CSC 431

Phantastic Fungi

System Architecture Specification (SAS)

Team 06

JuanCarlos Jimenez	Shroom Master
Nolan McCarter	Requirements Engineer
Matthew Rossi	System Architect

Version History

Version	Date	Author(s)	Change Comments
1.0	3/31/22	JuanCarlos Jimenez Nolan McCarter Matthew Rossi	First draft

Table of Contents

Table of Figures

1. System Analysis

1.1 System Overview

Phantastic Fungi is a mobile application whose main functionality allows users to identify a mushroom by taking or uploading a picture from their phone, from which they receive a genus and species classification. Computer vision models pretrained on the ImageNet dataset can be finetuned on a mushroom dataset to learn this identification task.

The basic architectural style for our application is model-view-viewmodel. The model, which consists of an account database and two neural networks, will be stored entirely on a backend web server. The viewmodel queries or serves input to the model and can view its result.

The system is divided into two key functionalities: Access Control (account creation/login, continue as guest, settings management, etc.) and Mushroom Classification.

1.2 System Diagram

Our use case diagram, which incorporates all functional requirements, demonstrates the application system.

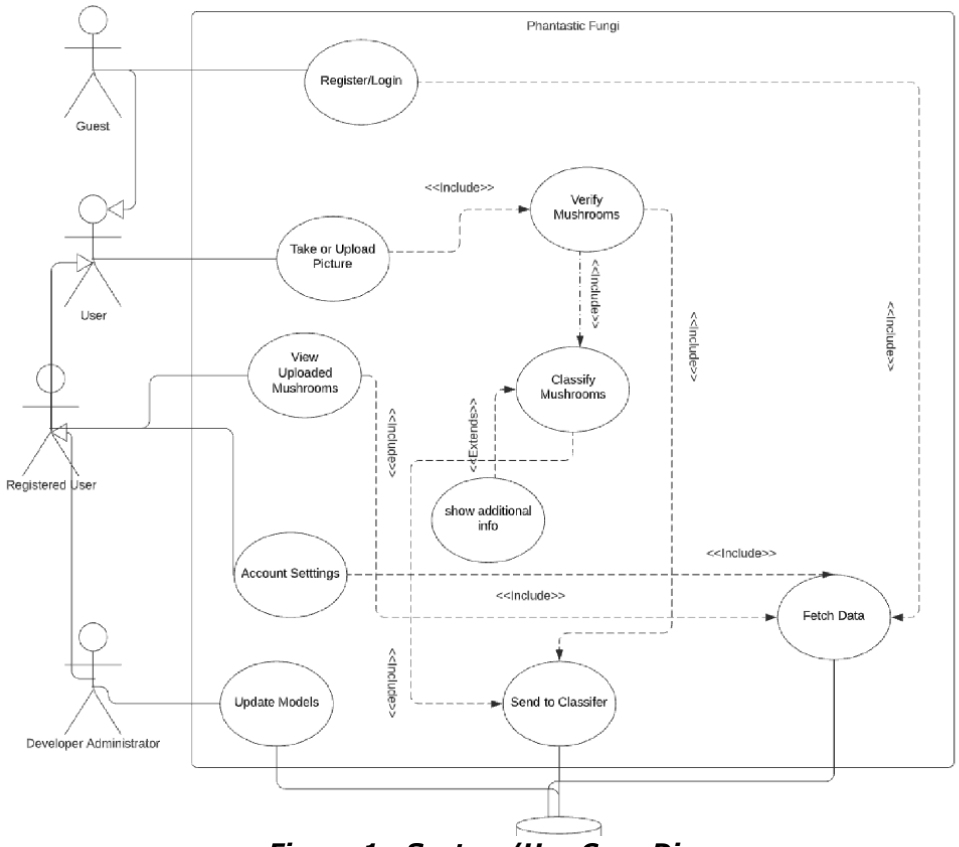

Figure 1 - System/Use Case Diagram

1.3 Actor Identification

- **User:** A human actor that interacts with the application to classify an image of a mushroom.
    - **Guest:** Limited-functionality subtype of user. Only has access to Register/Login and Take/Upload pictures.
    - **Registered User:** Full-functionality subtype of user; has an associated username, password, email, settings, and access to view their previously uploaded mushrooms.
- **Database Administrator (DevOps):** The database administrator is a human actor who maintains the neural networks, updating parameters with better models if/when necessary.
- **Camera:** The camera is an external system actor that can be used by the application (on devices with a camera) to take a picture of a mushroom that a user wants to identify.

1.4 Design Rationale

1.4.1 Architectural Style

Our application is designed according to the Model-View-Viewmodel (MVVM) architectural style, which is an evolution of the Model-View-Controller (MVC) style. In MVVM, the viewmodel replaces the controller, inheriting its responsibilities and taking on the additional role of formatting the data for the view so that the model doesn't have to know what the view looks like. This allows for a simplified black-box-style model and looser coupling than MVC.

Under such a paradigm, the model doesn't know about the view model – all it knows is that it receives input (in this case, image data or a database query) and returns the output. Similarly, the viewmodel does not know that the view exists – it need only know its internal logic and the information it receives from the model. Such loose coupling both facilitates application testing and makes it easier to add functionality in the future.

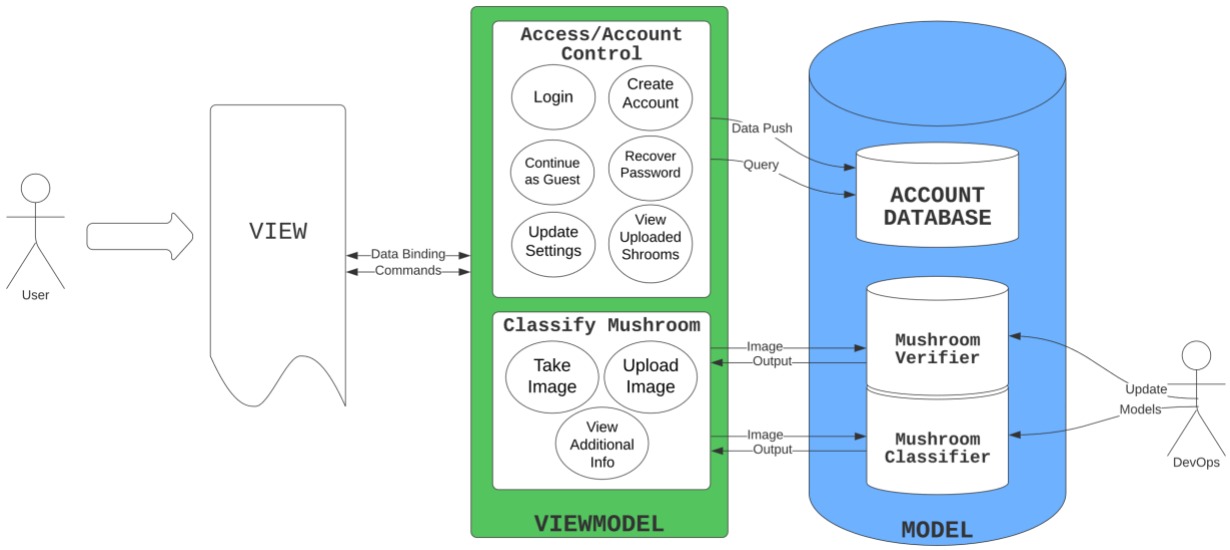

Figure 2 - Architectural Style Diagram

Here, the user sees the view, which observes the viewmodel and sends commands to it. The viewmodel is divided into two main areas of functionality: *Access/Account Control* and *Classify Mushroom*. The former contains protocols for account creation and updating settings (which push new data to the database); login, password recovery, viewing uploaded mushrooms, and fetching settings (which query the database); and continue as guest (which doesn't interact with the database at all). When the user indicates that they want to classify a mushroom,

the application instantiates a classification object which has the functionality detailed in the latter functional area. This classification object allows the user to take or upload an image, which it then sends to the mushroom verification binary classifier, which detects the presence of a mushroom. If a mushroom is detected (or the user decides to continue with the given image anyway), the image is sent to the mushroom classifier, which returns the predicted genus and species.

1.4.2 Design Pattern(s)

The separation of frontend and backend into viewmodel and model gives rise to a pattern similar, but not identical to, a façade. In our implementation, the number of ways in which the viewmodel can interact with the model is designed to be as limited as possible to simplify the integration and promote loose coupling. Although there is no explicit façade component, the pattern principle applies.

In the backend, the neural network objects are created according to the singleton design pattern. Consequently, there can never be multiple copies of the verifier or classifier, which would waste space and potentially lead to the viewmodel's API calls being handled ambiguously. However, this design pattern means that when the Database Administrator updates the model parameters, the old neural network object must first be destroyed before the constructor is called with the new parameters.

1.4.3 Framework

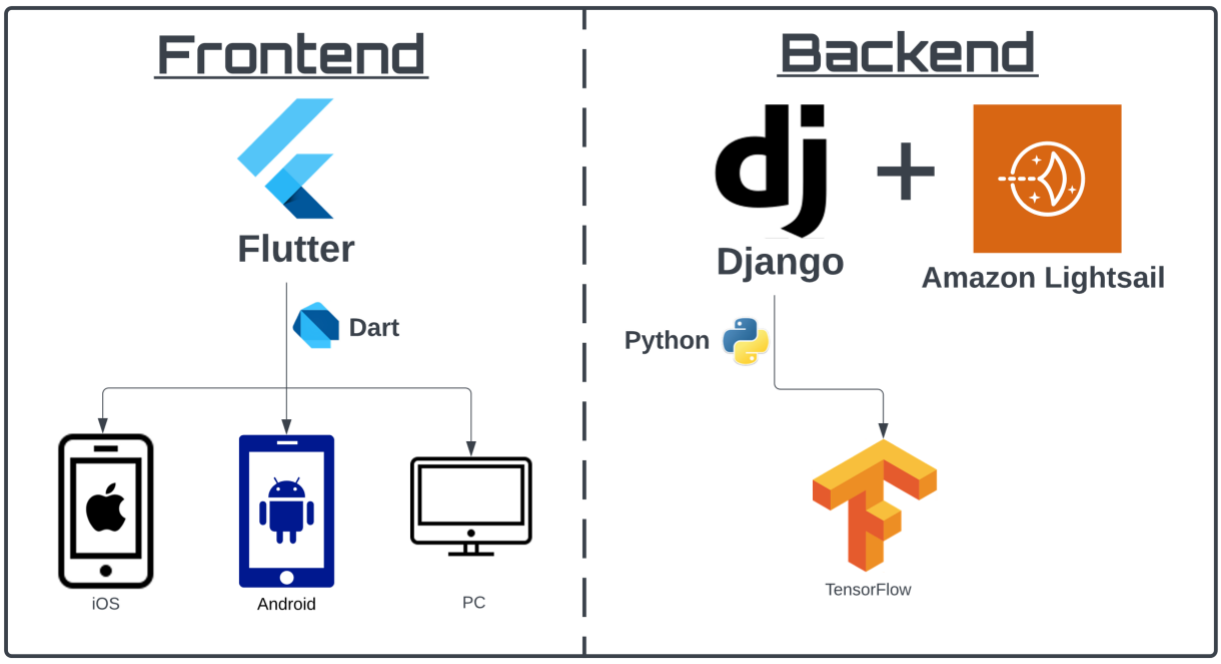

Figure 3 - Framework Diagram

Our user interface (UI) will be developed in Google's Flutter, a Dart-based development kit designed for building interactive applications. Flutter's platform-independence allows us to create an application that functions smoothly on both mobile and desktop with a single codebase. The frontend comprises both the view and viewmodel, for which Dart is powerful enough to implement the internal logic.

The frontend, particularly the viewmodel, has access to an application programming interface (API) exposed by the backend, which will be written in Python using Django. Django integrates easily with an Amazon Lightsail web

server, which is needed to host the account database and computer vision models. Django was a practical choice because it is Python based, simplifying integration with the scripts for the computer vision models, which are already implemented in Python (using TensorFlow) in the repository for the Google Brain paper *Big Transfer (BiT): General Visual Representation Learning*[1]. In addition to hosting the neural networks, the server hosts the account database, which contains information for all users, including username, email, password, settings, and uploaded mushrooms. Because this database is fairly simple, it will be implemented in Python, rather than something more powerful like SQL, which would require more effort to integrate.

In sum, the entire model section within the MVVM architecture is implemented in the backend in Python and hosted on an AWS web server.

[1] https://github.com/google-research/big_transfer

2. Functional Design

2.1 Account/Access Control Sequence Diagram

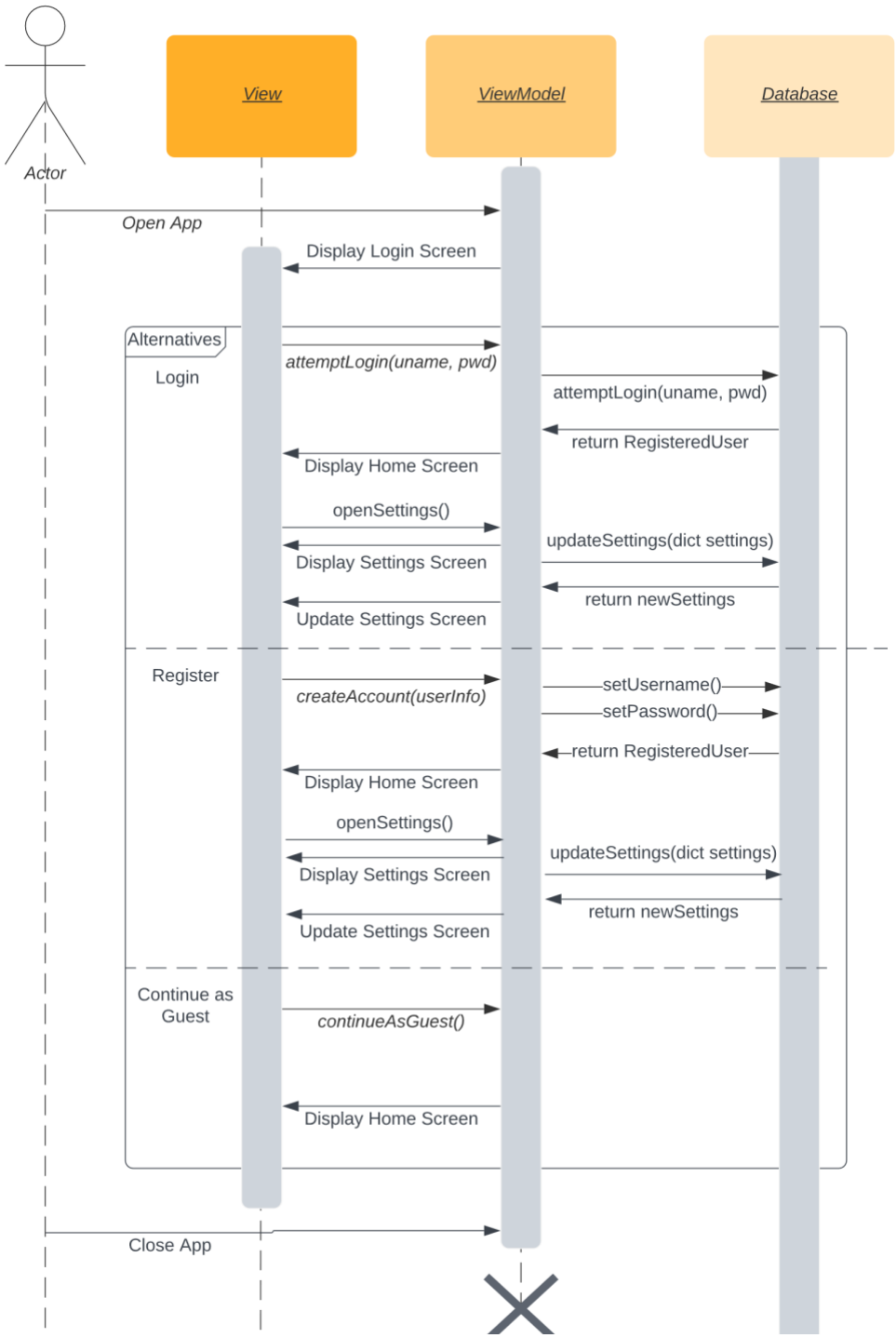

Figure 4. Account Sequence Diagram 1

2.2Mushroom Classification Sequence Diagram

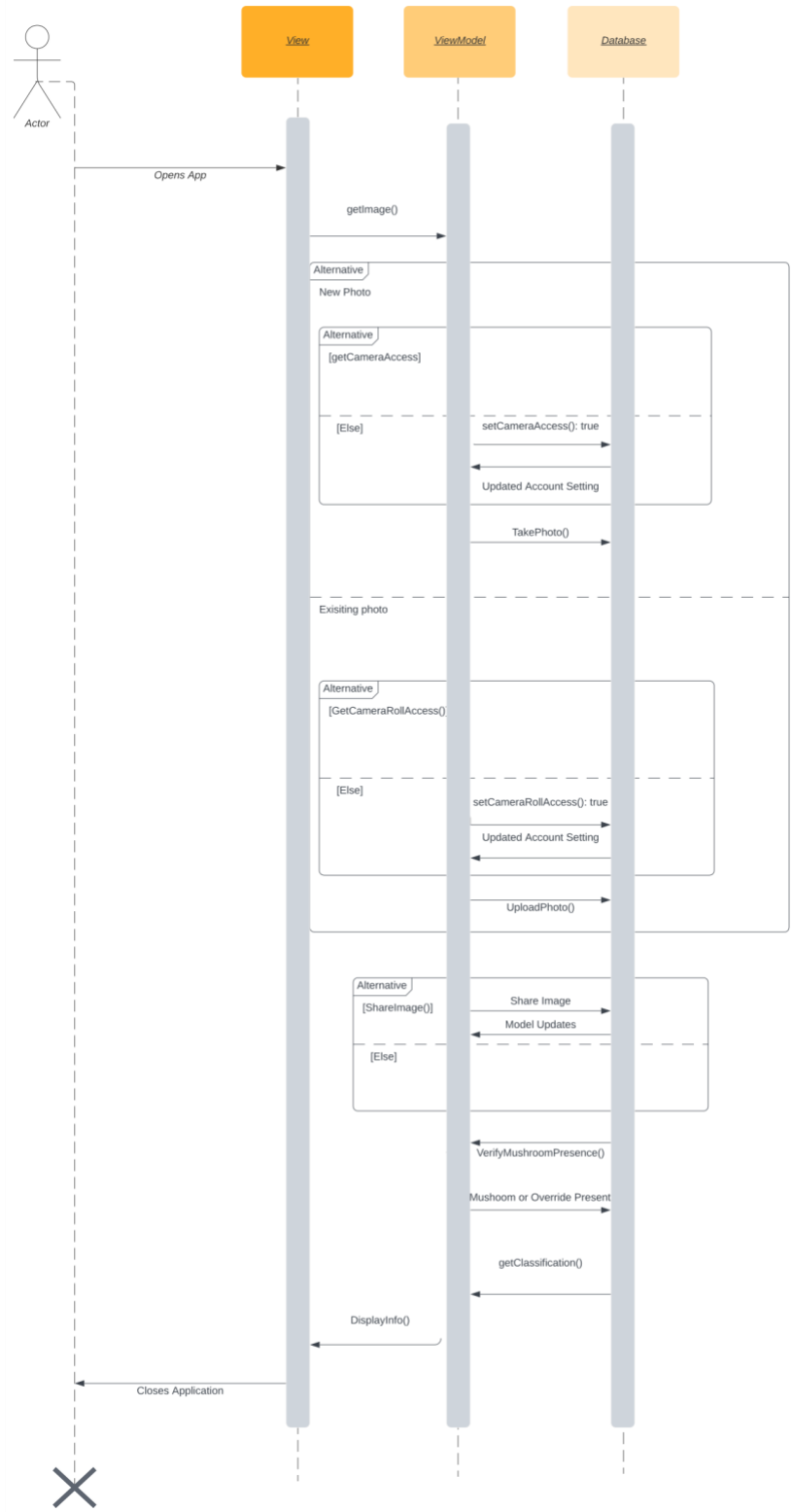

Figure 5. Classify Sequence Diagram 1

3. Structural Design

The following is a UML Class Diagram that depicts the core functionality of the viewmodel in our system.

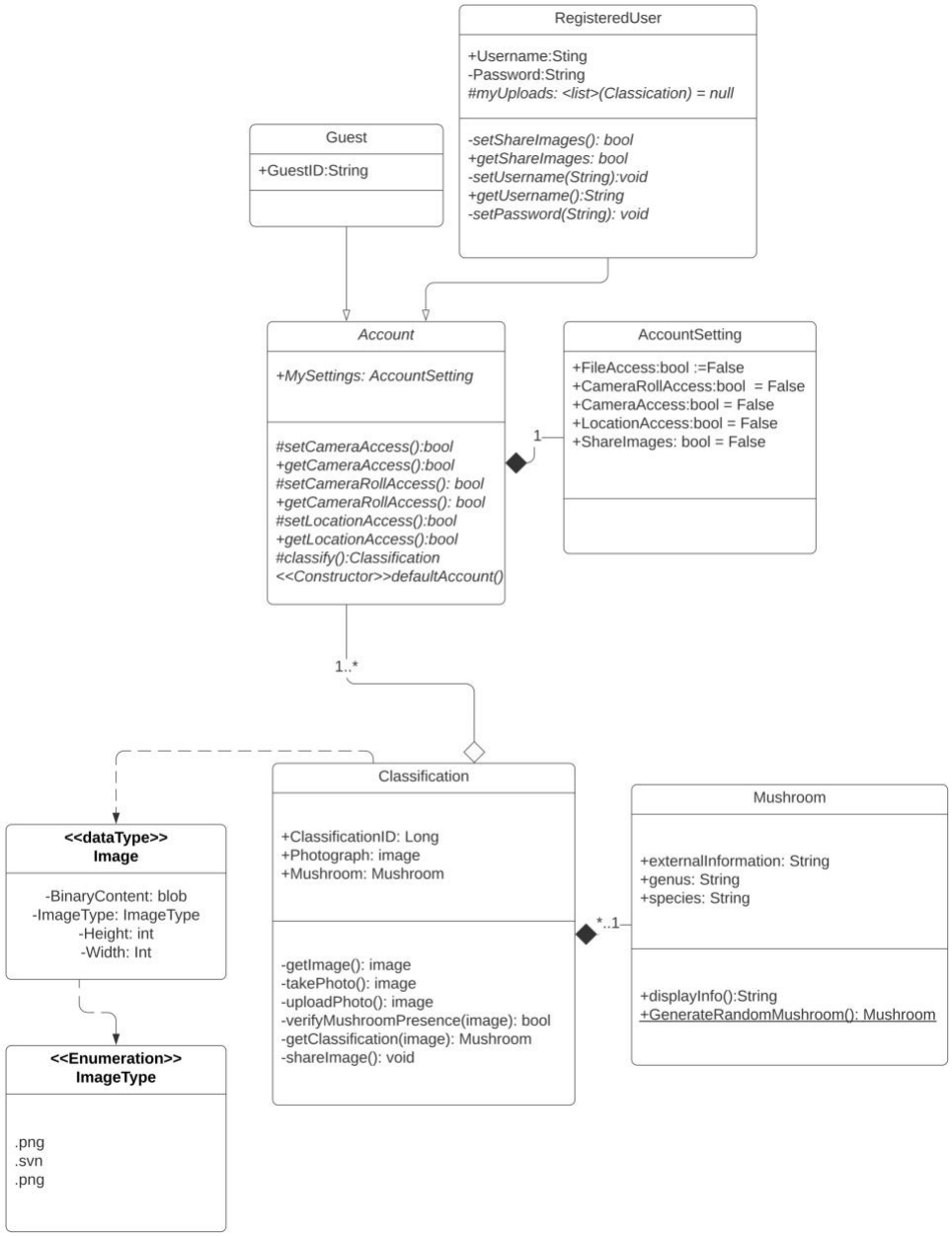

Figure 6. Phantastic Fungi Class Diagram 1

Account is an abstract class with concrete subclasses Guest and RegisteredUser. The RegisteredUser class can set account information or view uploaded mushrooms. An Account object can update settings, and it can access the main functionality of the application by interacting with a camera device and subsequently creating a Classification object.

The classification object is constructed with an image provided by the user, with which it can query the model for verification and classification.