – For integrating the classifier with the Android mobile application.

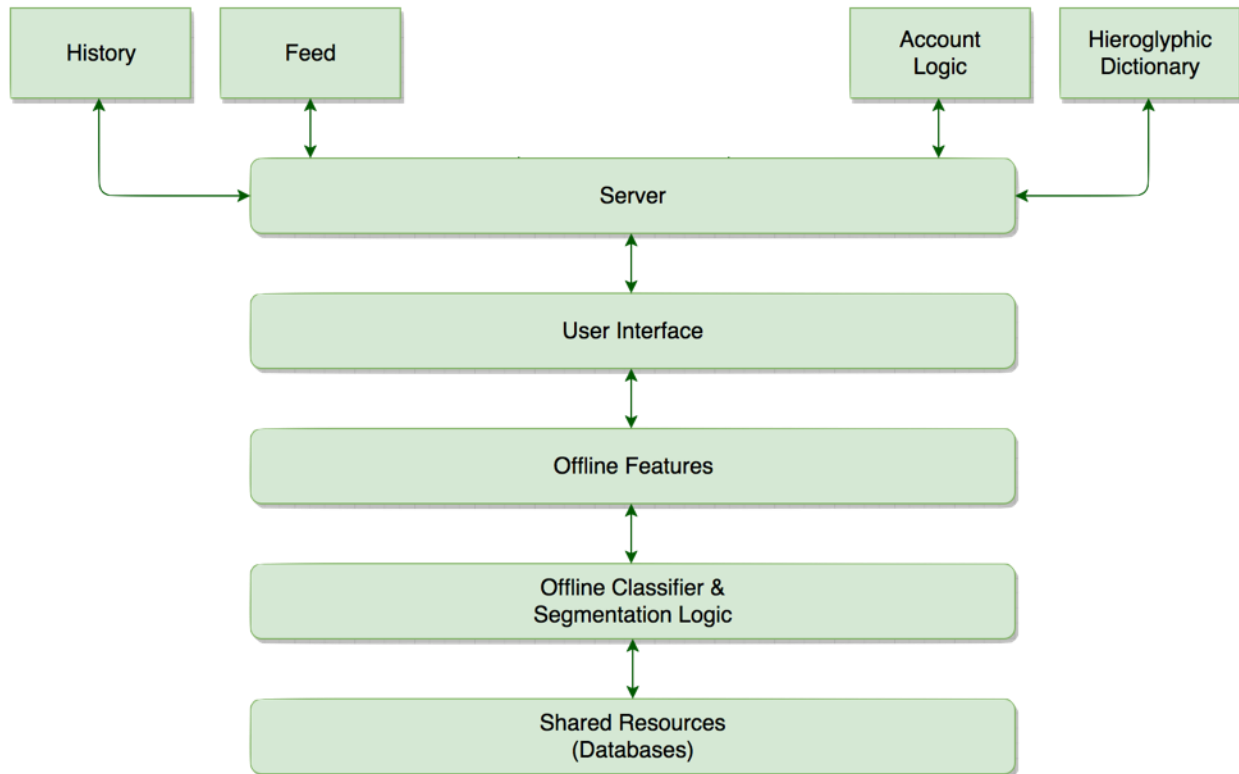
*Volley & AsyncHTTP* – For making requests to run scripts on the remote server.

*Wikipedia.com* – For providing users with more information about certain Egyptian themes.

### 3. SYSTEM ARCHITECTURE

#### 3.1 Architectural Design

The architectural design of the system will be a hybrid between **layered** and **client-server** architectures. Within the layered segment, the bottom-most layer would be all the shared resources, such as the database containing corresponding Gardiner codes and images and the database containing content posted within the social system. On top of that would operate the logic for the onboard classification and segmentation that would go on the mobile device itself for offline usage and above that would be the offline capabilities for photo acquisition and manipulation. Lastly would be the user interface which would also be able to link to the several online features available on the system's server (with the client being the mobile device and the user using it). Classification and segmentation will no longer be performed on the server as they have been fully implemented on the device itself so as to support full offline usage.



Excluding all classification and segmentation logic (these will be discussed separately), the classes included within the **Android** mobile application will be the following:

- *AnalyzingActivity*
- *Classifier*
- *CreditsDialogFragment*
- *CustomAdapter*
- *DevelopersDialogFragment*
- *FeedActivity*
- *HistoryActivity*
- *ImageSourceDialogFragment*
- *Item*
- *LogoutDialogFragment*
- *MainActivity (SignUpActivity)*
- *OneTimeActivity*
- *OpeningActivity*
- *ResultsActivity*
- *SearchActivity*
- *SignInActivity*
- *SubmitActivity*

- *SuccessActivity*
- *TensorFlowImageClassifier*
- *VerificationActivity*

The classes included within the **iOS** mobile application will be the following:

- *AnalyzingViewController*
- *CameraViewController*
- *CroppingViewController*
- *FeedViewController*
- *HistoryViewController*
- *OneTimeViewController*
- *OpeningViewController*
- *Reachability*
- *ResultsViewController*
- *SearchViewController*
- *SignInViewController*
- *SignUpViewController*
- *SubmitViewController*
- *SuccessViewController*
- *TableViewCell*
- *TableViewCell2*
- *VerificationViewController*

For the client-server section of the architecture, several PHP scripts will be used to interface between the application and the databases, as well as other features. They will be the following:

**- AddUser**

This is used after the user successfully receives a classification to add the content to the SocialContent database.

**- RemoveUser**

This is used when the user deletes an entry in their search history, this deletes the entry from the database, preventing it from showing up in users' feeds in the future and the user's history.

**- FindHistory**

This is used when the History page is opened in the application, it obtains all entries in the database who's owner is the current logged in user.

**- ClearHistory**

This is used when the user chooses to delete their entire history and thus remove all entries that belong to them from the database.

### - ShowFeed

This is used when the Feed page is opened in the application, it obtains all entries in the database.

### - SearchName

This is used when the user searches in the feed page, it obtains all entries in the database belonging to the entered name.

### - SearchCode

This is used when the user searches in the feed page, it obtains all entries in the database containing the entered Gardiner's code.

### -SearchGCode

This is used when the user is presented with a classification (if they are connected to the internet) to provide more information about the hieroglyphics detected using data from the GardinerCode database, and is also used when the user searches for a hieroglyph from the dictionary feature.

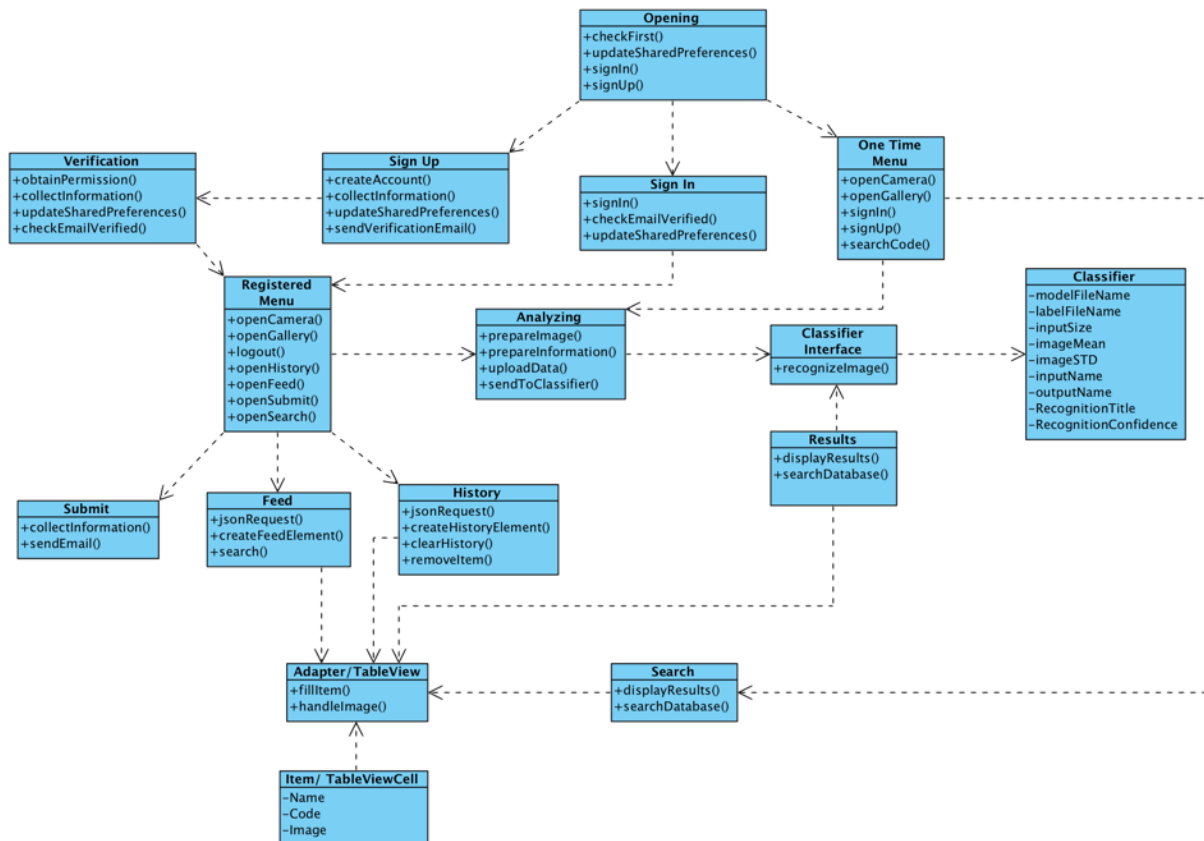


Figure 1. Utility Class Diagram

## 3.2 Decomposition Description

### 3.2.1 Android Application

Below is a detailed description of each class's attributes and functions. Here, no distinction is made between main classes and sub-classes. In the attributes section, the attribute name is in bold, and their data type is in italic. In the functions section, the function name is in bold, and arguments are in bold and italic.

It should be noted that all classes have an `onCreateOptionsMenu` and an `onOptionsItemSelected` function to be allow the user to access two dialogs from any screen; one with information about the application developers, and one with information about the sources of data for the application.

It should also be noted that all activities extend the `AppCompatActivity` class.

#### **Class** *AnalyzingActivity* **Name**

<b>Attributes</b>	<b>prgrs</b> <i>ProgressBar</i> <b>tick</b> <i>ImageView</i> <b>image</b> <i>ImageView</i> <b>u</b> <i>Uri</i> <b>RequestParams</b> <i>RequestParams</i> <b>onetime</b> <i>Boolean</i> <b>cimage</b> <i>CropImageView</i> <b>crop</b> <i>Button</i> <b>encodedString</b> <i>String[ ]</i> <b>classifierResult</b> <i>String[ ]</i> <b>img_C</b> <i>Mat</i> <b>edge_detected</b> <i>Mat</i> <b>img_BW</b> <i>Mat</i> <b>blurred</b> <i>Mat</i> <b>labels</b> <i>Mat</i> <b>stats</b> <i>Mat</i> <b>centres</b> <i>Mat</i> <b>clean</b> <i>Mat</i> <b>uploadimage</b> <i>String</i> <b>bmp</b> <i>Bitmap</i>
<b>Functions</b>	<b>void Setup();</b> Prepares all visual elements by linking between them and their counterpart variables within the code and sets the relevant items to invisible (until analyzing is complete). It also obtains the image the user

has taken/selected and cropped/manipulated from the previous activity and displays it. It also initializes the classifier. Finally, it determines whether the activity was called by the actions of a user that has logged in or a one time user (through the onetime variable) to know whether to upload the data to their history and the feed or not.

**void PrepareImage();**

Converts the image the user has taken/selected and cropped/manipulated into a bitmap, compresses it, converts it to a byte array, and finally encodes it using base64. It creates a bitmap of the image and feed it to the classifier, obtaining a classification result.

**void prepareInformation();**

It obtains the necessary information from the mobile device's shared preferences, and packages them into the RequestParams variable to prepare them for an AsyncHTTP request.

**void uploadData();**

Performs an AsyncHTTP request using the PHP script AddUser. Only performed if the user is not a onetime user.

**protected void onCreate(Bundle)**

Calls the Setup, prepareInformation, and uploadData functions.

**void onManagerConnected(int)**

Loads OpenCV asynchronously and initializes the relevant variables.

**protected void onResume()**

Called after OpenCV is loaded (acts as an asynchronous onCreate function) and then checks whether the activity has been called by the user wanting to analyze a new image or by the user clicking on an existing image in their history or feed and either analyzes the existing image or creates an ImageSourceDialogFragment to ask the user about the source of the new image (either camera or gallery).

**void onActivityResult(int, int, Intent)**

Called after the user selects an image from the gallery or takes an image from the camera, it extracts the image using the appropriate method and then sends it to the cropping function.

**void crop()**

Crops the selected or taken image by the user by utilizing the documented API, then sends the image to get segmented.

**Bitmap segment(Bitmap)**

Uses OpenCV to segment individual hieroglyphics from a given bitmap and returns another bitmap with bounding boxes as well as 2 arrays, one with images of each individual segmented hieroglyph (for the user) and another one with the images binarized (for the classifier).

**Class CreditsDialogFragment**  
**Name**

<b>Attributes</b>	<b>mContext</b> <i>Context</i>
<b>Functions</b>	<b>public Dialog onCreateDialog(<i>Bundle</i>)</b> Creates a dialog displaying the sources of data for the application and allows the user to dismiss it.

**Class Classifier**  
**Name**

<b>Attributes</b>	<b>id</b> <i>String</i> <b>title</b> <i>String</i> <b>confidence</b> <i>float</i> <b>location</b> <i>RectF</i>
<b>Functions</b>	<b>public Recognition()</b> Initializes all variables  <b>public recognizeImage(Bitmap bitmap)</b> Obtains the image's classification (further explained in the interface class TensorFlowImageClassifier).



**Class** *CustomAdapter*  
**Name**

<b>Attributes</b>	<b>N</b> <i>TextView</i> <b>C</b> <i>TextView</i> <b>I</b> <i>ImageView</i> <b>L</b> <i>LinearLayout</i> <b>c</b> <i>Item</i> <b>params</b> <i>ViewGroup.LayoutParams</i>
<b>Functions</b>	<b>void Setup(View);</b> Prepares all visual elements by linking between them and their counterpart variables within the code.  <b>void fillItem(int);</b> Fills the item variable with the appropriate item attributes at the corresponding given position.  <b>void handleImage();</b> Decodes the image corresponding to the item being currently handled in order to display it.  <b>void getView(int, View, ViewGroup);</b> Calls the Setup, fillItem, and handleImage functions.

**Class** *DevelopersDialogFragment*  
**Name**

<b>Attributes</b>	None
<b>Functions</b>	<b>public Dialog onCreateDialog(Bundle)</b> Creates a dialog displaying information about the developers of the application and allows the user to dismiss it or contact them via the function sendEmail.  <b>void sendEmail()</b> Opens an email screen to the developers so that the user may contact them with feedback.

**Class *FeedActivity***  
**Name**

**Attributes**

**temp** *Item*  
**obj** *JSONObject*  
**search** *Button*  
**name** *CheckBox*  
**code** *CheckBox*  
**text** *EditText*  
**feedArray** *ArrayList<Item>*

**Functions**

**void Setup();**

Prepares all visual elements by linking between them and their counterpart variables within the code.

**void feedListOnClickListener(int i)**

Passes information to the AnalyzingActivity class to display more information about the selected feed element.

**void JsonRequest(*String*);**

Uses the Android volley SDK to issue a JSONArray request, calling the script corresponding to the URL given to it. It then calls createFeedElement, passing the request's response to it or presents the appropriate error message.

**void createFeedElement(*JSONArray, int*);**

Uses the response passed to it to fill in the appropriate attributes in an Item variable at the given position to be displayed.

**void searchOnClickListener();**

Called when the search button is pressed, it performs the appropriate validations on the search information and parameters and then either uses the SearchCode or SearchName PHP or ShowFeed scripts depending on the search parameters.

**protected void onCreate(*Bundle*);**

Calls the Setup function, then uses the ShowFeed PHP script to populate the feed by the calling JsonRequest function.

**Class *HistoryActivity***  
**Name**

**Attributes**

**temp** *Item*  
**obj** *JSONObject*  
**mail** *String*  
**list** *ListView*  
**clear** *Button*  
**historyArray** *ArrayList<Item>*

**Functions**

**void Setup();**

Prepares all visual elements by linking between them and their counterpart variables within the code.

**void actOnSource();**

Examines the class that called this class, if it is indeed the history button from the SuccessActivity class, it acts as a history page, however if the source is AnalyzingActivity, then this class will display the detected symbols in the image taken by the user.

**void listOnClickListener(int i)**

Passes information to the AnalyzingActivity class to display more information about the selected history element. Or calls ResultActivity to display more information about the selected hieroglyph (if activity was called from an analyzed photo).

**void createHistoryElement(*JSONArray*, *int*);**

Uses the response passed to it to fill in the appropriate attributes in an Item variable at the given position to be displayed.

**void findHistory(*JSONArray*);**

Populates the history by sequentially calling createHistoryElement().

**void removeUser(*int*);**

Deletes the corresponding element in the ArrayList containing the history elements and calls the RemoveUser PHP script.

**void clearHistory();**

Clears the ArrayList containing the history elements and calls the ClearHistory PHP script.

**void JsonRequest(*String*, *int*);**

Uses the Android volley SDK to issue a JSONArray request, calling the script corresponding to the URL given to it. It then calls the findHistory, removeUser, or clearHistory.

**protected void onCreate(*Bundle*);**

Calls the Setup and JsonRequest functions and utilizes the FindHistory PHP script and the RemoveUser or ClearHistory scripts depending on the user's actions.

**Class *ImageSourceDialogFragment***  
**Name**

<b>Attributes</b>	<b>c</b> <i>Context</i>
	<b>camera</b> <i>Button</i>
<b>Functions</b>	<b>gallery</b> <i>Button</i>
	<b>void ImageSourceDialogFragment(<i>Activity</i>);</b> Initializes variables.
	<b>void onCreate(<i>Bundle</i>);</b> Sets button onClickListeners.
	<b>void onClick(<i>View</i>);</b> Requests the necessary permissions and then either opens the camera and then saves the taken image in a file and passes its URI to AnalyzingActivity or opens the gallery and captures the selected image and passes it to AnalyzingActivity.

**Class *Item***  
**Name**

<b>Attributes</b>	<b>Name</b> <i>String</i>
	<b>Code</b> <i>String</i>
	<b>FeedImage</b> <i>String</i>
	<b>SpaceWidth</b> <i>int</i>
	<b>history</b> <i>int</i>

**Class** *LogoutDialogFragment*  
**Name**

<b>Attributes</b>	<b>mContext</b> <i>Context</i>
<b>Functions</b>	<b>void updateSharedPreferences();</b> Updates the shared preferences on the mobile device, returning the values to 'none'.  <b>void Logout();</b> Calls the updateSharedPreferences function and then signs the account out on Firebase.  <b>void onCreateDialog(<i>Bundle</i>);</b> Creates the dialog and calls the Logout function upon the user's request.

**Class** *MainActivity (SignUpActivity)*  
**Name**

<b>Attributes</b>	<b>name</b> <i>EditText</i> <b>email</b> <i>EditText</i> <b>password</b> <i>EditText</i> <b>vpassword</b> <i>EditText</i> <b>Name</b> <i>String</i> <b>Email</b> <i>String</i> <b>Password</b> <i>String</i> <b>Vpassword</b> <i>String</i> <b>sent</b> <i>Boolean</i> <b>done</b> <i>Button</i> <b>mAuth</b> <i>FirebaseAuth</i> <b>connected</b> <i>boolean</i>
<b>Functions</b>	<b>void checkInternet();</b> Uses the ConnectivityManager and updates the connected variable as other function use this to determine whether the device has an active internet connection.  <b>void Setup();</b> Prepares all visual elements by linking between them and their

counterpart variables within the code.

**void createAccount();**

Interfaces to Firebase and uses the entered information to register an account.

**void collectInformation();**

Obtains the information entered by the user.

**void doneOnClickListener();**

Performs the needed validation on the entered information once the done button is pressed, and then either calls the createAccount function or displays the appropriate error message.

**void updateSharedPreferences();**

Updates the shared preferences on the mobile device.

**void sendVerificationEmail();**

Interfaces to Firebase to send a verification email to the user once they have created an account.

**protected void onCreate(*Bundle*);**

Calls the Setup and checkInternet functions.

**Class Name** *OneTimeActivity*

**Attributes**

**photo** *Button*  
**signin** *Button*  
**signup** *Button*  
**search** *Button*  
**connected** *boolean*

**Functions**

**void Setup();**

Prepares all visual elements by linking between them and their counterpart variables within the code.

**void checkInternet();**

Uses the ConnectivityManager and updates the connected variable as other function use this to determine whether the device has an active

internet connection.

**void photoOnClickListener();**

Called when the photo button is pressed. Obtains the required permissions to use the camera and then begins interfacing to the image manipulation system.

**void signupOnClickListener();**

Called when the signup button is pressed. Opens the sign up page.

**void signinOnClickListener ();**

Called when the signin button is pressed. Opens the sign in page.

**void searchOnClickListener ();**

Called when the search button is pressed. Opens the dictionary page.

**protected void onCreate(*Bundle*);**

Calls the Setup and checkInternet functions.

**protected void onActivityResult(*int, int, Intent*);**

Called when the user is finished choosing and manipulating an image, it passes the image to the analyzing page and opens the page,

**Class *OpeningActivity***  
**Name**

***Attributes***

**email** *String*  
**firsttime** *SharedPreferences*  
**cont** *Button*  
**signin** *Button*  
**singup** *Button*  
**connected** *boolean*

***Functions***

**void Setup();**  
Prepares all visual elements by linking between them and their counterpart variables within the code.

**void checkInternet();**  
Uses the ConnectivityManager and updates the connected variable as other function use this to determine whether the device has an active

internet connection.

**void checkFirst();**

Checks the shared preference under the key 'first' to determine whether this is the first time the user opens the application after installing it or logging out and either displays the opening page (for signing up, in or one time usage) or goes directly into the main menu (if the user is already signed in) based upon its value.

**void updateSharedPreferences();**

Sets the shared preference under the key 'first' to false, called upon signing up or in so that this page does not get displayed the immediate consecutive time until the user logs out or deletes the application.

**void signupOnClickListener();**

Called when the signup button is pressed. Opens the sign-up page.

**void signinOnClickListener ();**

Called when the signin button is pressed. Opens the sign in page.

**protected void onCreate(*Bundle*);**

Calls the checkInternet and checkFirst function and opens the main menu screen if applicable.

**protected onStop();**

Provides the same function as updateSharedPreferences as another page is being loaded (to ensure the operation has been done).

**Class *ResultsActivity***  
**Name**

**Attributes**

**code** *TextView*  
**description** *TextView*  
**meaning** *TextView*  
**link** *TextView*  
**linktitle** *TextView*  
**listlink** *TextView*  
**alimage** *ImageView*  
**gImage** *Imageview*  
**u** *Uri*



## Functions

**Code** *String*  
**imageString** *String*

### **void Setup();**

Prepares all visual elements by linking between them and their counterpart variables within the code. It also obtains the image that the user has taken/uploaded and manipulated from the previous screen and displays it.

### **void actOnSource();**

Examines the class that called this class, if it is either the HistoryActivity or FeedActivity classes, it will display information about the element the user has selected. If it is AnalyzingActivity then it will display information about the symbols detected in the image that the user has taken.

### **void searchDatabase();**

Issues a volley request that returns a JSONArray containing information about the detected hieroglyph using the PHP script SearchGCode.

### **void displayResults();**

Fills all the relevant fields with the information obtained from searchDatabase.

### **protected void onCreate(*Bundle*);**

Calls the Setup and searchDatabase functions.

## Class **SearchActivity** Name

### Attributes

**code** *TextView*  
**description** *TextView*  
**meaning** *TextView*  
**link** *TextView*  
**linktitle** *TextView*  
**listlink** *TextView*  
**codeTitle** *TextView*  
**descriptionTitle** *TextView*  
**meaningTitle** *TextView*  
**title** *TextView*

## Functions

**glImage** *ImageView*  
**u** *Uri*  
**Code** *String*  
**imageString** *String*  
**searchButton** *Button*  
**input** *EditText*

### **void Setup();**

Prepares all visual elements by linking between them and their counterpart variables within the code. It also sets the result elements to be invisible until the user has issued a search.

### **void searchDatabase();**

Issues a volley request that returns a JSONArray containing information about the detected hieroglyph using the PHP script SearchGCode.

### **void displayResults();**

Fills all the relevant fields with the information obtained from searchDatabase.

### **protected void onCreate(*Bundle*);**

Calls the Setup and searchDatabase functions.

## Class *SignInActivity* Name

### Attributes

**done** *Button*  
**email** *EditText*  
**password** *EditText*  
**mail** *String*  
**pass** *String*  
**mAuth** *FirebaseAuth*  
**connected** *boolean*

### Functions

#### **void Setup();**

Prepares all visual elements by linking between them and their counterpart variables within the code.

#### **void checkInternet();**

Uses the ConnectivityManager and updates the connected variable as other function use this to determine whether the device has an active internet connection.

**void doneOnClickListener();**

Called when the done button is pressed, obtains the information entered by the user and interfaces to Firebase to log the user in or display the appropriate error message.

**protected void onCreate(*Bundle*);**

Calls the Setup and CheckInternet functions.

**void updateSharedPreferences();**

Updates the user's shared preferences based upon their entered data.

**void checkIfEmailVerified();**

Interfaces to Firebase to check that the user has verified their account before opening the main menu screen.

**Class *SubmitActivity***  
**Name**

**Attributes**

**submit** *Button*  
**code** *EditText*  
**description** *EditText*  
**Code** *String*  
**Description** *String*  
**ImageView** *image*  
**connected** *boolean*

**Functions**

**void Setup();**

Prepares all visual elements by linking between them and their counterpart variables within the code. Obtains the image that the user has taken/selected and manipulated and displays it as well.

**void checkInternet();**

Uses the ConnectivityManager and updates the connected variable as other function use this to determine whether the device has an active internet connection.

**void submitOnClickListener();**

Called when the submit button is clicked, it obtains the information entered by the user, validates it, and then calls the sendEmail function.

**protected void onCreate(*Bundle*);**

Calls the Setup and CheckInternet functions.

**void sendEmail();**

Sends the user's information to the developers in an email if there is an email client installed on the user's device. The email includes the image chosen, its Gardiner's code, and a description (optional).

**Class *SuccessActivity***  
**Name**

***Attributes***

**photo** *Button*  
**submit** *Button*  
**connected** *boolean*  
**feed** *Button*  
**history** *Button*  
**logout** *Button*  
**image** *ImageView*  
**resultUri** *Uri*

***Functions***

**void Setup();**

Prepares all visual elements by linking between them and their counterpart variables within the code.

**void photoOnClickListener();**

Called when the photo button is pressed. Obtains the required permissions to use the camera and then begins interfacing to the image manipulation system.

**void checkInternet();**

Uses the ConnectivityManager and updates the connected variable as other function use this to determine whether the device has an active internet connection.

**void logoutOnClickListener();**

Called when the logout button is pressed. Checks if the device has an active internet connection. Then interfaces to the LogoutDialogFragment.

**void historyOnClickListener();**

Called when the history button is pressed. Checks if the device has an active internet connection. Then interfaces to the history screen.

**void feedOnClickListener();**

Called when the feed button is pressed. Checks if the device has an active internet connection. Then interfaces to the feed screen.

**void submitOnClickListener();**

Called when the submit button is pressed. Checks if the device has an active internet connection. Then interfaces to the submit screen.

**void searchOnClickListener ();**

Called when the search button is pressed. Opens the dictionary page.

**protected void onCreate(*Bundle*);**

Calls the Setup and checkInternet functions.

**void startActivity(*int*);**

Starts either the analyzing or submit screens based on the button pressed and thus the incoming input.

**void onActivityResult(*int, int, Intent*);**

Calls startActivity with the relevant input based on whether the photo the user has taken/chosen and manipulated is meant for classification or submission.

**Class Name** *TensorFlowImageClassifier*

<b>Attributes</b>	<b>MAX_RESULTS</b> <i>int</i> <b>THRESHOLD</b> <i>float</i> <b>inputName</b> <i>String</i>
-------------------	--

## Functions

**outputName** *String*  
**inputSize** *int*  
**imageMean** *int*  
**imageStd** *float*  
**labels** *Vector<String>*  
**intValues** *int[ ]*  
**floatValues** *float[ ]*  
**outputs** *float[ ]*  
**outputNames** *String[ ]*

**Classifier create(AssetManager assetManager, String modelFileName, String labelFileName, int inputSize, int imageMean, float imageStd, String inputName, String outputName);**

This initializes all the needed variables and creates the classifier. The model file name references the .pb file, the label file name references the file containing the names of the labels given to different classifications, the input size is always set to 150 and the image mean and STD are set based on trial and error. The input name represents the first layer's first input node name in the network while the output represents the last output node in the final layer.

**Float [ ] recognizeImage(Bitmap bitmap);**

Normalizes the image and feeds it into the classifier, obtains the highest MAX\_RESULTS predictions and returns them.

## Class *VerificationActivity* Name

### Attributes

**verify** *Button*  
**tick** *ImageView*  
**prgrs** *ProgressBar*  
**name** *String*  
**email** *String*  
**password** *String*  
**mAuth** *FirebaseAuth*  
**connected** *boolean*

### Functions

**void Setup();**

Prepares all visual elements by linking between them and their counterpart variables within the code. It also obtains the shared preferences of the user in case the application closes before the user is able to verify their account.

**void obtainPermission();**

Obtains user permission to read external storage.

**void verifyOnClickListener();**

Called when the verify button is pressed. It checks for an active internet connection and then interfaces to Firebase and signs the user into their account.

**protected void onCreate(*Bundle*);**

Calls the checkInternet, obtainPermission, and Setup functions.

**void updateSharedPreferences();**

Updates the shared preferences of the user stored on the mobile device.

**void checkInternet();**

Uses the ConnectivityManager and updates the connected variable as other function use this to determine whether the device has an active internet connection.

**void checkIfEmailVerified();**

Interfaces to Firebase and checks whether the user has verified their account or not, either displaying an error message or opening the main menu screen.

### 3.2.2 iOS Application

Below is a detailed description of each class's attributes and functions. Here, no distinction is made between main classes and sub-classes. In the attributes section, the attribute name is in bold, and their data type is in italic. In the functions section, the function name is in bold, and arguments are in bold and italic.

It should be noted that any functions taking default variables not added by the user and not used within the function's purpose are omitted. All classes have a didReceiveMemoryWarning function and other various utility functions not discussed

within the scope of this document. The Reachability class was obtained online and contains functions that are able to check the connectivity of the mobile device so that the application is able to determine which functions the user will be able to perform, and which functions they will not be able to perform until they connect to the internet. It should also be noted that all classes except for `TableViewCell` and `TableViewCell2` extend `UIViewController`. The `HistoryViewController` and `FeedViewController` also extend `UITableViewDataSource` and `UITableViewDelegate`. `OneTimeViewController` and `SuccessViewController` also extend `UIImagePickerControllerDelegate` and `UINavigationControllerDelegate`. And `OneTimeViewController`, `SuccessViewController`, and `SubmitViewController` also extend `MFMailComposeViewControllerDelegate`. Additionally, `UITableView+Cache`, `CornerpointClientProtocol`, `CroppableImageViewDelegateProtocol`, `CornerpointView`, and `Utils` are also third-party classes that are used for caching images and adding functionality to crop images that the user has taken.

### **Class *AnalyzingViewController*** **Name**

<b>Attributes</b>	<b>userDefaults</b> <i>UserDefaults.standard</i> <b>pickedImageData</b> <i>UIImage</i> <b>onetime</b> <i>Bool</i> <b>pickedimage</b> <i>UIImageView</i> <b>inceptionv3model</b> <i>Classifier</i> <b>results</b> <i>[String]</i> <b>arrayOfCodes</b> <i>[String]</i> <b>arrayOfPhotos</b> <i>[String]</i> <b>images</b> <i>[UIImage]</i> <b>imagesBW</b> <i>[UIImage]</i> <b>allresults</b> <i>NSMutableArray</i> <b>results1</b> <i>NSMutableArray</i> <b>results2</b> <i>NSMutableArray</i>
<b>Functions</b>	<b>void viewDidLoad();</b> Updates the <code>UIImageView</code> <code>pickedImage</code> with the data received in <code>pickedImageData</code> from the previous page. It also calls the <code>predict</code> function which handles classification.  <b>void predict();</b> Obtains the prediction from the classifier using the <code>resize</code> and <code>buffer</code>



functions as needed and matches it to the appropriate label from the given text file and then either calls the ResultsViewController or calls uploadResults depending on whether or not the user is registered.

**void uploadResults();**

Prepares all information concerning this post and uploads it using the AddUserDuaKhety PHP script so that it appears in the user's history and all users' feeds in the future.

**CVPixelBuffer buffer(UImage image);**

Receives an image and transforms it into a CVPixelBuffer which is understood by the classifier.

**CVPixelBuffer resize(CVPixelBuffer pixelBuffer);**

Resizes any incoming image to 150x150 which is what the classifier requires to optimize results as this was the size of the dataset.

**void viewDidAppear();**

Based on information received from the previous page regarding whether the user is signed in or is a onetime user, this function either uploads the data to appear in the user's history and other user's feeds in the future, or it only continues to the results page (if it is a onetime user).

**void prepare();**

Sends information regarding the image and whether the user is onetime or not to the next page (which is the results page).

**String randomString(int);**

Generates a random string to act as an identifying code for the image that users can search for in the FeedViewController

**Class CameraViewController**  
**Name**

<b>Attributes</b>	<b>submit</b> <i>Int</i>
	<b>captureSession</b> <i>AVCaptureSession</i>
	<b>backCamera</b> <i>AVCaptureDevice</i>
	<b>frontCamera</b> <i>AVCaptureDevice</i>
	<b>currentDevice</b> <i>AVCaptureDevice</i>

**photoOutput** *AVCapturePhotoOutput*  
**cameraPreviewLayer** *AVCaptureVideoPreviewLayer*  
**image** *UIImage*  
**toggleCameraGestureRecognizer** *UISwipeGestureRecognizer*  
**zoomInGestureRecognizer** *UISwipeGestureRecognizer*  
**zoomOutGestureRecognizer** *UISwipeGestureRecognizer*

## **Functions**

### **void galleryButtonPressed();**

Opens the device's photo gallery to choose a photo instead of taking one.

### **void imagePickerController();**

Compresses the chosen image, saves it to the user's gallery and calls the CroppingViewController.

### **void takephotoButtonPressed();**

Captures the current image, compresses it, saves it to the user's gallery and calls the CroppingViewController.

### **void prepare();**

Sends information regarding the image and whether the user is onetime or not to the next page (which is the cropping page).

### **void viewDidLoad();**

Initializes all recognizers and sets up the camera.

## **Camera Setup Functions**

The application implements its own camera to be in control of capturing settings and format the resulting image as needed as well as to be able to interface to the gallery from within the camera which iOS' default camera interface is unable to do. The following functions build upon the AVFoundation framework and allow a custom camera to be implemented:

setupCaptureSession( )  
setupDevice( )  
setupInputOutput( )  
setupPreviewLayer( )  
switchCamera( )  
zoomIn( )  
zoomOut( )

**Class** *CroppingViewController*  
**Name**

<b>Attributes</b>	<b>Submit</b> <i>Int</i> <b>pickedImageData</b> <i>UIImage</i> <b>onetime</b> <i>Bool</i> <b>cropButton</b> <i>UIButton</i>
<b>Functions</b>	<b>void haveValidCropRect();</b> Checks whether the cropping rectangle is in a valid position or not.  <b>void cropButtonPressed();</b> Saves the cropped image to the device's gallery and examines the class that called this one depending on the Submit variable. If the intention was classification, the AnalyzingViewController is called, if submission was the intent then the SubmitViewController is called.  <b>void prepare();</b> Sends information regarding the image and whether the user is onetime or not (if the AnalyzingViewController is the destination) or whether the image has been cropped or not (if the SubmitViewController is the destination).

**Class** *FeedViewController*  
**Name**

<b>Attributes</b>	<b>arrayOfOwners</b> <i>[String]</i> <b>arrayOfPhotos</b> <i>[String]</i> <b>arrayOfCodes</b> <i>[String]</i> <b>arrayOfDescriptions</b> <i>[String]</i> <b>tv</b> <i>UITableView</i> <b>segment</b> <i>UISegmentedControl</i> <b>searchBox</b> <i>UITextField</i> <b>searchButton</b> <i>UIButton</i> <b>chosenIndex</b> <i>Int</i>
-------------------	--

<b>Functions</b>	<b>void searchButtonPressed();</b> Empties all arrays and refills them based on the result of a search query that uses either SearchNameDuaKhety or SearchCodeDuaKhety depending on the user's selection on the segment which calls the search function with the appropriate URL.
	<b>void search();</b> Performs the dataTask which obtains information from the database using the relevant script based on the URL sent to it.
	<b>void numberOfRowsInSection();</b> Returns the number of elements in the arrayOfOwners array which corresponds to the number of feed elements available to display.
	<b>void cellForRowAt();</b> Updates the tableview elements with information from the arrays that have been filled from the previous page which is the SuccessViewController.
	<b>void didSelectRowAt();</b> Calls the AnalyzingViewController and send information about the selected image to it.

**Class *HistoryViewController***  
**Name**

<b>Attributes</b>	<b>arrayOfOwners</b> [String] <b>arrayOfPhotos</b> [String] <b>arrayOfCodes</b> [String] <b>arrayOfDescriptions</b> [String] <b>tv</b> UITableView
<b>Functions</b>	<b>void clearHistoryButtonPressed();</b> Opens a dialog for the user to warn them that if they accept, all elements will be removed from all arrays and removed from the database permanently, preventing them from appearing in their history and all users' feeds in the future using the ClearHistoryDuaKhety PHP script.

**void numberOfRowsInSection();**

Returns the number of elements in the arrayOfOwners array which corresponds to the number of feed elements available to display.

**void didSelectRowAt();**

If the ViewController has been selected to display the actual history, it opens a dialog for the user to choose whether to delete the element, in which case this element will be removed from all arrays and removed from the database permanently, preventing it from appearing in their history and all users' feeds in the future using the RemoveUserDuaKhety PHP script. Or if they want to view more information, it calls the AnalyzingViewController and send information about the selected image to view more information about it.

If the page has been called from an analyzed photo, this calls ResultsViewController to display more information about the individual hieroglyph.

**void cellForRowAt();**

Updates the tableview elements with information from the arrays that have been filled from the previous page which is the SuccessViewController.

**Class**   ***OneTimeViewController***  
**Name**

<b>Attributes</b>	<b>image</b> <i>UIImage</i>
<b>Functions</b>	<b>void sendEmail();</b> Sets the appropriate recipients and subject and then interfaces to email providers installed on the users' phones to email the developers with feedback.  <b>void mailComposeController();</b> Closes the third-party email provider when the user dismisses it and returns to the application.  <b>void developersButtonPressed();</b> Displays a dialog containing information about the application's developers and allows the user to send them an email with feedback.

**void creditsButtonPressed();**

Displays a dialog containing information about important entities that made developing the application possible.

**void searchButtonPressed();**

Checks the phone's internet connection, and if possible opens the SearchViewController dictionary page to search for hieroglyphics.

**void takePhotoButtonPressed();**

Interfaces to the device's camera and then allows the user to edit the image.

**void uploadPhotoButtonPressed();**

Interfaces to the device's gallery and then allows the user to edit the image.

**void imagePickerController();**

Saves the image taken or chosen to the user's device and then passes it to the next page (AnalyzingViewController) and closes the camera or gallery when the user dismisses them or has finished selecting/taking an image.

**void prepare();**

Either passes the image and information regarding whether the user is onetime or not to the next page (if the next page is AnalyzingViewController) or only the user information (if the next page is SearchViewController).

***Class Name***    ***OpeningViewController***

***Attributes***

**None**

***Functions***

**void viewDidAppear();**

Checks whether the user has previously logged in or signed up or this is their first time to open the application (or their last operation was to log out of the application) and opens either the SuccessViewController (if the user is logged in) or the OpeningViewController.

**Class Name**    **ResultsController**

<b>Attributes</b>	<b>dictionaryLink</b> UILabel <b>usefullinksTitle</b> UILabel <b>usefullinksField</b> UILabel <b>webImage</b> UIImageView <b>pickedImage</b> UIImageView <b>codeField</b> UILabel <b>meaningField</b> UILabel <b>descriptionField</b> UILabel <b>onetime</b> Bool <b>waiting</b> UIActivityIndicatorView <b>pickedDataImage</b> UIImage
<b>Functions</b>	<b>void backButtonPressed();</b> Either returns to the OneTimeViewController or the SuccessViewController depending on information obtained from onetime about the status of the user (logged in or onetime).  <b>void tap1();</b> Opens the link displayed in the usefullinksField.  <b>void tap2();</b> Opens the link displayed in the dictionaryLink field.  <b>void viewDidLoad();</b> Hides the results fields until actual results are obtained and displays the activity indicatory (waiting).  <b>void viewDidAppear();</b> Checks for internet connectivity and if applicable it uses the SearchGCodeDuaKhety PHP script to display additional information about the obtained hieroglyph.

**Class Name**    **SearchViewController**

<b>Attributes</b>	<b>dictionaryLink</b> UILabel <b>usefullinksTitle</b> UILabel <b>usefullinksField</b> UILabel <b>webImage</b> UIImageView
-------------------	--

**codeField** *UILabel*  
**codeTitle** *UILabel*  
**meaningField** *UILabel*  
**meaningTitle** *UILabel*  
**resultsTitle** *UILabel*  
**descriptionTitle** *UILabel*  
**descriptionField** *UILabel*  
**codeInput** *UITextField*  
**onetime** *Bool*  
**waiting** *UIActivityIndicatorView*

### **Functions**

**void backButtonPressed();**

Either returns to the OneTimeViewController or the SuccessViewController depending on information obtained from onetime about the status of the user (logged in or onetime).

**void searchButtonPressed();**

Checks for internet connectivity and if applicable it uses the SearchGCodeDuaKhety PHP script to display information about the hieroglyph corresponding to the entered Gardiner code (or an error message if an invalid code is entered).

**void tap1();**

Opens the link displayed in the usefullinksField.

**void tap2();**

Opens the link displayed in the dictionaryLink field.

**void viewDidLoad();**

Hides the results fields until actual results are obtained and displays the activity indicatory (waiting).

### **Class Name** *SignInViewController*

### **Attributes**

**emailField** *UITextField*  
**passwordField** *UITextField*  
**nameField** *UITextField*



<b>Functions</b>	<b>void signInButtonPressed();</b> Checks for internet connectivity and if applicable, checks that all the fields contain valid information and if so, it attempts to sign the user in (by interfacing to Firebase) and either continues on to the SuccessViewController or displays an error message. It also updates the SharedPreferences on the user's device.
------------------	---

**Class Name**    **SignUpViewController**

<b>Attributes</b>	<b>emailField</b> <i>UITextField</i> <b>passwordField</b> <i>UITextField</i> <b>nameField</b> <i>UITextField</i> <b>doneButton</b> <i>UIButton</i> <b>vpasswordField</b> <i>UITextField</i> <b>Name</b> <i>String</i> <b>Email</b> <i>String</i> <b>Password</b> <i>String</i> <b>vPassword</b> <i>String</i>
<b>Functions</b>	<b>void doneButtonPressed();</b> Checks for internet connectivity and if applicable, checks that all the fields contain valid information and if so, it attempts to sign the user up (by interfacing to Firebase) and either continues on to the VerificationViewController or displays an error message. It also updates the SharedPreferences on the user's device.

**Class Name**    **SubmitViewController**

<b>Attributes</b>	<b>pickedImage</b> <i>UIImageView</i> <b>codeField</b> <i>UITextField</i> <b>descriptionField</b> <i>UITextField</i>
-------------------	--

<b>Functions</b>	<b>void submitButtonPressed();</b> Checks that an image has been chosen and a Gardiner code has been inputted, and then calls sendEmail.
	<b>void sendEmail();</b> Sets the appropriate recipients, subject, body, and attachments for an email and then interfaces to a third-party email provider if possible to send an email to the developers with the submission image and its information.
	<b>void mailComposeController();</b> Closes the third-party email provider when the user dismisses it.
	<b>void imagePickerController();</b> Saves the taken/chosen image to the user's gallery and closes the camera or gallery when the user dismisses it or has finished selecting/taking an image.
	<b>void takePhotoButtonPressed();</b> Interfaces to the device's camera and then allows the user to edit the image.
	<b>void uploadPhotoButtonPressed();</b> Interfaces to the device's gallery and then allows the user to edit the image.

**Class Name**    **SuccessViewController**

<b>Attributes</b>	<b>arrayOfOwners</b> [String]
	<b>arrayOfPhotos</b> [String]
	<b>arrayOfCodes</b> [String]
	<b>arrayOfDescriptions</b> [String]
	<b>image</b> UIImage
	<b>waiting</b> UIActivityIndicatorView

## ***Functions***

### **void sendEmail();**

Sets the appropriate recipients and subject and then interfaces to email providers installed on the users' phones to email the developers with feedback.

### **void mailComposeController();**

Closes the third-party email provider when the user dismisses it and returns to the application.

### **void developersButtonPressed();**

Displays a dialog containing information about the application's developers and allows the user to send them an email with feedback.

### **void creditsButtonPressed();**

Displays a dialog containing information about important entities that made developing the application possible.

### **void searchButtonPressed();**

Checks the phone's internet connection, and if possible opens the SearchViewController dictionary page to search for hieroglyphics.

### **void takePhotoButtonPressed();**

Interfaces to the device's camera and then allows the user to edit the image.

### **void uploadPhotoButtonPressed();**

Interfaces to the device's gallery and then allows the user to edit the image.

### **void imagePickerController();**

Saves the image taken or chosen to the user's device and then passes it to the next page (AnalyzingViewController) and closes the camera or gallery when the user dismisses them or has finished selecting/taking an image.

### **void prepare();**

Either passes the image and information regarding whether the user is onetime or not to the next page (if the next page is AnalyzingViewController) or only the user information (if the next page is SearchViewController), or it passes the information contained in the

various arrays obtained from PHP scripts (if the next page is HistoryViewController or FeedViewController).

**void submitButtonPressed();**

Checks for internet connectivity and if possible opens the SubmitViewController page.

**void logoutButtonPressed();**

Checks for internet connectivity and if possible opens a dialog that can allow the user to log out, which would return them to OpeningViewController and erase any stored SharedPreferences.

**void feedButtonPressed();**

Checks for internet connectivity and if possible opens the feedViewController page. It uses the ShowFeedDuaKhety PHP script to fill the arrays with content to be displayed.

**void historyButtonPressed();**

Checks for internet connectivity and if possible opens the historyViewController page. It uses the FindHistoryDuaKhety PHP script to fill the arrays with content to be displayed.

**Class    *TableViewCell***  
**Name**

<b>Attributes</b>	<b>PhotoDescription</b> <i>UILabel</i>
	<b>PhotoGraph</b> <i>UIImageView</i>
	<b>code</b> <i>UILabel</i>
	<b>owner</b> <i>UILabel</i>
<b>Functions</b>	This class describes the content and layout of an element in the HistoryViewController's TableView.

**Class    *TableViewCell2***  
**Name**

<b>Attributes</b>	<b>desc</b> <i>UILabel</i>
	<b>owner</b> <i>UILabel</i>

	<b>code</b> <i>UILabel</i> <b>photo</b> <i>UIImageView</i>
<b>Functions</b>	This class describes the content and layout of an element in the FeedViewController's TableView.

**Class Name** *VerificationViewController*

<b>Attributes</b>	<b>spinner</b> <i>UIActivityIndicatorView</i> <b>verifyButton</b> <i>UIButton</i>
<b>Functions</b>	<b>void verifyButtonPressed();</b> Signs the user in by interfacing to Firebase and then checks whether the user's account has been verified or not (based on whether they have clicked on the link in the email sent to them when they signed up or not) and based upon this it either displays an error message or continues to the SuccessViewController.

### 3.3 Design Rationale

The layered architecture was chosen as the bottom layers are mainly the functions that will be available under any conditions as well as the resources needed by the entire application. Any new added facilities or features could then be added on top of these as well as the features requiring internet connectivity (closer to the server side). The layered architecture also provides modularity, making it easier to alter layers while keeping the interfaces constant as well as making the development process easier by having clearly defined sections that can be split amongst the team.

The client-server architecture was chosen for the upper part of the system since a range of services (the social features, account system, submission system, and history system) and shared data in the database will need to be accessed from a large number of locations. This allows services to be distributed across a network.

## **4. DATA DESIGN**

### **4.1 Data Description**

All data obtained from the databases is either in the form of strings or images which are stored as BLOBs on the database and are decoded and encoded upon downloading and uploading. The requests made to the database always return either JSONArrays or JSONObjects for easier sorting of the data.

#### **4.1.1 Android Application**

Most data types used throughout the system are simple UI or data containers. An ArrayList is used to deal with instances where the Item class is needed, namely the Feed and History classes. UserDefaults variables are also used when needed. Images are stored within ImageViews by regarding either their Uri's or their Bitmaps.

#### **4.1.2 iOS Application**

Most data types used in the iOS application also correspond to UI or visual elements. The exception being simple Boolean and String values used whenever needed, as well as SharedPreferences variables. Data corresponding to elements in the History and Feed TableViews are typically stored in NSArray's to have dynamically modifiable lengths and the data returned from an requests to the database are fed into an NSDictionary before being passed to the arrays to make accessing them more convenient.

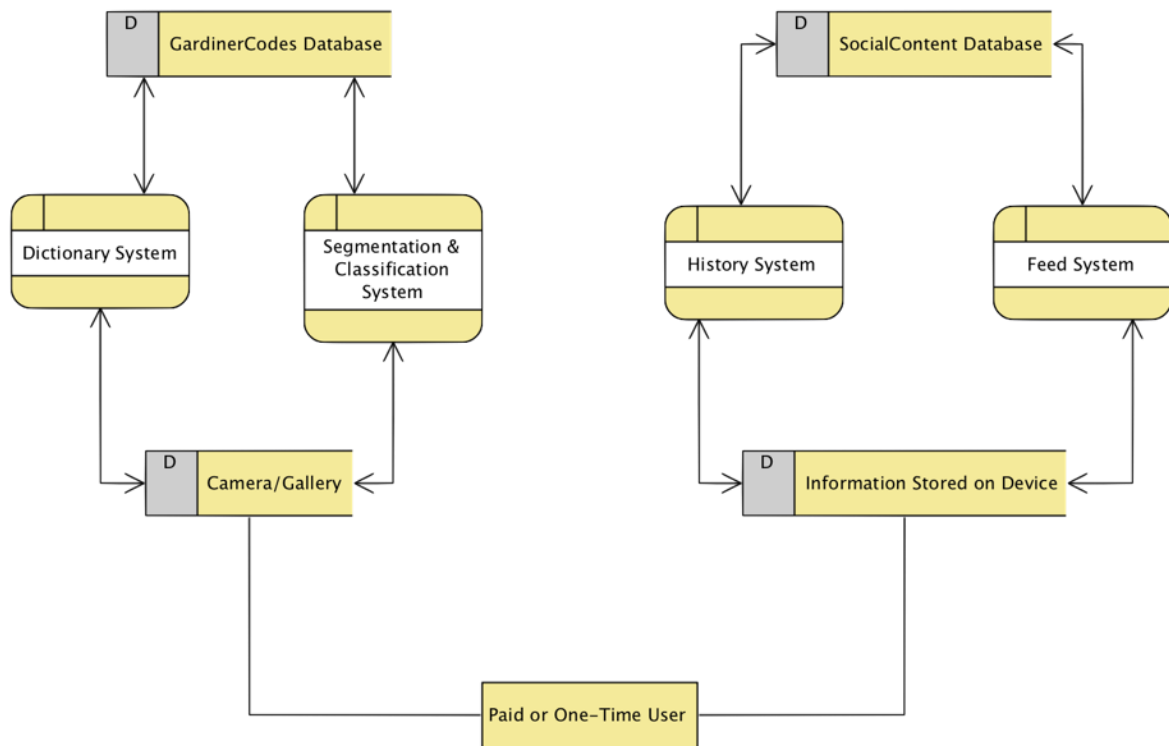


Figure 2. Data Flow Diagram

## 4.2 Data Dictionary

### 4.2.1 Android Application

#### A

alimage *ImageView*

#### B

blurred *Mat*

bmp *Bitmap*

#### C

C *TextView*

c *Item*

c *Context*

camera *Button*

centres *Mat*

classifierResult *String[ ]*

clean *Mat*

cimage *CropImageView*

code *CheckBox*

code *TextView*

Code *String*

codeTitle *TextView*

connected *boolean*

confidence *float*

crop *Button*

## **D**

description *TextView*

descriptionTitle *TextView*

## **E**

Edge\_detected *Mat*

email *EditText*

encodedString *String[ ]*

## **F**

feedArray *ArrayList<Item>*

FeedImage *String*

firsttime *SharedPreferences*

floatValues *float[ ]*

## **G**

glImage *Imageview*

gallery *Button*

## **H**

history *int*

historyArray *ArrayList<Item>*

## **I**

I *ImageView*

id *String*

image *ImageView*

imageMean *int*

imageStd *float*

imageString *String*

img\_BW *Mat*

img\_C *Mat*

Input *EditText*

inputName *String*

inputSize *int*

intValues *int[ ]*

## **L**



L *LinearLayout*  
labels *Mat*  
link *TextView*  
linktitle *TextView*  
list *ListView*  
listlink *TextView*  
location *RectF*

## **M**

mail *String*  
mAuth *FirebaseAuth*  
MAX\_RESULTS *int*  
mContext *Context*  
meaning *TextView*  
meaningTitle *TextView*

## **N**

N *TextView*  
name *CheckBox*  
Name *String*  
name *EditText*

## **O**

obj *JSONObject*  
onetime *Boolean*  
outputName *String*  
outputNames *String[ ]*  
outputs *float[ ]*

## **P**

params *ViewGroup.LayoutParams*  
password *EditText*  
Password *String*  
prgrs *ProgressBar*

## **R**

RequestParams *RequestParams*  
resultUri *Uri*

## **S**

sent *Boolean*  
SpaceWidth *int*  
Stats *Mat*

## **T**

temp *Item*  
text *EditText*  
THRESHOLD *float*  
tick *ImageView*  
title *String*  
title *TextView*

## **U**

u *Uri*  
uploadImage *String*

## **V**

vpassword *EditText*  
Vpassword *String*

### **4.2.2 iOS Application**

## **A**

allresults *NSMutableArray*  
arrayOfCodes *String[ ]*  
arrayOfDescriptions *String[ ]*  
arrayOfOwners *String[ ]*  
arrayOfPhotos *String[ ]*

## **B**

backCamera *AVCaptureDevice*

## **C**

cameraPreviewLayer *AVCaptureVideoPreviewLayer*  
captureSession *AVCaptureSession*  
chosenIndex *Int*  
codeField *UITextField*  
codeInput *UITextField*  
currentDevice *AVCaptureDevice*

## **D**

descriptionField *UITextField*

## **E**

Email *String*  
emailField *UITextField*

## **F**

frontCamera *AVCaptureDevice*

## **I**

image *UIImage*

images *[UIImage]*  
imagesBW *[UIImage]*  
inceptionv3model *Classifier*

## **N**

Name *String*  
nameField *UITextField*

## **O**

onetime *Bool*

## **P**

Password *String*  
passwordField *UITextField*  
pickedimage *UIImageView*  
pickedImageData *UIImage*  
PhotoGraph *UIImageView*  
photoOutput *AVCapturePhotoOutput*

## **R**

results *[String]*  
results1 *NSMutableArray*  
results2 *NSMutableArray*

## **S**

searchBox *UITextField*  
segment *UISegmentedControl*  
submit *Int*

## **T**

toggleCameraGestureRecognizer *UISwipeGestureRecognizer*  
tv *UITableView*

## **U**

userDefaults *UserDefaults.standard*

## **V**

vpasswordField *UITextField*  
vPassword *String*

## **W**

webImage *UIImageView*

## **Z**

zoomInGestureRecognizer *UISwipeGestureRecognizer*  
zoomOutGestureRecognizer *UISwipeGestureRecognizer*

## 5. HUMAN INTERFACE DESIGN

### 5.1 Overview of User Interface

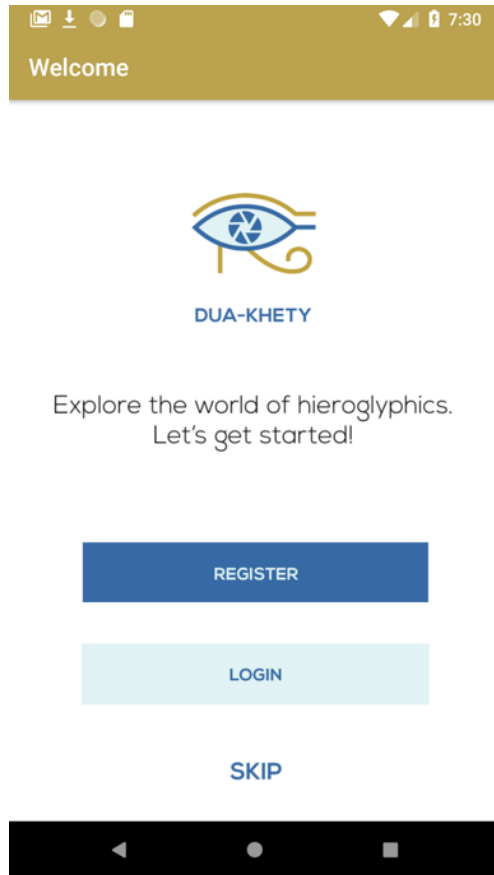
When the user opens the application for the first time upon installation or logging out, they will be presented with the **opening** screen that allows them to sign in, sign up, or continue as a one-time user. The **sign up** and **sign in** pages allow users to enter the needed information and the sign-up page is followed by a **verification** screen that is displayed until the user clicks on a link in an email sent to them upon signing up. One-time users have a limited **menu** screen containing options to analyze a photo, search for a symbol, sign in, or sign up. Registered users have a more extensive menu that allows them to analyze a photo, view the social feed, view their own history, search for a symbol, or submit a symbol. Choosing to analyze a photo opens the camera or gallery based on the user's choice, then a **cropping** interface appears followed a waiting screen while the application segments and classifies the images (**analyzing**). After analysis, a list of the detected symbols is shown (which is the same page used to display the users' search **history**). Clicking on a symbol from the list would open a **results** page with more information about it. Open the **social feed**, **symbol dictionary**, and **photo submission** all open their respective pages.

### 5.2 Page Tree:

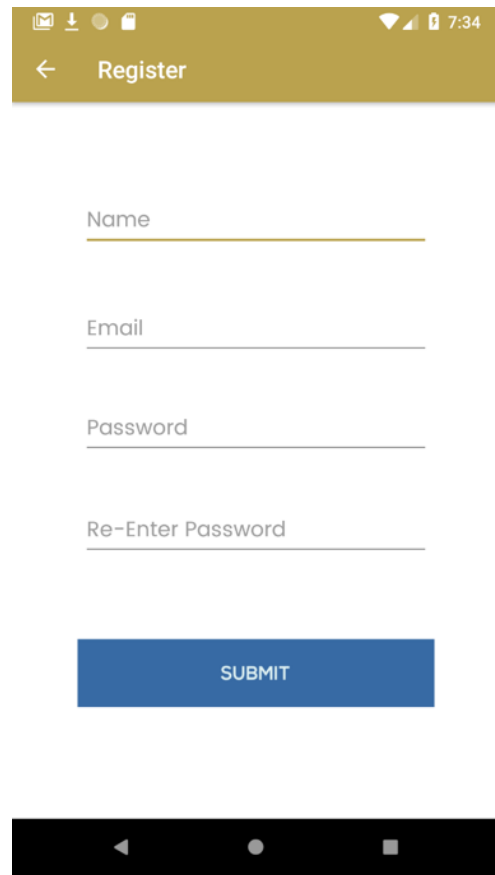
- Opening Page
  - Sign In
  - Sign Up
  - One Time Menu
    - Search Symbol
    - Register
    - Login
    - Analyze Photo
      - Cropping
      - History
      - Results
  - Success Menu
    - History
    - Submit Symbol
    - Search Symbol
    - Photo Feed
    - Analyze Photo
      - Cropping
      - History
      - Results

## 5.3 Screen Images, Screen Objects and Actions

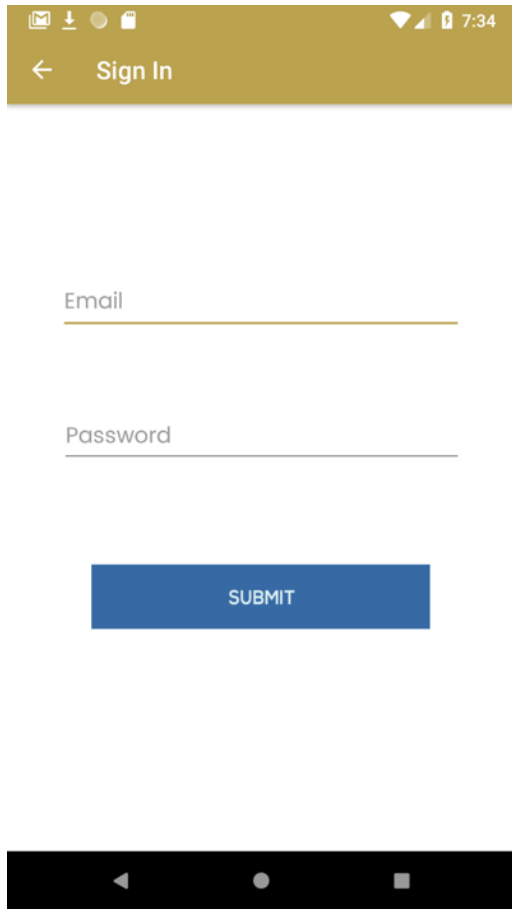
### 5.3.1 Android Application



Welcome/Start Up Screen

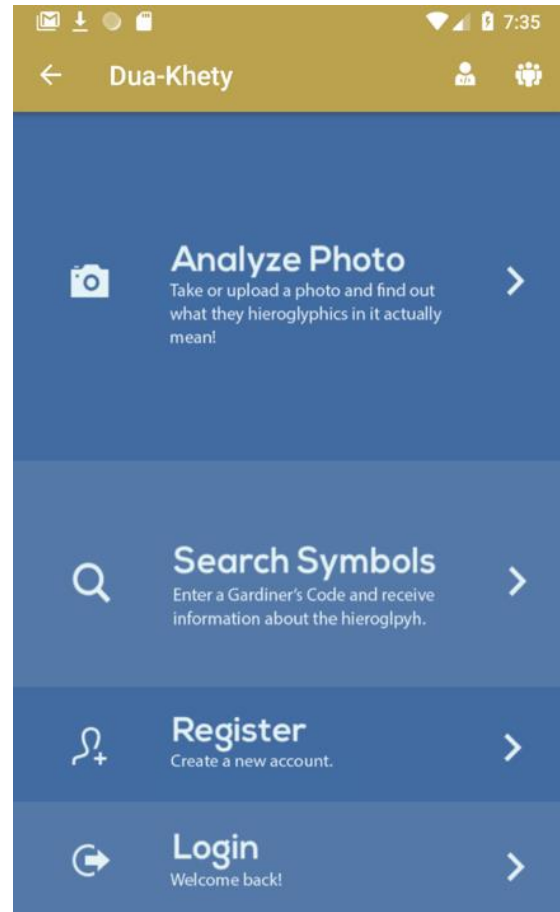


Register/Sign Up Screen



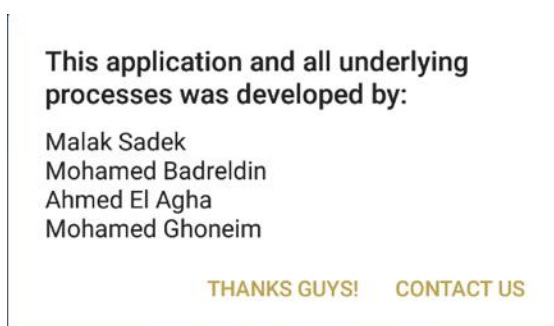
The Sign In screen features a gold header with a back arrow and the text "Sign In". Below the header, there are two input fields: "Email" and "Password", each with a gold underline. A blue "SUBMIT" button is positioned below the password field. At the bottom, there is a black navigation bar with three icons: a left arrow, a circle, and a square.

Sign In Screen



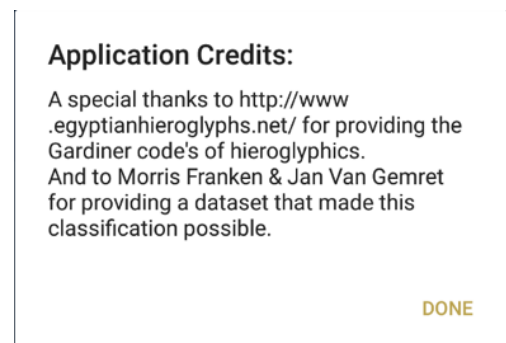
The One Time User Menu has a gold header with a back arrow, the text "Dua-Khety", and two user icons. The main content area is blue and contains four menu items, each with an icon, a title, a description, and a right arrow: "Analyze Photo" (camera icon), "Search Symbols" (magnifying glass icon), "Register" (person with plus icon), and "Login" (circular arrow icon). The time 7:35 is shown in the top right corner.

One Time User Menu



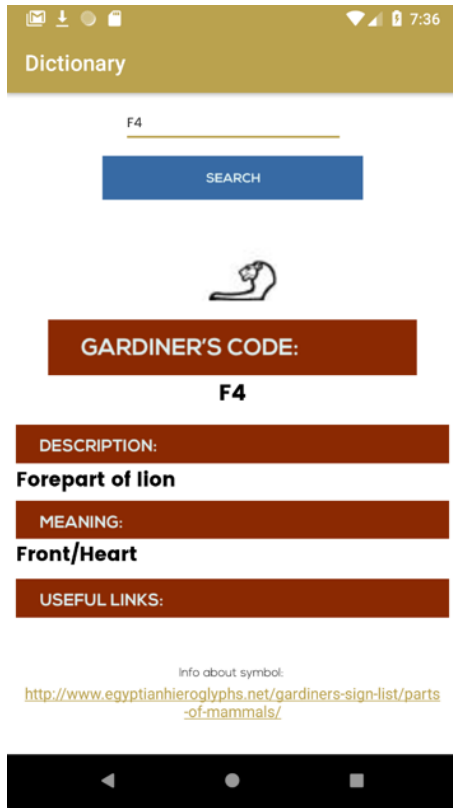
The Developers Acknowledgment screen has a white background with a thin black border. It contains the text "This application and all underlying processes was developed by:" followed by the names "Malak Sadek", "Mohamed Badreldin", "Ahmed El Agha", and "Mohamed Ghoneim". At the bottom, there are two gold buttons: "THANKS GUYS!" and "CONTACT US".

Developers Acknowledgment

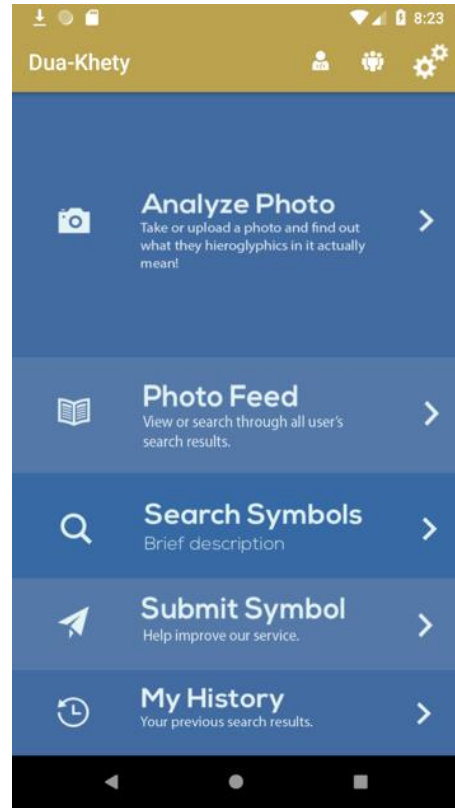


The Sources Acknowledgment screen has a white background with a thin black border. It features the heading "Application Credits:" followed by the text "A special thanks to <http://www.egyptianhieroglyphs.net/> for providing the Gardiner code's of hieroglyphics. And to Morris Franken & Jan Van Gemret for providing a dataset that made this classification possible." At the bottom right, there is a gold "DONE" button.

Sources Acknowledgment



Symbol Dictionary



Registered User Menu



Search History Screen

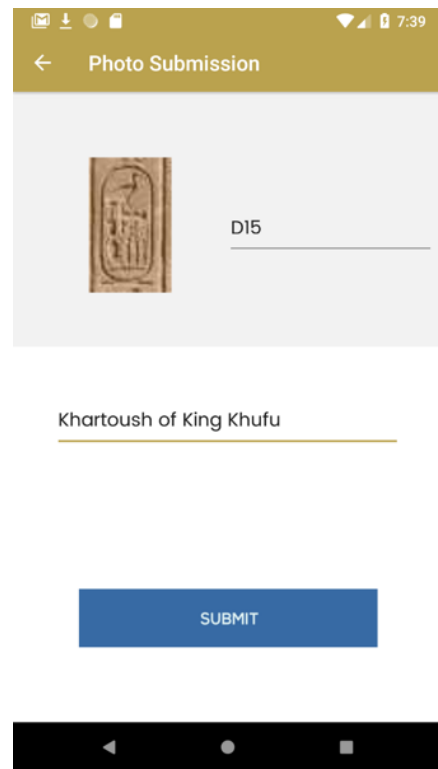
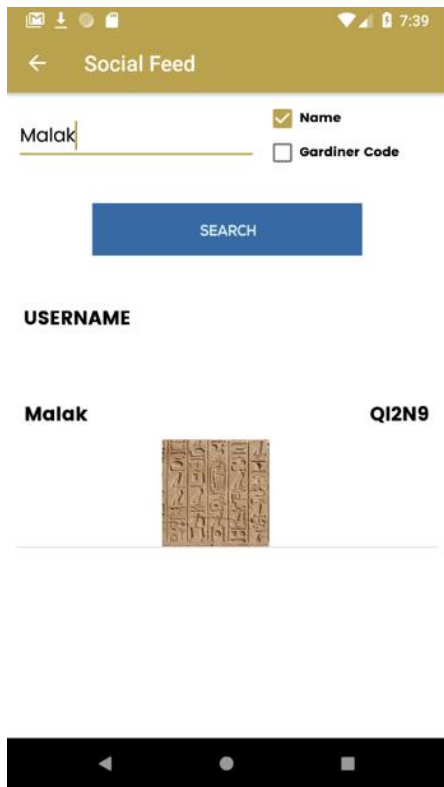
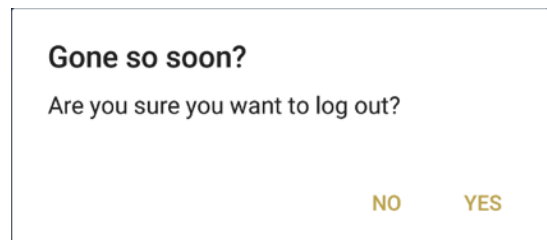


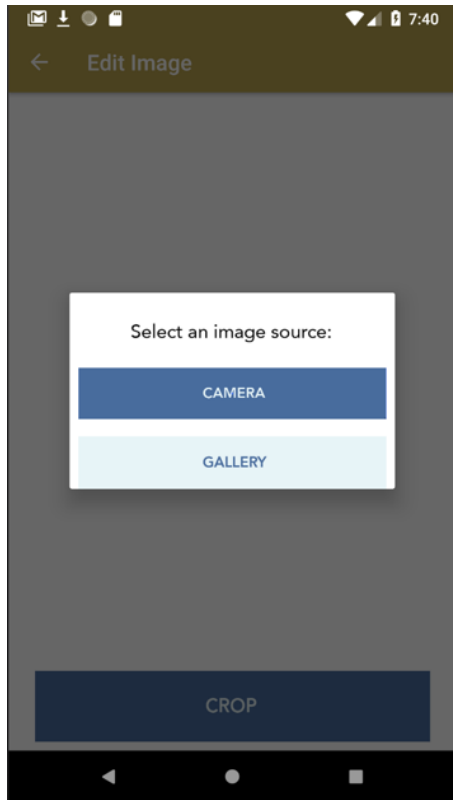
Photo Submission Screen



Social Feed Screen



Logout Dialog

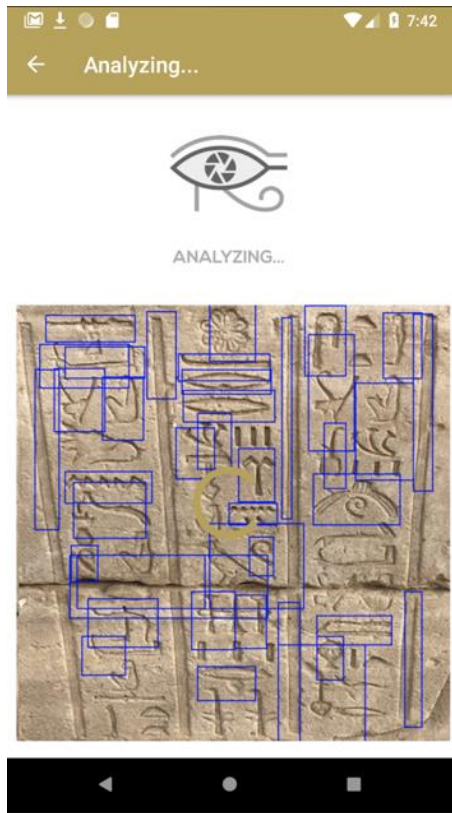


Select Image Source Dialog

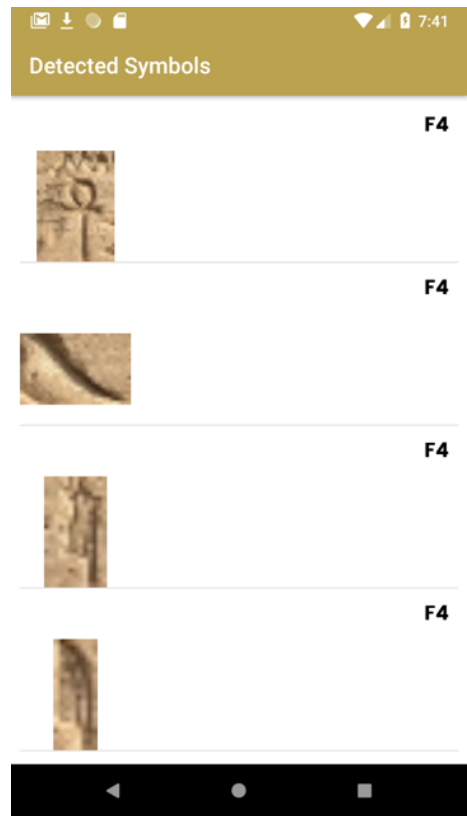


Crop Image Screen





Analyzing Image Screen



Detected Symbols Screen

### 5.3.2 iOS Application

22:05 2G LTE



**DUA-KHETY**

Explore the world of hieroglyphics.  
Let's get started!

REGISTER

LOGIN

SKIP

Opening/Start Up Screen

22:05 2G LTE

< REGISTER

**USERNAME**

Name

**EMAIL**

Email

**PASSWORD**

Password

**CONFIRM PASSWORD**

Reenter Password

SUBMIT

Register/Sign Up Screen

22:05 2G LTE

< LOGIN

EMAIL

PASSWORD

SUBMIT

Sign In Screen


22:04 2G LTE

< SYMBOL SEARCH

ENTER A SYMBOL

SEARCH

GARDINER'S CODE:

 G6

DESCRIPTION:

Falcon with flail

MEANING:

Falcon

USEFUL LINKS:

Info about symbol:  
<http://www.egyptianhieroglyphs.net/gardiners-sign-list/birds/>

Symbol Dictionary Screen

22:05 2G LTE

DUA-KHETY

Analyze Photo  
Take or upload a photo and find out what they hieroglyphics in it actually meant!

Search Symbols  
Enter a Gardiner's Code and receive information about the hieroglyph.

Register  
Create a new account.

Login  
Welcome back!

One Time User Menu

22:03 2G LTE

< HISTORY




CLEAR HISTORY

Search History Screen

22:04 2G LTE

< SUBMIT SYMBOL



GARDINER'S CODE

DESCRIPTION

SUBMIT

Photo Submission Screen



Cropping Screen

22:04 2G LTE


< PHOTO FEED

BY USERNAME BY CODE

SEARCH

RECENT PHOTOS

MALAK QL2N9



>

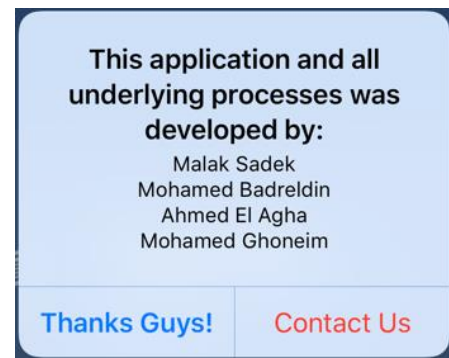
Social Feed Screen



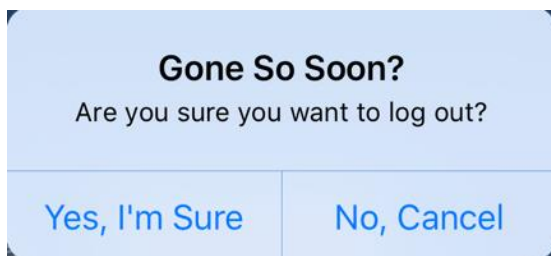
Analyzing Image Screen



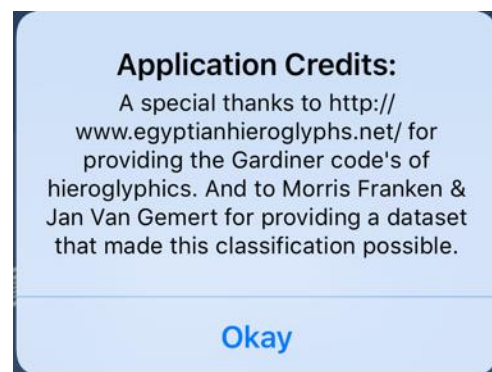
Detected Symbols Screen



Developers Acknowledgement



Logout Dialog



Sources Acknowledgement

## 6. Appendix A - Sequence Diagrams

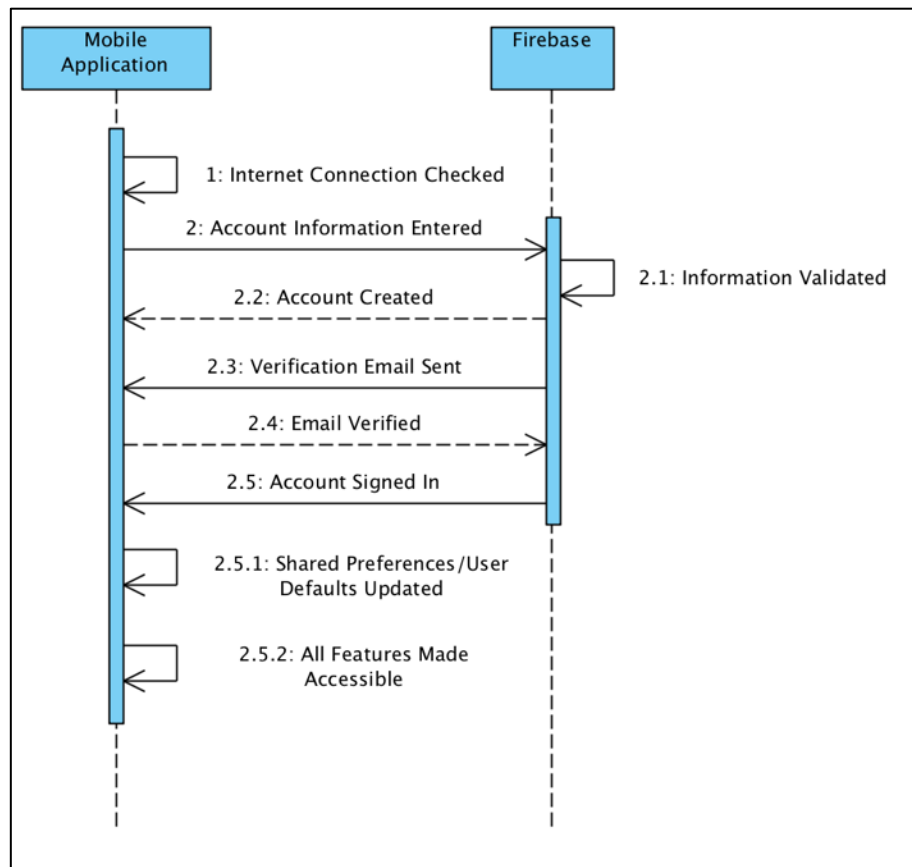


Figure 3 - User Signs Up For a New Account

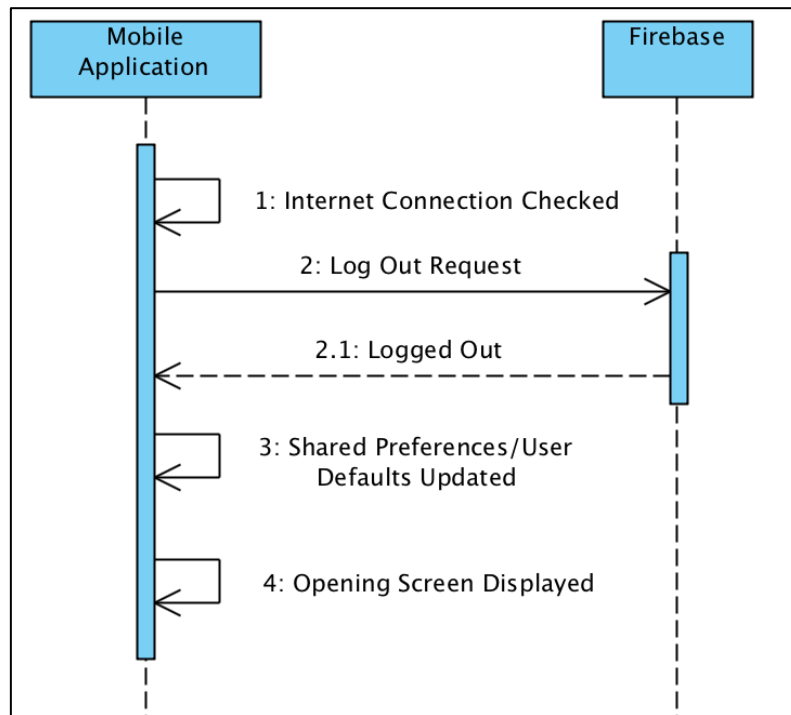


Figure 4 – User Logs Out of Current Account

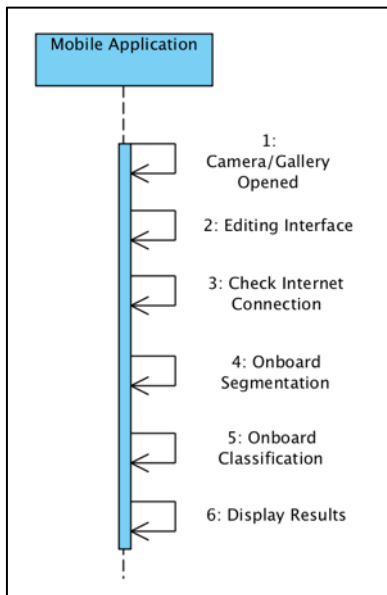


Figure 5 – User Taking/Choosing a Photo For Classification Without Internet

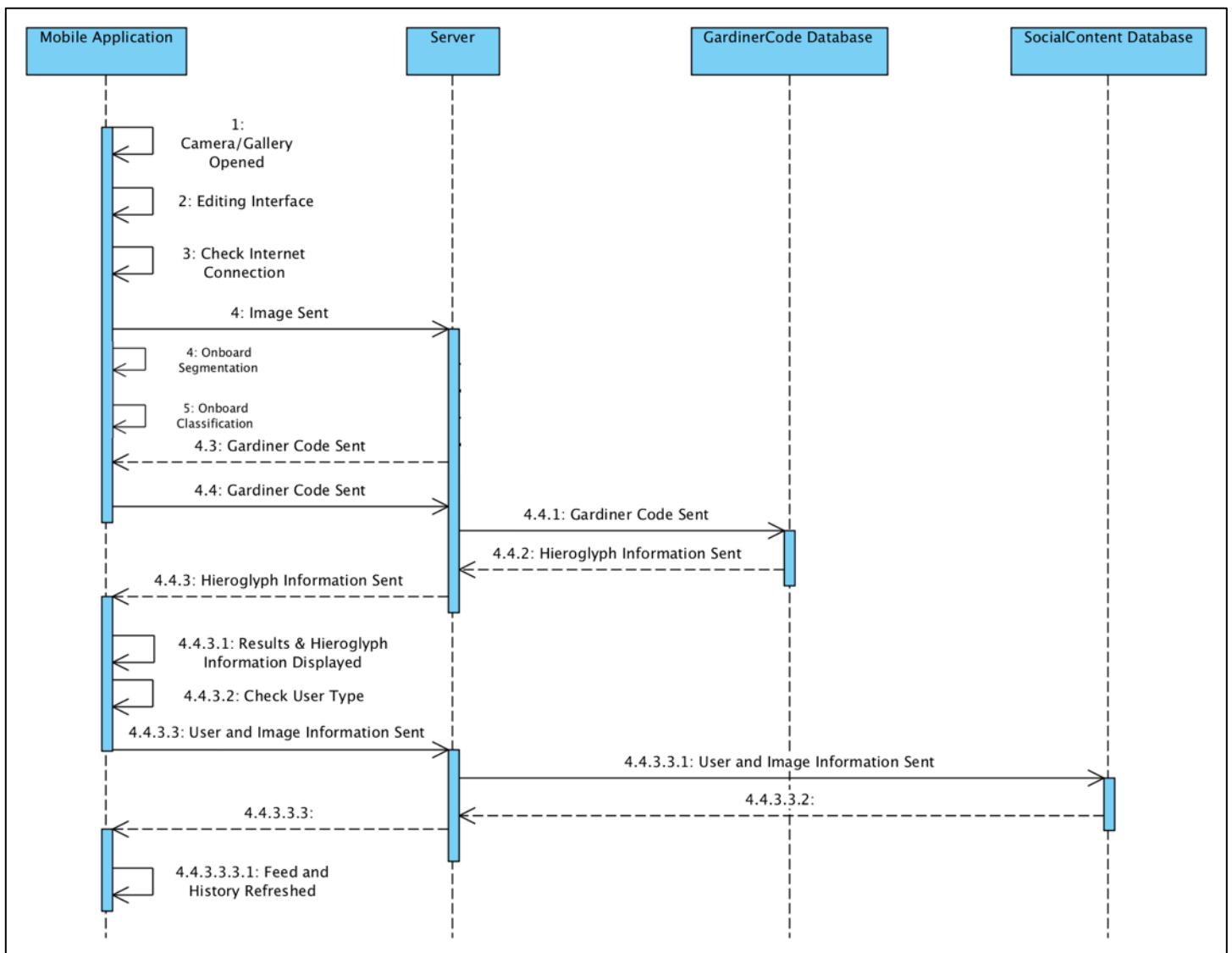


Figure 6 - User Taking/Choosing a Photo For Classification With Internet

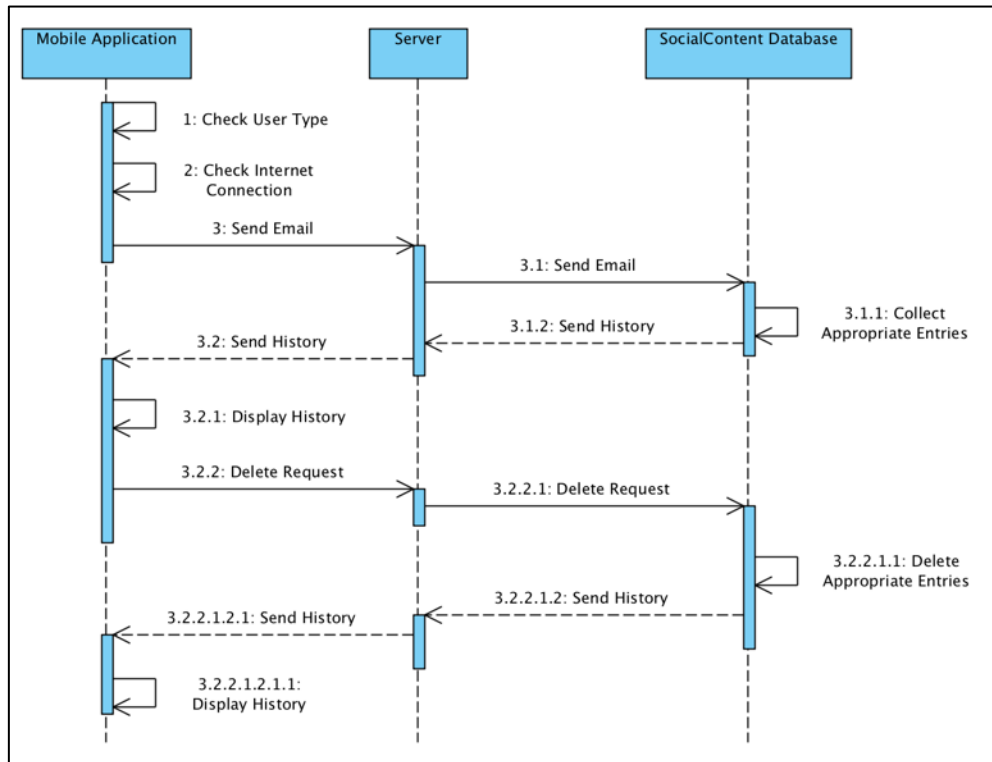


Figure 7 – User Opens Their History, Then Deletes an Item or Clears the History

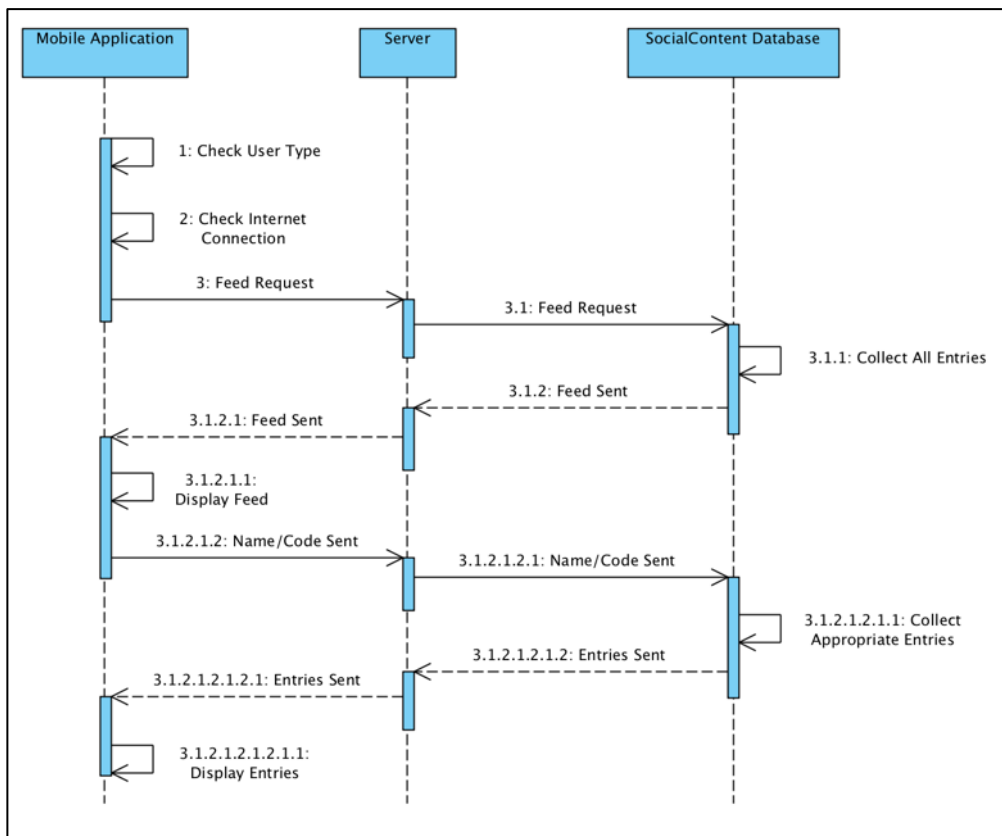


Figure 8 – User Opens Their Feed, Then Searches Using Gardiner's Code or Owner's Name



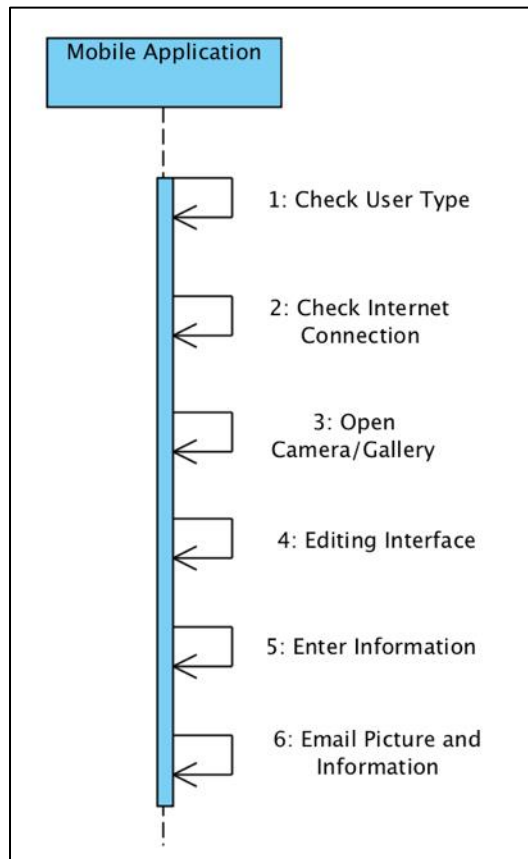


Figure 9 – User Submits a Photo to the Developers

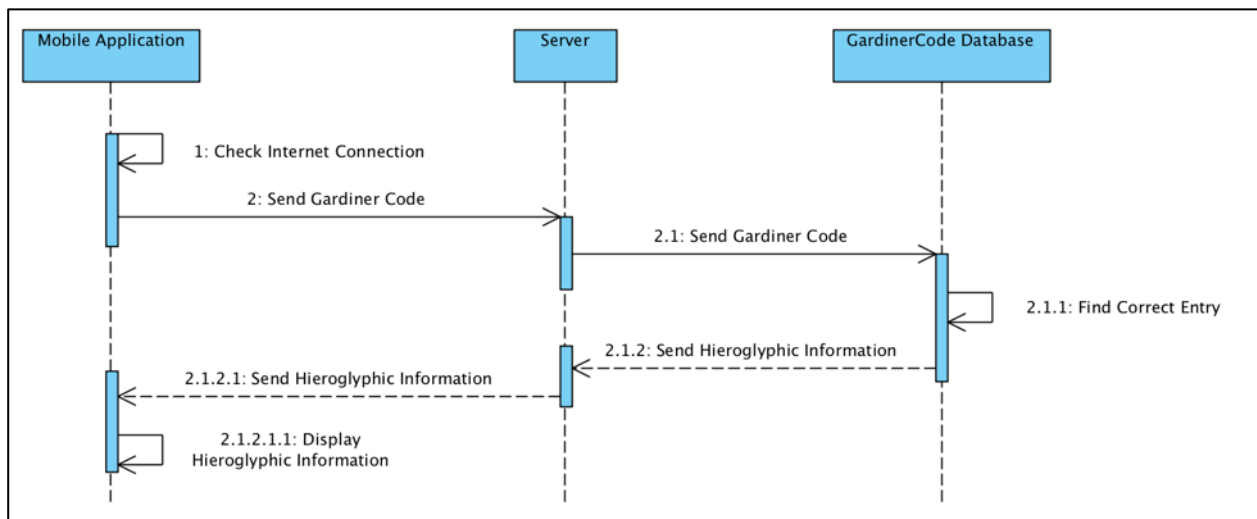


Figure 10 – User Searches For Information of a Hieroglyph Using Its Gardiner Code

## 7. Appendix B - Results

Overall, the application was completed in iOS, and mostly completed on Android. All extra features were fully implemented on both versions. The social feed needs to be enriched to include more social interactions such as sharing to other social media outlets, and liking or commenting on other posts, as well as a location sharing system for posts.

Segmentation was completed on both versions as well, however while it is sufficient, it does not detect every single symbol within an image, and on rare occasions, the segmented images have trimmed portions that stifle classification. Both these limitations need to be addressed, as well as being able to detect whether the hieroglyphics are written horizontally or vertically in order to provide more advanced transliteration in the future.

Classification began at 2% accuracy when using a deep layer network with 100+ layers, and using the raw images provided by Franken and Gemet. Binarizing the images increased the accuracy to 10% as the dataset was grayscale, while actual images taken from the application were colored. Augmenting the images increased accuracy by a further 3%. Using a smaller neural network increased the accuracy to 55% and increasing the kernel size to 30x30 reached an accuracy of 70% however it suffered from extreme overfitting.

Classification currently has a 66% accuracy when predicting the top match, and an 82% accuracy when predicting the top five expected hieroglyphs using a Siamese neural network. The dataset needs to be supplemented through more photos and further augmenting the existing ones in order to improve this accuracy and allow other types of classifiers requiring more extensive datasets to be experimented with. It would also be beneficial to involve an Egyptology for more advanced feature extraction as different classes within the Gardiner code system often have similar identifying features.