Moth Classifier Design Documentation

Ben Giangrasso, Reese Jones, Abid Ahmed

Overall Project Description

Ecologists classify species to observe their role in their respective ecosystems. Discover Life is an ecological organization that classifies images of insects. However, their current process of image classification requires the certification of multiple experts, and the entire process takes around 35 seconds per image. Currently, Discover Life has thousands of unclassified images of moths. Image classification from an autonomous system will expedite the process and will remove the need of humans to classify images by hand. Moth Classifier will classify the thousands of unclassified images from the Discover Life website using machine learning. These classifications will then be sent back to the Discover Life team. Additionally, Moth Classifier will allow both researchers and non-ecologist users to receive instant moth species classifications through a mobile application interface. Users will be able to submit pictures from their camera or image gallery, and the application will provide a species name using the same machine learning model that will be used to populate Discover Life's website with classifications. Unifying the system of picture-taking and image classification will benefit the ecologists of Discover Life since it will allow for faster results, and the classification process does not require human professionals for a prediction to be made.

Users and Use Cases

There are two main demographics of users for the Moth Classifier project. First there are biologists (from now on called the researchers), mostly those working with the Discover Life project, who are using the product with the goal of gathering more data points and refining data collection methods for their research of moth populations. The second, and significantly larger, class of users for the Moth Classifier project are the general public. This second class of users is focused on submitting new data points to the Discover Life project in order to benefit the insect research community. Both classes of users will interact with the project using the mobile application being developed using Google's Flutter framework, meaning that both classes of user will have access to the project on both Android and iOS devices. Although both classes of user interact with the same application the researcher and general user view will be different once they are logged in so as to benefit the specific use cases of each class of user.

For the researcher class of users, their use case involves actions related to the refinement of data analysis through our machine learning models. A researcher will be using the application that we are building in order to adjust classification of images previously submitted by either other researchers or by a general user. The classification adjustments submitted by researchers will be used to retrain our machine learning models so that they provide more accurate data analytics for the Discover Life project. Additionally a researcher will be able to submit images to the project for classifications, just as is the case for general users. The use case for a general user is much more limited in that they will only have the capability to submit images of moths to the mobile application and then reviewing the results of those classifications in the future. The general user will not be able to change any classifications or otherwise affect the data stored by Discover Life, they will only be allowed to contribute data points to the project.

Use Case: Obtain Researcher Account
In order for users to obtain an account with researcher privileges, they must first obtain a general account. From that point, there will be a button in their "Profile" screen where they will be able to request a researcher account by providing the nature of their research as well as information about which organization they do research with. These requests will be manually reviewed and account updates will be done by system administrators outside of the mobile application.

Use Case: Obtain General Account
In order to obtain a general account for the Moth Classifier project, users will have to download the mobile application on either an iPhone with iOS 9 or higher or an Android device with Android Lollipop or newer. Once downloaded, users will be able to register an account in three different ways. They will be able to register an account with their email and created password, but they will also be able to create an account using direct login from either Google Sign In or Sign in With Apple. Sign in with Apple is platform specific and will only be available to devices running iOS, but Google Sign In and email/password creation will be available to any device.

Use Case: General and Researcher Authentication
One a user, either general or researcher, has created an account with one of the methods specified in "Use Case: Obtain General Account" they will be able to login to the application through the "Login" screen. Depending on the method with which a user created their account, they will log in with one of three methods. If they have an email/password account created through the mobile application then they will simply login by inputting that information. If their account was created using Sign in with Apple, the user will be prompted by the device for a form of authentication (for Apple users that can be AppleId password, fingerprint, or FaceId depending on their specific device). If the account was created using Google Sign In, they will be prompted to undergo the normal login flow that a Google account generally follows (allowing for 2FA if they have it set up with their personal accounts).

Use Case: User Submits Photo for Classification
For both types of accounts, researcher and general, the process of submitting images for classification by the Moth Classifier system are the same. First, an authenticated user will navigate to the "Submit" tab within the mobile application. Then they will be able to view the proper guidelines for a "good" submission to the project. Next, the user will have to choose whether to upload a photo from their camera roll or to directly upload a picture by taking it at the time of submission using the device's camera. Once an image has been selected (or taken with the camera) the user will be able to view a thumbnail image representing the scaled original image, and they will have the option to either delete that image and choose a new one or to

submit the currently selected image. After an image has been uploaded, the user will be shown a dialog confirming their submission was successful as well as presenting them with the ID associated with their submitted image.

Use Case: User Views Their Previous Submissions
Once a user has previously submitted images to the Moth Classifier project for classification (see "Use Case: User Submits Photo for Classification"), they will be able to view their submissions on the "Profile" page. On this page, in addition to their general user-specific data, there is a scrollable list of previous submissions to the project. Nothing will be able to be edited or changed about previously submitted information there, but users will still be able to view all of their previous submissions.

Use Case: Researcher Submits Adjusted Classification
Once authenticated, a researcher will be able to navigate to the "Reclassify" screen within the mobile application. From that point they will be presented with a scrollable list of images which have been determined by the system as needing review after their original classification by the system. From that list, the researcher will be able to select one image to view details about it (such as the previously assigned classification as well as the confidence rating which that classification had). From that screen, researchers will be able to use two dropdown windows to specify the exact species of moth in the image (or denote the image as not a moth at all), or create a new species entry if the species they wish to assign does not already exist within the system. Once the researcher has selected the proper information to include in their reclassification and submits it, that data will be updated within the database via the API and the user who submitted the image will be notified of a change in status to their submission.

Use Case: User Checks Status of Their Submissions
Once a user has submitted images to the Moth Classifier system for classification (using the steps in "Use Case: User Submits Photo for Classification"), they will be able to navigate to the "Notifications" page within the mobile application. There, the user will be able to see a list view of all previous submissions that they have made to the project as well as the current status of each submission. The valid statuses that they will be able to see are as follows: Processing, Classified, and Needs Review. No information can be edited or otherwise updated from this point, and can only be viewed by the user.

Minor Use Cases
The following additional use cases are currently supported by the Moth Classifier system:

- Both classes of user can choose to logout at any point, at which time their "automatic saved login" information will be deleted and they will have to re-authenticate in order to access the application.

- Users who login with third party tools (Google Sign In and Sign in with Apple) will have their credentials automatically stored for future logins without needing to supply their credentials directly.

Future Additions

The system is being designed in such a way that will allow for these additional features to potentially accommodate additional use cases by biologists at Discover Life.

- Creation of an Administrative panel on the Web so that Research teams can monitor data and statistics about image submissions over time.
- Addition of an option within the mobile application to store the most recent versions of the machine learning models on the device. This will allow for classification without an internet connection.

General System Design

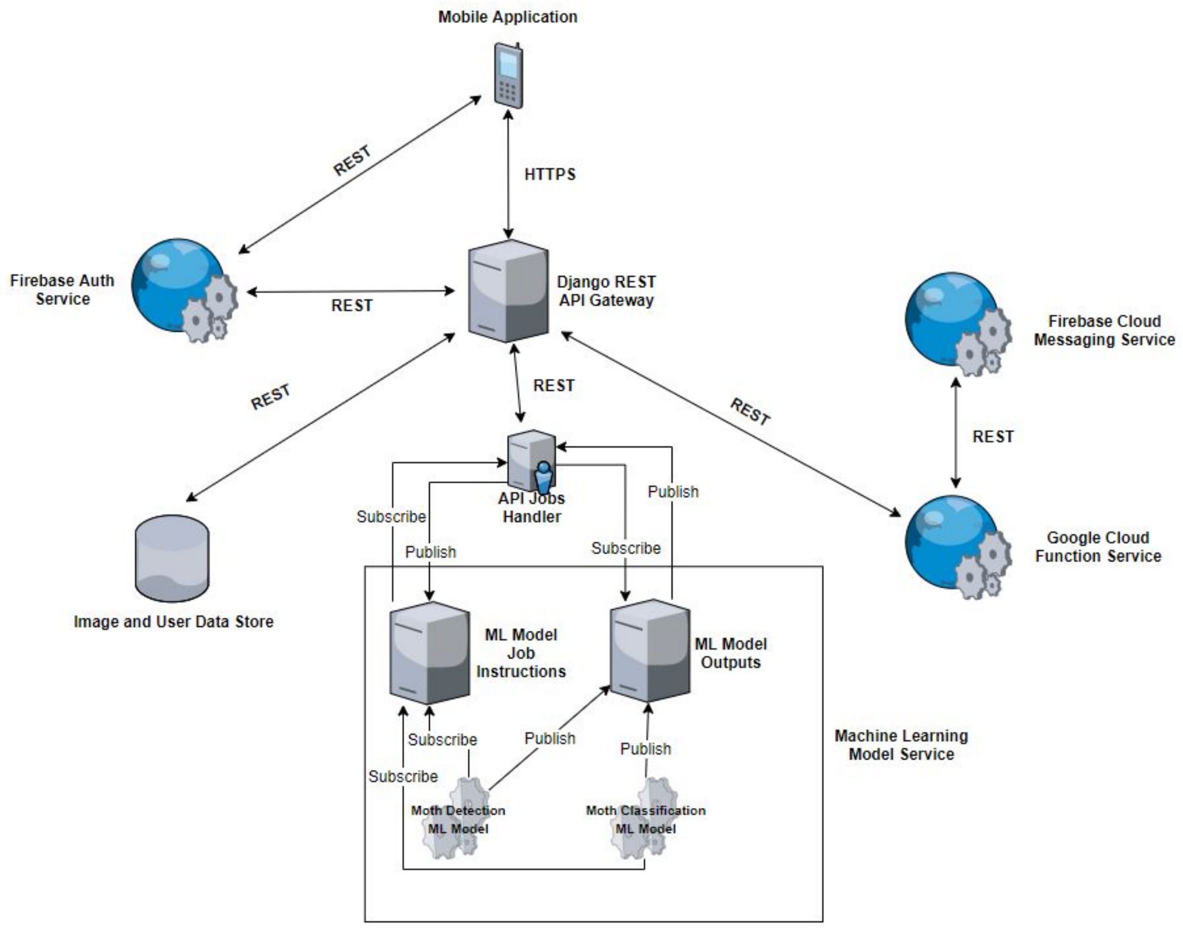

Illustration 1: Server-Side Design Architecture of Moth Classifier project
(link to original image)

The Moth Classifier system is designed using a Microservices architecture for the server-side systems. The above illustration (Illustration 1) displays all of the main interactions between different services. In our design, the main Django REST API functions as a gateway and a coordinator between the different services utilized to provide functionality to the rest of the project. Communication between the mobile application and the Rest API which is facilitating interactions with services is done using RESTful calls protected by HTTPS. Each of the server-side components is designed to run on a VPS hosted on the Digitalocean cloud.

Generally users will submit image data (both classifications and submissions) through the mobile application, which will then be sent over to the API through HTTP requests. From there, the API will publish images, as well as the "jobs" associated with their classification, to the machine learning service. Within the machine learning service, the actual component running the machine learning tasks will subscribe to the job handler within the service, and its output will be consumed by the job handler. The job handler will distribute and consume all data relating to the actual classification and identification of the different machine learning models, as well as informing the ML service when to retrain and what information to retrain with. The only information which will be directly accessible to Researchers and general user accounts is the information which has been updated within the User and Image data store by the Django API and exposed through the mobile app interface.

Additionally, when the API gateway receives updated information regarding the conclusion of a classification or identification job, it will send out a REST call to a Google Cloud Function that is responsible for triggering Firebase Cloud Messaging service for asynchronous notifications. Another form of asynchronous communication used throughout the design are calls to the Firebase authentication service from both the mobile app directly and through the API gateway to enforce security.

As indicated by Illustration 1, all components will interface their requests through the REST API, which will be responsible for coordinating jobs to the Machine Learning service, the Google Cloud Functions, and the Firebase Authentication service. The machine learning service will interface with the API's job handler using a Pub/Sub pattern facilitated by the Redis message broker which will sit between the two services.

Major Components

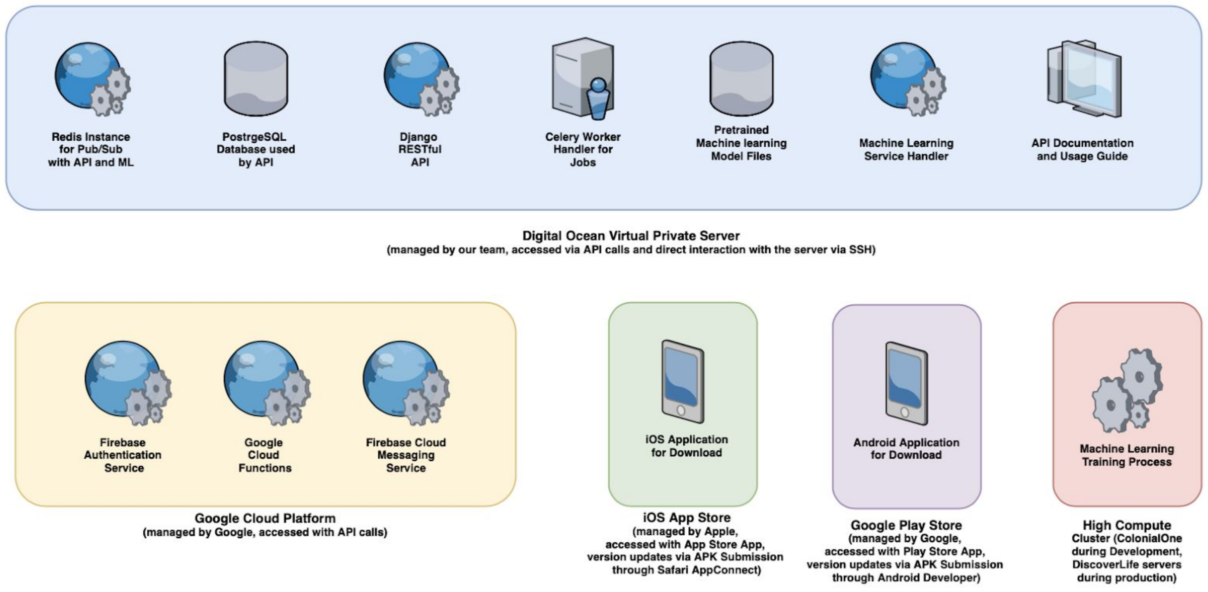
Redis Instance for Pub/Sub with API and ML
PostrgeSQL Database used by API
Django RESTful API
Celery Worker Handler for Jobs
Pretrained Machine learning Model Files
Machine Learning Service Handler
API Documentation and Usage Guide
Digital Ocean Virtual Private Server
(managed by our team, accessed via API calls and direct interaction with the server via SSH)
Firebase Authentication Service
Google Cloud Functions
Firebase Cloud Messaging Service
Google Cloud Platform
(managed by Google, accessed with API calls)
iOS Application for Download
iOS App Store
(managed by Apple, accessed with App Store App, version updates via APK Submission through Safari AppConnect)
Android Application for Download
Google Play Store
(managed by Google, accessed with Play Store App, version updates via APK Submission through Android Developer)
Machine Learning Training Process
High Compute Cluster (ColonialOne during Development, DiscoverLife servers during production)

Illustration 1. Major System Components and their Containers in the Cloud
(link to Original Image)

As shown in the diagram above (Illustration 1), there are thirteen major components of the Moth Classifier project, and together they run, or will run, in five separate containers. The five major containers which components of the project run in are as follows: a Digital Ocean Virtual Private Server, the Google Cloud Platform, the iOS App Store/an iPhone, the Google Play Store/an Android phone, and a High Compute Cluster. The first, and largest, container of different components within the project is the Digital Ocean VPS which houses seven of the thirteen total components of the Moth Classifier project. The components housed within the VPS which our team will manage directly are as follows: the Django RESTful API, a Redis instance, a PostgreSQL database, API documentation, a Celery based task queue, the machine learning service handler, and pretrained machine learning model files.

The Django API has a number of functional requirements as it relates to the system as a whole because the API serves as the main method of communication between the front end, back end, and other web services used by the project. First, the API must be able to facilitate the creation of data records by the mobile application. This can come in the form of creation of user records or submission of images and their classifications within the mobile application, and the API must be able to handle a number of requests at one time to permit multiple users of our product at once. Second, the API must be able to manipulate data to fit pre-defined formats and store it within the Postgres relational database which houses all of the information regarding images, their classifications, and users of the mobile application. Additionally, the API must be able to receive image file uploads via a POST request as well as move those images around a folder structure which represents the varying levels of image classification. As an example when an image is just submitted it should be stored under /images/unknown/{img_id}.jpg, and once the image is determined to be a moth it should be moved to /images/moths/{species}/{img_id}.jpg. Another functional requirement of the API is that it should be able to use the Celery worker framework to coordinate "jobs" for classification, identification, and other tasks relating to the training and usage of the machine learning models which we are developing. Also, the API should provide the functionality to download images via a URL, whether that is individual images to be used one at a time or entire batches of images to be used as a large training set of data for the machine learning algorithm. The final major functional requirement of the API is that it should be able to facilitate the usage of asynchronous notifications to different users of the mobile application using HTTP calls to Google Cloud Functions and the Firebase Cloud Messenger services.

In addition to the functional requirements of our API, there are a number of non-functional requirements which help the system overall meet the expectations of the Discover Life project. First, the API should be able to have the capacity to coordinate a large number of "jobs" for the machine learning service, allowing for more concurrent tasks to occur at any given point in time. The API should also be secure across all of its endpoints so as to prevent unintended use of the services it provides by users who have not been explicitly permitted to do so. As an example, only administrators should have access to the master list of users or jobs currently active on the system, and User X should only have access to information directly relating to their submissions and profile information. Additionally, the API should utilize implementation practices which allow for scalability, something which is being done through the current use of the Celery job framework and Redis pub/sub communication with the machine learning service. A final non-functional requirement of the API is that it should be efficient with storage and manipulation of data in order to limit the total number of database writes and external API calls which need to be made during normal operation of the system.

The next container that the Moth Classifier project takes advantage of is the Google Cloud Platform, which hosts and manages our instances of Firebase Authentication, Google Cloud Functions, and Firebase Cloud Messaging. All of the services which the project utilizes that are hosted by Google Cloud Platform are utilized using publicly exposed API endpoints. The functional requirements of each component of the Google Cloud are narrow, but each service was chosen for its ability to meet a specific functional requirement of our overall system. There are three main functional requirements of the Firebase Authentication service. First, the Firebase Authentication service must be able to do CRUD operations on user accounts using API calls made from within the mobile application. Second, the Firebase Authentication API must support authentication verification using tokens so that the Moth Classifier API can verify user authentication information submitted by the mobile application without needing direct access to a user's inputted credentials. Lastly, the Firebase Authentication service has to be able to support login using a third party system (i.e. Google Sign On and Sign In with Apple) so that a user can use an account which already exists if that is what they would prefer. There are only two main functional requirements of the Google Cloud Functions that the project will be implementing for the production-ready product. First, the Cloud Function should be able to be triggered using an API call *or* by observing a database path within the Firebase Cloud Notification datastore and triggering every time the store of notifications is updated. As an example, if the API makes a write to the FCM path /notifications/{user_id}/{noitification_id}, then the Cloud Function should trigger a notification to any device associated with {user_id} and it should format and publish the notification with id: {notification_id}. The second functional requirement for the project's Cloud Functions is that they should be able to interact with the Firebase Cloud Messaging API in order to generate and distribute asynchronous notifications. The Firebase Cloud Messaging service has one main functional requirement as it relates to the function of the Moth Classifier project. For the purpose of this project, the Firebase Cloud Messenger must have the capability to send asynchronous notifications both to individual users and to large groups of users (referred to as topics) to facilitate the exchange of information about classification job status between the API and individual users.

Although the different services from Google Cloud Platform that the Moth Classifier project will take advantage of have differing functional requirements, many of their non-functional requirements are similar in nature. All of the services which the project uses share the non-functional requirement of continuing to function efficiently at scale. That was one of the major reasons that the team decided to utilize the Google Cloud Platform, as their services scale automatically and the only thing which changes is the cost. The Google Cloud Platform group of services will automatically scale in a performant way such that the services will always be available and will only actually scale if demand on each service is high enough to cause an impact to performance, a contrast to our VPS which we would have to scale manually. Additionally, the GCP services being used for the project have the non-functional requirement of needing to be available as close to 100% of the time as possible. This is another benefit of using the GCP in that their services achieve almost perfect uptime with no actual monitoring or maintenance needed by our team directly, and there are ways to set up automated notifications should there be any impact to the uptime of any of the services we use. Overall the suite of tools which GCP provides allows for efficient scalability and as close to 24/7/365 availability as possible with little to no maintenance overhead, which is important to a small team like ours with limited personnel resources.

The third major component of the Moth Classifier project is the mobile application built using Google's Flutter framework. The "containers" that the mobile application can run in are any cellular devices running either Android or iOS. Due to the fact that the functional and non-functional requirements of the application do not differ on either platform they will from this point on be referred to as if there is only one container in which the mobile application can run on. The mobile application has five major functional requirements, the first of which is the ability for users to submit images to the system to be classified automatically using the team's machine learning algorithms. Secondly, the mobile application also has to provide the ability for privileged "researcher" users to adjust classification data of incorrectly classified images in order to improve the pool of accurate data used for refining our machine learning models. The mobile application also has to be able to handle asynchronous notifications regarding status updates for a user's submitted images. This has to be done in such a way that the application can properly load and display notifications that are received when the application is running in the foreground or background, as well as when the application is closed entirely. Additionally, the application should support displaying the same notifications on every device that a user is logged into. The final functional requirement of the mobile application is the ability to create and manage user accounts from within the application itself without the user needing to use additional tools or be signed up by an administrator.

There are three main non-functional requirements of the mobile application, and each of them is intended to increase the ease of use of our product for all users. First, the application needs to be able to render screen information with as few network calls as possible in order to eliminate a potential bottleneck from a slow internet connection. This is achieved through the use of Dart's "futures" as a method for asynchronous data gathering and storage across sessions of the application. Additionally, the application should be available as close to 24/7 as possible so that access to the service is available to as many people as possible regardless of location. This is being achieved through implementation of request caching in lieu of actually making HTTP requests when the device has no internet connection. That way the user experience will not change regardless of whether or not they have an internet connection. The final non-functional requirement of the mobile application is that accounts registered within the app must be easily transferable to another device, something being facilitated by the usage of Firebase Authentication service for login and registration.

In the development stage of the machine learning algorithm, each model will be retrained with varying hyperparameter tunes and different datasets. When building a machine learning model that is trained with thousands of images, the calculations are time consuming when done locally. Instead, each machine learning model will be built on a GPU instance. Our machine learning code will use Colonial One as its high-performance computing container. A GPU instance will allow the models to be built with the assistance of hundreds of computing cores. This is beneficial to the development stage because it allows machine learning models to be deployed at a faster rate, and more models can be built with the time gained from computing each one virtually. After the development phase of the project concludes, the retraining of the deployed machine learning model will be computed on Discover Life's personal high-performance container. This container will be used to retrain the model with new pictures indefinitely.

How System Components Interact

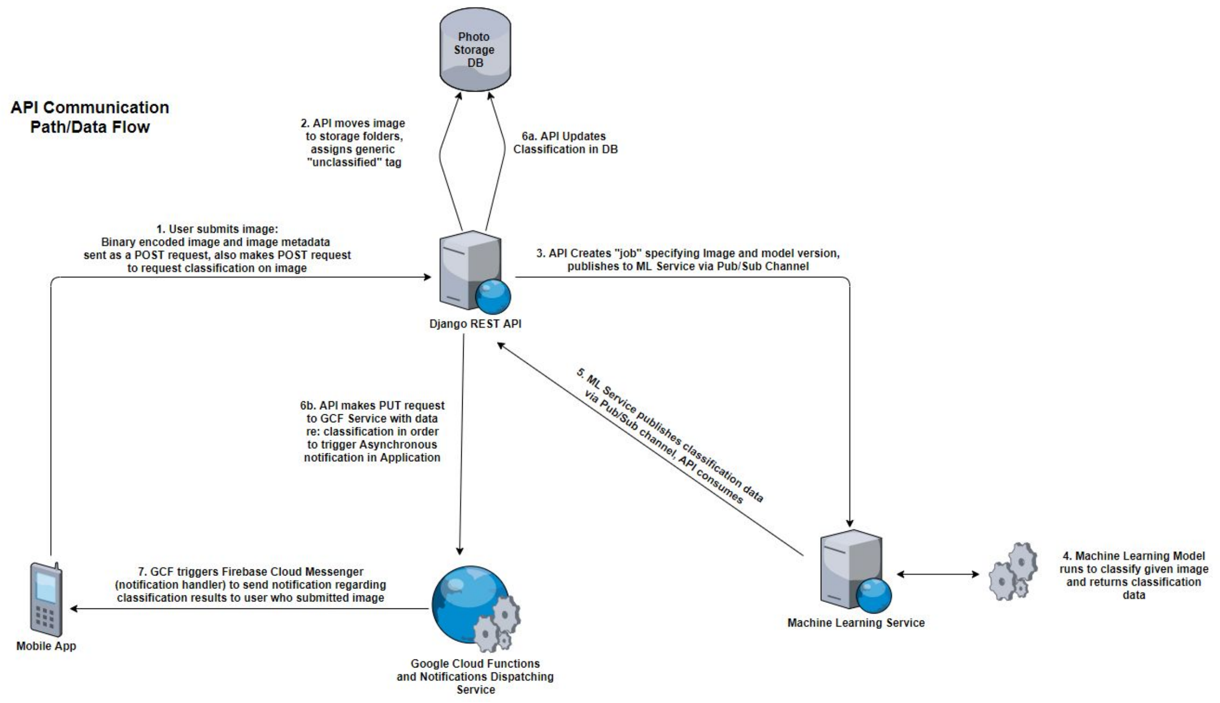
API Communication Path/Data Flow
Photo Storage DB
2. API moves image to storage folders, assigns generic "unclassified" tag
6a. API Updates Classification in DB
1. User submits image: Binary encoded image and image metadata sent as a POST request, also makes POST request to request classification on image
Django REST API
3. API Creates "job" specifying Image and model version, publishes to ML Service via Pub/Sub Channel
6b. API makes PUT request to GCF Service with data re: classification in order to trigger Asynchronous notification in Application
5. ML Service publishes classification data via Pub/Sub channel, API consumes
Machine Learning Service
4. Machine Learning Model runs to classify given image and returns classification data
7. GCF triggers Firebase Cloud Messenger (notification handler) to send notification regarding classification results to user who submitted image
Mobile App
Google Cloud Functions and Notifications Dispatching Service

Illustration 2. Moth Classifier Project Data Flow Example
(link to Original Image)

Overview of API Requests

To retrieve and manipulate the data collected by the Moth Classifier system, the mobile application communicates with the Moth Classifier API. To do so, the mobile application will make HTTP requests to the Moth Classifier API. The API supports GET, POST, PUT, PATCH, and DELETE HTTP requests. These types of requests are enough to fulfill the needs of the mobile application and the entire Moth Classifier system. A full description of each endpoint is listed later in the document. Once a valid HTTP request has been made, the API executes the code in the Django view associated with the HTTP request. A Django view holds the business logic relating to the HTTP request and can make calls to other services. For example, a view relating to the jobs endpoint can call the service that handles creating and scheduling a classification job. Once the HTTP request has been handled by the view, an HTTP response is returned to the mobile application. The mobile application mainly cares about the HTTP response code and the JSON data returned. The mobile application can use the HTTP response code to make sure the HTTP request and related logic was successful, and the application can parse the JSON data for its needs.

Redis Publish/Subscribe

The next major component interaction in the Moth Classifier system lies in the communication between the Moth Classifier API and the machine learning service. To do so, the Moth Classifier system uses two pub/sub channels, using Redis as the message broker, between the API and machine learning service. The first channel, named “job_channel” in our code, is responsible for sending data from the API to the machine learning service. With this channel, the API sends information relating to a job. Thus, it sends the job id, image id, and the machine learning model to use. The machine learning service reads the messages from this channel and classifies the specified image using the specified model.

The second channel, named “ml_channel”, is responsible for sending data from the machine learning service to the API. Once the machine learning service receives a job and classifies the corresponding image, it publishes the classification (species and accuracy) and job id to this channel. The API (in a separate listener process called ml_susbcribe.py) reads the messages from this channel. After that the API will update the job’s status to finished and will update the associated classification for that image. Finally, the API will send a push notification to the mobile application to notify the user that the job is done and that the image has been classified.

Authentication and Access Control
To ensure the security of our system and to prevent unauthorized access/modifications to important data, we are using Firebase Authentication to authenticate and validate the users using our system. Through the mobile application, users can log in using a standard email/password sign in, Google sign in, or Apple sign in. Regardless of the method they use, these credentials will be sent to the Firebase Authentication service. There the service will validate the user’s credentials. If valid, the authentication service will return an authentication token to the mobile application. This authentication token is necessary to make HTTP requests to the Moth Classifier API. When the mobile application calls the API, it will send that authentication token in the header of the request as a Bearer token. Once the API receives this token, it will contact the Firebase Authentication service to validate the token. If the token is valid, the API will execute the request. Otherwise, the API will return a permission denied or invalid token response back to the mobile application. With this workflow, the API only cares about receiving a valid authentication token. The mobile application is responsible for logging in and signing up users.

With the authentication scheme in place, the API has access control protocols in place to provide certain users with certain functionalities. The Moth Classifier API breaks users down into four different groups. The groups are anonymous users, standard users, researchers, and administrators. Each HTTP request, specifically each HTTP request with a specific HTTP verb, has a permission class associated with it. For example, only administrators can access the list of all users in the system. All other users trying to access this endpoint will receive a permission denied response. The API defines these permission classes and the Django REST framework handles the access control. The permissions for each endpoint are listed later in the document.

Push Notifications

To ensure that users of the mobile application know when a job has been completed or errored, notifications need to be sent to the mobile application. Thus, we will use the Google Cloud Platform and its cloud functions to build and push notifications through Firebase and into the mobile application. Once the API closes out a job either through the finish() or error() method in the Job class, the API will call the cloud function with the information needed for the notification. The cloud function will then build the notification with the information from the API. Then, the cloud function will contact Firebase. Firebase will then associate the notification with the user who submitted the job, and it will push the notification to the user's device(s).

Server Side (API) Interface

The Moth Classifier API is broken up into five different major endpoints that are responsible for executing requests. Each category of endpoints revolve around specific database models that store information vital to the entire system. The endpoints are as follows:

- users - Handles the logic relating to users of the Moth Classifier API
  - users/, users/{uid}/
- images - Handles the logic relating to the images stored in the Moth Classifier API
  - images/, images/download/, images/{id}/, users/{uid}/images/
- jobs - Handles the logic relating to jobs performed by the Moth Classifier API
  - jobs/, jobs/{id}/, images/{id}/jobs/
- classifications - Handles the logic relating to the classifications in the Moth Classifier API
  - classifications/, images/{id}/classification/
- models - Handles the logic relating to the machine learning models utilized by the Moth Classifier API
  - models/, models/{id}/, jobs/{id}/models/

The API root is located at /api/v1/

Retrieving Users

This function will be accessible via the endpoint at users/ using the GET verb. Only administrators can access this. It returns a paginated list of JSON objects. Each JSON object has the following fields:

- id
- uid
- url
- email
- first_name
- last_name
- date_joined
- last_login
- is_researcher
- is_staff

- is_active
- images (uri)

Viewing a User
This function will be accessible via the endpoint at users/{uid}/ using the GET verb. Only administrators or the specified user can access this. It returns a JSON object with the same fields as above (Retrieving Users).

Updating a User
This function will be accessible via the endpoint at users/{uid}/ using the PUT or PATCH verb. Only administrators or the specified user can access this. It takes a JSON object with the following fields (fields are optional if they are not being updated):
- first_name
- last_name
- is_researcher

This function returns an indicator of success or failure using standard HTTP response codes. On success, it will also return a JSON object containing the updated information. The fields returned are the same fields as above (Retrieving Users). In addition, on success, the User model in the database will be updated accordingly.

Deleting a User
This function will be accessible via the endpoint at users/{uid}/ using the DELETE verb. Only administrators can access this. This function returns an indicator of success or failure using standard HTTP response codes. On success, it will change the is_active field to false. It will not delete the user from the database.

Retrieving Images
This function will be accessible via the endpoint at images/ using the GET verb. Anyone can access this endpoint. It returns a paginated list of JSON objects. Each JSON object has the following fields:
- id
- url
- user
- file
- country
- region (same as a state if in the United States)
- county
- city
- zip_code
- street
- lat
- lng
- date_taken

- width
- height
- is_training
- hash
- jobs (uri)
- classification (uri)

Downloading Images
This function will be accessible via the endpoint at images/download/ using the GET verb. Anyone can access this endpoint. It returns a ZIP file of the images in the system. The ZIP file contains a csv file called metadata.csv. This holds pertinent metadata about the images returned by the endpoint. The ZIP file also holds an images folder that contains all the images in the ZIP file from the system.

Viewing an Image
This function will be accessible via the endpoint at images/{id}/ using the GET verb. Anyone can access this. It returns a JSON object containing the same fields as above (Retrieving Images).

Deleting an Image
This function will be accessible via the endpoint at images/{id}/ using the DELETE verb. Only an administrator or the owner of the image can access this. This function returns an indicator of success or failure using standard HTTP response codes. On success, it will remove the image from the database and remove the image from disk.

Retrieving Images by User
This function will be accessible via the endpoint at users/{uid}/images/ using the GET verb. Only an administrator or the specified user can access this. It returns a paginated list of JSON objects containing the same fields as above (Retrieving Images). In this endpoint, the image list is filtered and only includes images owned by the specified user.

Submitting an Image
This function will be accessible via the endpoint at users/{uid}/images/ using the POST verb. Only an administrator or the specified user can access this. It takes a JSON object with the following fields:

- file (required)
- country
- region
- county
- city

- zip_code
- street
- lat
- lng
- width
- height
- is_training

This function returns an indicator of success or failure using standard HTTP response codes. On success, it will also return a JSON object containing the new image. The fields returned are the same fields as in the Retrieving Images endpoint. In addition, on success, the image will be added to the database and saved to disk. A blank classification will be created for the image.

Retrieving Jobs
This function will be accessible via the endpoint at jobs/ using the GET verb. Only an administrator can access this. It returns a paginated list of JSON objects, and each JSON object has the following fields:

- id
- url
- job_type
- image
- date_issued
- last_modified
- status
- status_message
- models (uri)

Viewing a Job
This function will be accessible via the endpoint at jobs/{id}/ using the GET verb. Only an administrator or the user who submitted the job can access this. It returns a JSON object containing the same fields as above (Retrieving Jobs).

Retrieving Jobs by Image
This function will be accessible via the endpoint at images/{id}/jobs/ using the GET verb. Only an administrator or the owner of the image can access this. It returns a paginated list of JSON objects containing the same fields as above (Retrieving Jobs). In this endpoint, the jobs list is filtered and only includes jobs for the specified image.

Submitting a Job
This function will be accessible via the endpoint at images/{id}/jobs/ using the POST verb. Only an administrator or the owner of the image can access this. It takes a JSON object with the following fields:

- job_type

This function returns an indicator of success or failure using standard HTTP response codes. On success, it will also return a JSON object containing the new job. The fields returned are the same fields as in the Retrieving Jobs endpoint. In addition, on success, the job will be saved to the database, and will be sent to the Celery task queue to be executed with the process described in Redis Publish/Subscribe.

Retrieving Classifications
This function will be accessible via the endpoint at classifications/ using the GET verb. Anyone can access this. It returns a paginated list of JSON objects, and each JSON object has the following fields:

- url
- species
- accuracy
- image (uri)
- is_automated
- needs_review

Viewing a Classification
This function will be accessible via the endpoint at images/{id}/classification/ using the GET verb. Anyone can access this. It returns a JSON object with the same fields as above (Retrieving Classifications).

Updating a Classification
This function will be accessible via the endpoint at images/{id}/classification using the PUT or PATCH verbs. Only administrators or researchers can access this. It takes a JSON object with the following fields (fields are optional if they are not being updated):

- species
- accuracy
- is_automated

This function returns an indicator of success or failure using standard HTTP response codes. On success, it will also return a JSON object containing the updated information. The fields returned are the same fields as above (Retrieving Classifications). In addition, on success, the Classification model in the database will be updated accordingly.

Retrieving Machine Learning Models
This function will be accessible via the endpoint at models/ using the GET verb. Only administrators can access this. It returns a paginated list of JSON objects, and each JSON object has the following fields:

- id
- url
- name
- file_name
- date_created
- model_type
- rating
- comments

Submitting a Machine Learning Model
This function will be accessible via the endpoint at models/ using the POST verb. Only administrators can access this. It takes a JSON object with the following fields:

- name (required)
- file_name (required)
- model_type (required)
- rating
- comments

This function returns an indicator of success or failure using standard HTTP response codes. On success, it will also return a JSON object containing the new machine learning model. The fields returned are the same fields as in the Retrieving Machine Learning Models endpoint. In addition, on success, the machine learning model will be saved to the database.

Viewing a Machine Learning Model
This function will be accessible via the endpoint at models/{id}/ using the GET verb. Only administrators can access this. It returns a JSON object with the same fields as above (Retrieving Machine Learning Models).

Updating a Machine Learning Model
This function will be accessible via the endpoint at models/{id}/ using the PUT or PATCH verbs. Only administrators can access this. It takes a JSON object with the following fields (fields are optional if they are not being updated):

- rating
- comments

This function returns an indicator of success or failure using standard HTTP response codes. On success, it will also return a JSON object containing the updated information. The fields returned are the same fields as above (Retrieving Machine Learning Models). In addition, on success, the MLModel model in the database will be updated accordingly.

Retrieving Machine Learning Models by Job
This function will be accessible via the endpoint at jobs/{id}/models/ using the GET verb. Only administrators can access this. It returns a paginated list of JSON objects containing the same fields as above (Retrieving Machine Learning Models). In this endpoint, the machine learning models list is filtered and only includes machine learning models for the specified job.

Viewing API Documentation
This function will be accessible via the endpoint at docs/ using the GET verb. Anyone can access this. This page contains all of the documentation relating to the Moth Classifier API. It contains more detailed information about each API endpoint, includes a description of each field, lists potential query parameters, and contains sample requests and responses.

Moth Classifier Project's Core Algorithm
The core algorithm that will be used for the Moth Classifier project is an image classification model that is coded in Python with the Tensorflow library. This library allows for high-level machine learning models to be created with the availability of tuning hyperparameters. The image classification algorithm is implemented inside a Python class data structure titled *mmodel.py*. The algorithm will be trained from the thousands of images of moths that are already classified from the Discover Life website. These images will be web scraped and organized into a ZIP file by the API. The machine learning class can then extract and interpret the image directory from that zip file. This can be done through Tensorflow's *keras.utils.getfile()* function. The machine learning algorithm can then be used to classify images outside of this dataset through their respective image addresses (URLs).

Image Classification Class Functions
__init__(m_name, m_type, comment, f_url, f_name, num_classes, batch_size)

- **Purpose**
    - Extracts an image dataset from the input link of the zip file.
    - Initializes machine learning model with an 80-20 image training to validation split
- **Parameters**
    - *m_name* - name of the current model created
    - *m_type* - specifier of machine learning model, either DETECTOR or CLASSIFIER
    - *comment* - date and purpose of the machine learning model built
    - *f_url* - URL of zip file that contains an image directory to train the model
    - *f_name* - desired name of the extracted folder being created
    - *num_classes* - number of moth species from the dataset

        - *batch_size* - number of training samples per iteration when training model (default = 32)

__str__()

- **Purpose**
    - Prints out build summary and the metadata information that will be sent to the API

train(epochs)

- **Purpose**
    - Builds image classification model with the default 80-20 split
    - Uses data augmentation before processing images
    - Trains model by grouping a stack of layers
    - Returns training summary of the model
    - Saves the model as [*m_name*].h5
- **Parameters**
    - *epochs* - number of iterations that the model will run through the training dataset

load()

- **Purpose**
    - Loads the model [*m_name*].h5
    - Prevents the retraining of an identical model

test(i_url, i_name)

- **Purpose**
    - Tests the trained machine learning model on an input image
    - Returns classification and confidence rating
- **Parameters**
    - *i_url* - URL of test image
    - *i_name* - name of image from URL

Division of Labor

The Moth Classifier design consists of three distinct components of equal complexity: mobile application/user-interface development, API, and machine learning. Since the development team is made up of three people, each person will be assigned one of the three components. Reese Jones is responsible for developing the front-end mobile application interface through Flutter. He is also responsible for creating a Google Firebase storage platform that will hold information needed by the mobile application. Abid Ahmed is responsible for creating the API of the entire system. He is responsible for linking the machine learning code to the mobile application through the Django REST Framework. Additionally, Abid will be responsible for the communication between our platform and the Discover Life website. Ben Giangrasso is responsible for creating machine learning code with Python and Tensorflow. Ben will also retrain machine learning models to ensure that the models used in the mobile application will give the highest classification grade.

Timeline of Intermediate Milestones

The design's modules will be completed by the timeline listed below. Each milestone of the timeline will build up to the initial release of the mobile application on various marketplaces as well as its release to Discover Life.

09 December 2020

- Basic integration with API and machine learning using a pub/sub pattern
- Basic data flow between API and mobile application using HTTP calls
- Testing moth classification model with nontrivial accuracy
- Finalize skeleton of all blank and semi-developed screens in mobile application

20 January 2021

- All screens have business logic and data to handle some type of interaction
- API can successfully return data to the Firestore DB and the Google Cloud Messaging service
- API can successfully create jobs for the machine learning model and collect its responses
- Expand classification to include at least one species with reasonable accuracy
- Test an improved accuracy version of moth classification model (separate from species classification)

24 February 2021

- Submitted mobile application to marketplaces for approval
- Create and deploy API to production environment
- Deploy machine learning to production environment
- Test machine learning model that can classify two species of moths with sufficient accuracy

07 April 2021

- Successful data flow from image submission and classification through all components
- Production environment is stable
- Project is packaged to hand off to Discover Life
- Mobile application automatically monitors and reports Crashlytics for bug identification
- Mobile application is published to Play Store and iOS App Store