pup.ai Final Documentation

Group Members:

Frontend: Jinhoo Jeung, Duncan Shepherd, Jian Du

Backend: Tiangang Chen, Yuxian Zhu, Dillon Davis

Model: Dillon Davis

Table of Contents:

Description

pup.ai is inspired by the Kaggle Dog Breed Identification challenge. It proposed a technically challenging problem of understanding and classifying imagery that could be easily packaged into a fun, usable product for all dog lovers.

pup.ai allows you to instantly identify the breed of a dog using computer vision and deep learning techniques by simply taking a picture of it. It will give the top three most probable breeds along with a confidence score for each proposed breed. The application then helps you learn more about the breed through traits such as physical characteristics and behaviors such as friendliness. You can also search for this information about any breed you might be interested in and favorite breeds that you liked the most. Finally it connects you with people who are also interested in similar breeds so users can share their knowledge of breeds and dogs that may not be included in the application.

Process

We followed the XP without strict enforcement of pair programming. It worked efficiently for the scale of the project and the organization of iterations that lasted approximately two weeks. Our team is divided into three subteams: Model, Server, and Client. We began every iteration by meeting and assessing technical debt and bugs, planning user stories and features for the next iteration, and assigning tasks to team members. Tasks and features were assigned evenly to members of subteams but the workload often became lopsided on certain subteams due to teammate issues. We focused our user stories on realistic scenarios of how a customer may want to use our product. Subteam members met during iterations to work together on new features and iron out bugs when necessary. We met again a few days before each iteration to sync on progress, evaluate each other's work, and work out any kinks in new features. Finally we met right before each iteration to assign Moderator/Reviewer/Scribe roles and plan our presentation.

We put a large emphasis on testing with extensive tests and test coverage that ran automatically for every pull request for the server and Espresso UI tests that evaluated the client. We required that every user story be tested with several tests and manually tested behavior of the server through the client as well. This improved communication between subteams and the quality of our code and tests. We enforce code style through linters that automatically run on the CI to enforce PEP8 and AOSP Java styles.

We used standard git development practices with branches for subteams and features as well as frequent commits to have a detailed history of changes to our code. This allowed us to quickly and efficiently isolate and solve bugs. Subteam members reviewed eachothers code during meetings. We also continuously made an effort to refactor the codebase every iteration in accordance with requirements. This allowed every team member to develop a better understanding of code they may have not written and improve the quality of the codebase through modularization and deduplication of logic.

Once we worked out the kinks and refined our process, it allowed us to efficiently and correctly iterate on features through accountability and collaboration when necessary.

Requirements & Specifications

Iteration 1 (Jan 30 - Feb 5)

First team meet-up, we discussed some basic logistics and everyone was assigned subteam based on his or her experience. We also spent some time evaluating available frameworks and tools, and some set up work was done as well.

Estimated	Story Description
3 units	Write Project Proposal
2 units	Pitch/Form Teams
1 units	Setup Version Control and Team Communication
1 units	Evaluate Platforms, Libraries, Frameworks
1 units	Choose Testing Framework and Coding Style

Iteration 2 (Feb 6 - Feb 20)

The focus of this iteration was on researching, planning, learning, and setting up our approaches for the client, server and model. Layouts for user login and signup are implemented as well.

Estimated	Story Description
8 units	Setup Web Server, Database, Cloud Service
2 units	Design API
5 units	Image Classification Model Research/Selection
6 units	Data Preprocessing/Model Prototyping

2 units	Android and Espresso Setup
4 units	Create Login/Sign-up page
4 unit	Create user main page

Iteration 3 (Feb 20 - Mar 5)

More design details were settled down during this iteration: a non-registered use will allowed to use core functionality (image classification), registered and logged in user can enjoy additional features like view other user profiles, add his or her favorite breeds and friend other users. Breed information was populated into database, manual login, registration and image upload were performed.

Estimated	Story Description
4 units	Model/Build Database Schema
6 units	Convert Flask Server to Django
5 units	Registration Endpoint
5 units	Login Endpoint
3 units	Deploy server on Azure
4 units	Prototype/Train All Selected Models
6 units	Build Dog Info Scraper
2 units	Send Login information to actual server
2 units	Add Logout and put register and login button in the menu
2 units	Asks permissions from a user to access camera and local storage

3 units	Send images taken from a camera to server
3 units	Browse images from a local storage and upload it to server
2 units	Automated Testing

Iteration 4 (Mar 6 - Mar 16)

Different components implemented by each subteam were integrated in this iteration. Despite sporadic failures, users can upload a picture to pup.ai, which the server will respond with the top three most probable breeds of the dog as well as the likelihood of being each of the given breeds. Additional API endpoints were implemented to allow client query information stored in database.

Estimated	Story Description
4 units	Hypertuning and Evaluating Dog Breed Classification Model
3 units	Dog Breed Classification Endpoint
5 units	User profile endpoints
8 units	Connect model to classification endpoint
5 units	Set up continuous deployment
4 units	Set up CI tests for Android
2 units	Receives dog breed data from server
2 units	Separate register and login page
2 units	Automated Testing

Iteration 5 (Mar 26 - Apr 9)

We focused on returning the classification results and related breed information to the user during this iteration. So, once a user logged in he or she can can edit profile information and upload profile pictures. The server will also respond breed information from classification results or by searching with provided keyword, and the client will render those information for the user.

Estimated	Story Description
3 unit	Favorite Breeds Endpoints
5 units	User Profile Picture Endpoints
4 units	Breed Info Endpoint
4 units	Breed Search Endpoint
4 units	Breed Results Frontend
4 units	Favorite Breeds Frontend
4 units	Breed Info Frontend
4 units	Breed Search Frontend
3 units	Improve User Interface

Iteration 6 (Apr 10 - Apr 23)

Some nice-to-have features were added, and we spent a lot of time debugging server and client. Now the users will be notified if they attempt to classify an image that does not contain a dog as the subject. Users interested in similar breeds will be suggested for user to friend with, simply by sending a friend request.

Estimated	Story Description
3 unit	Non Dog Image Detection Endpoint
3 units	Connecting Users with Same Favorites Endpoint
3 units	Friending Endpoint

3 units	Non Dog Image Frontend
3 units	Suggested Friends Frontend
3 units	Friending Frontend
4 units	Improve User Interface
1 units	Run App on Phone

Architecture & Design

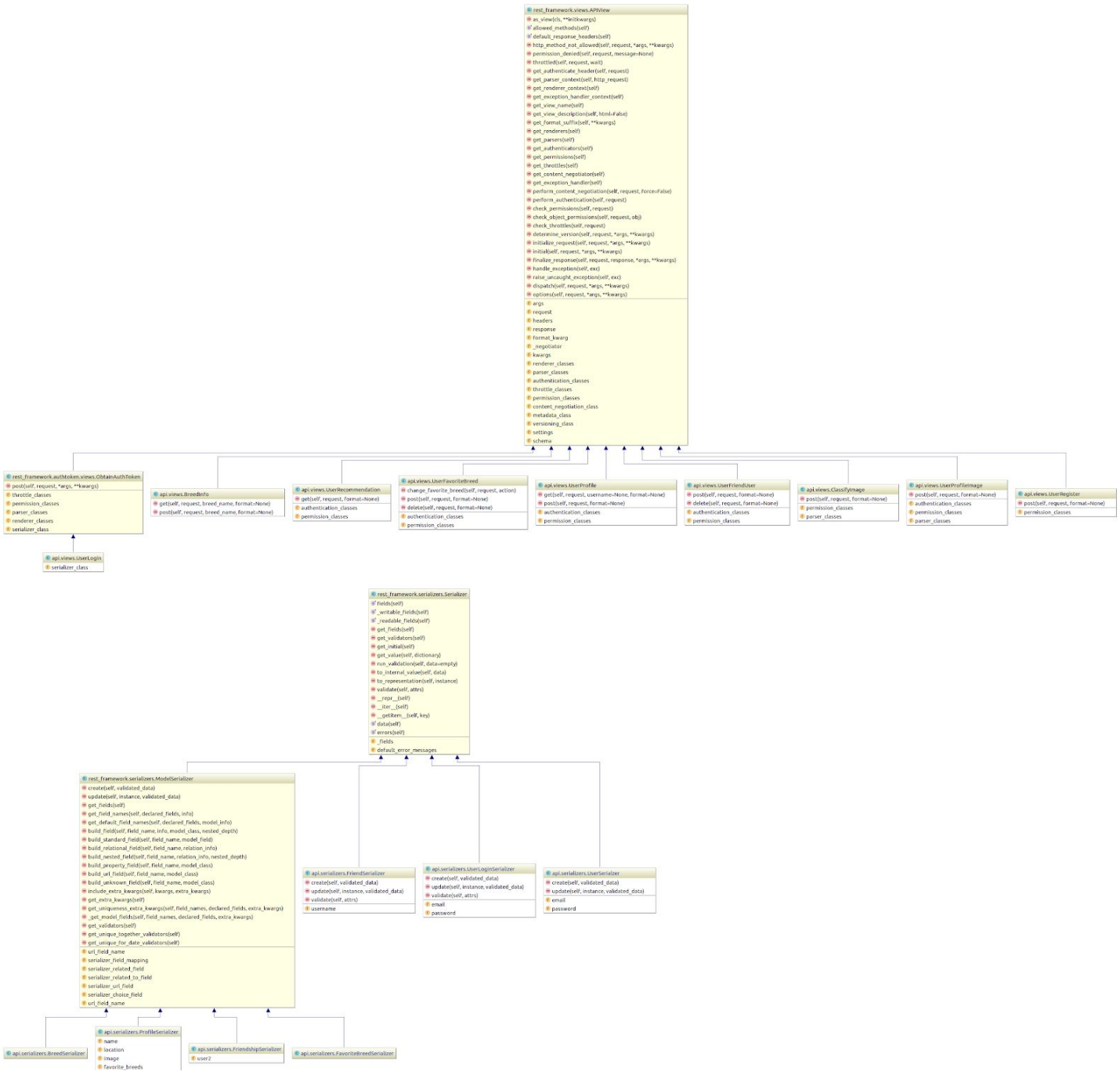
Top Down Architecture

Model

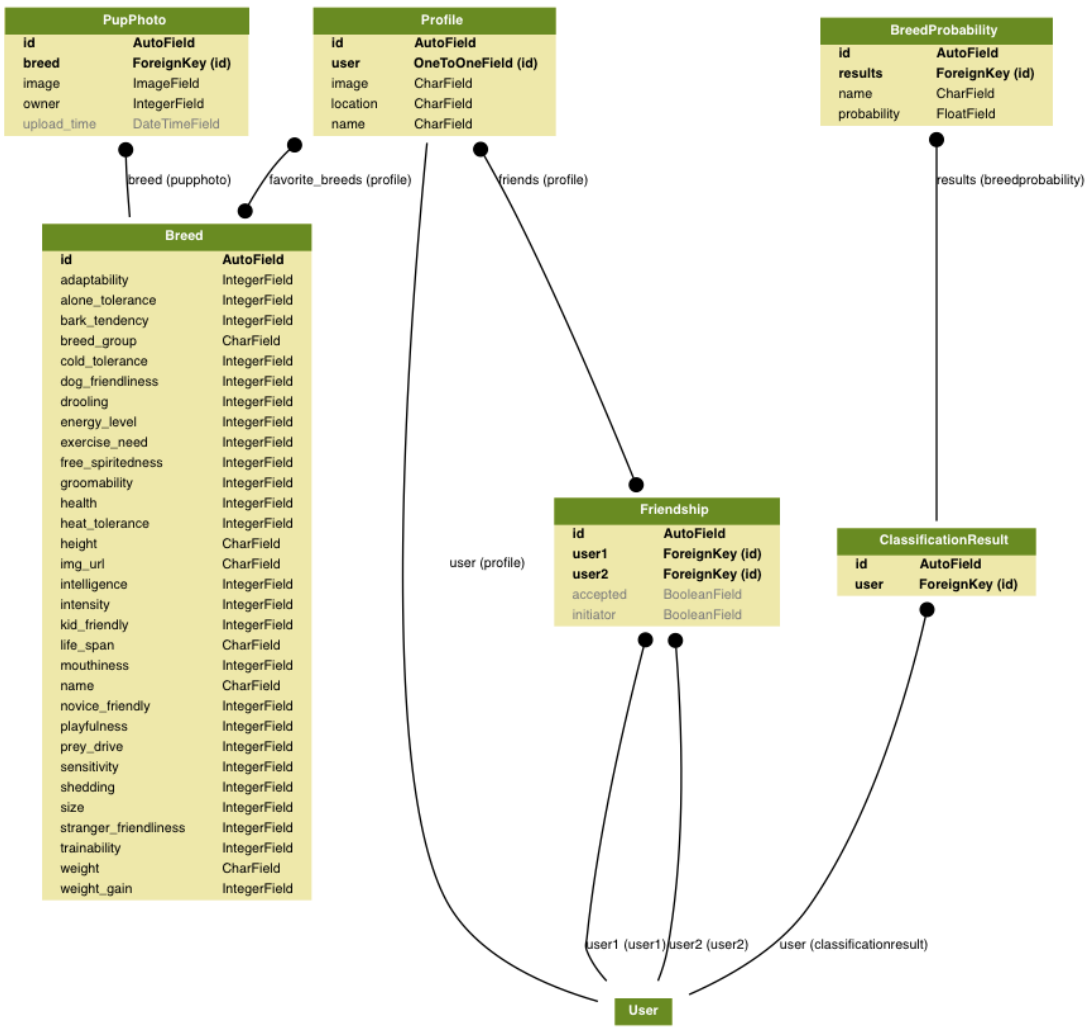
This component holds convolutional neural networks, training, and data processing as well as public logic required to use the trained model to determine the breed of a dog image. There are five different convolutional neural networks: `PupInceptionNet`, `PupVGGNet`, `PupResNet`, `PupDenseNet`, `PupForestNet` and two datasets: `StanfordDogs` and `KaggleDogs` we experimented with. These networks are trained with the datasets using a `Manager` and an `Optimizer`. The manager consumes the dataset, initializes a model, trains the model, and saves the model. The optimizer updates the internal state of the model when the manager trains the model. Trained models are automatically saved under `model/checkpoints` and data should be placed in `model/data`. `model/src/utils.py` contains the public method `classify_image` that takes an image and classifies it using the trained network. Our choice of PyTorch as a framework made prototyping and training very simple and easy.

Server

This component consists of the database and a web service framework that processes user request and responds with information from either a query to database or a call to the model. Initially we wanted to use Flask for its lightweight, but later we decided to switch to a more full-fledged framework Django. It also has support for REST APIs, and we validate user requests with `Serializers`, and the depending on where the request was routed, logic in `views` will do the actual database query or model invoke, accordingly. Talking about database, the schema of models we used is down below. Information that `Breed Scraper` scraped from Dogtime was stored in the database for search or query. Our choice to use a relational database like MySQL instead of a NoSQL database such as MongoDB was heavily influenced by Django. It has built-in robust support for many relational databases which made working with a relational database much easier for our service.



(`View` and `Serializer` class diagrams: we utilized existing framework classes very heavily)



(Model diagram: database model like `User`, `Friendship`, `Breed`, `Profile` etc were defined)

Client

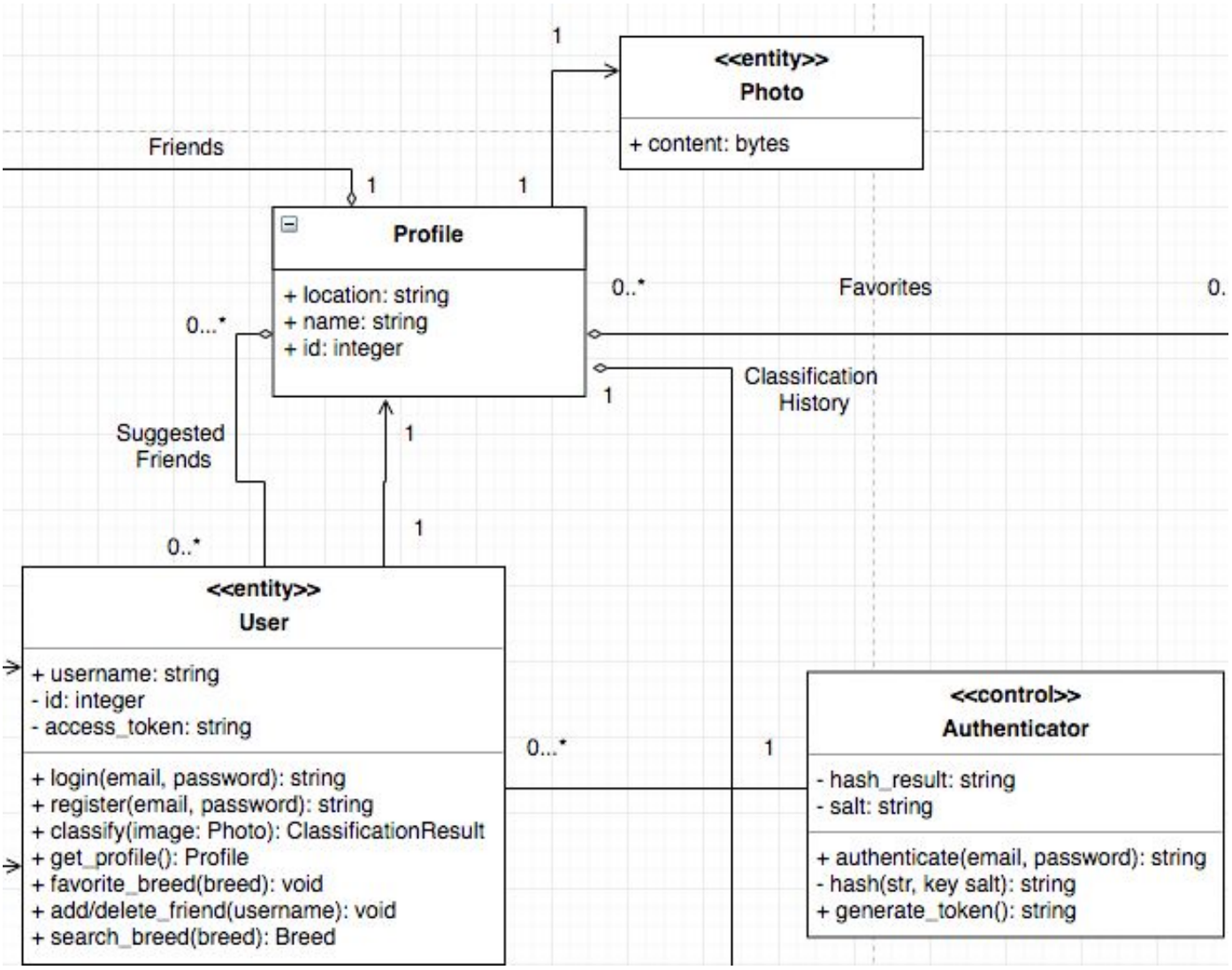
This client component followed the MVC architecture. The model portion consumes data from the server and makes it readily available. `ApiConnector` in 'utils' directory performs most of this. The controller responds to user input by retrieving, editing and creates objects with the data model. All the `Activities` associated with their `XML` views behave like the controller. Finally, the view displays the information to the user and consumes user input. In Android environment, all the views are inside `Layout` folder under 'res' directory. Android enforces this architecture so it heavily influenced the architecture of the client.

UML Diagrams

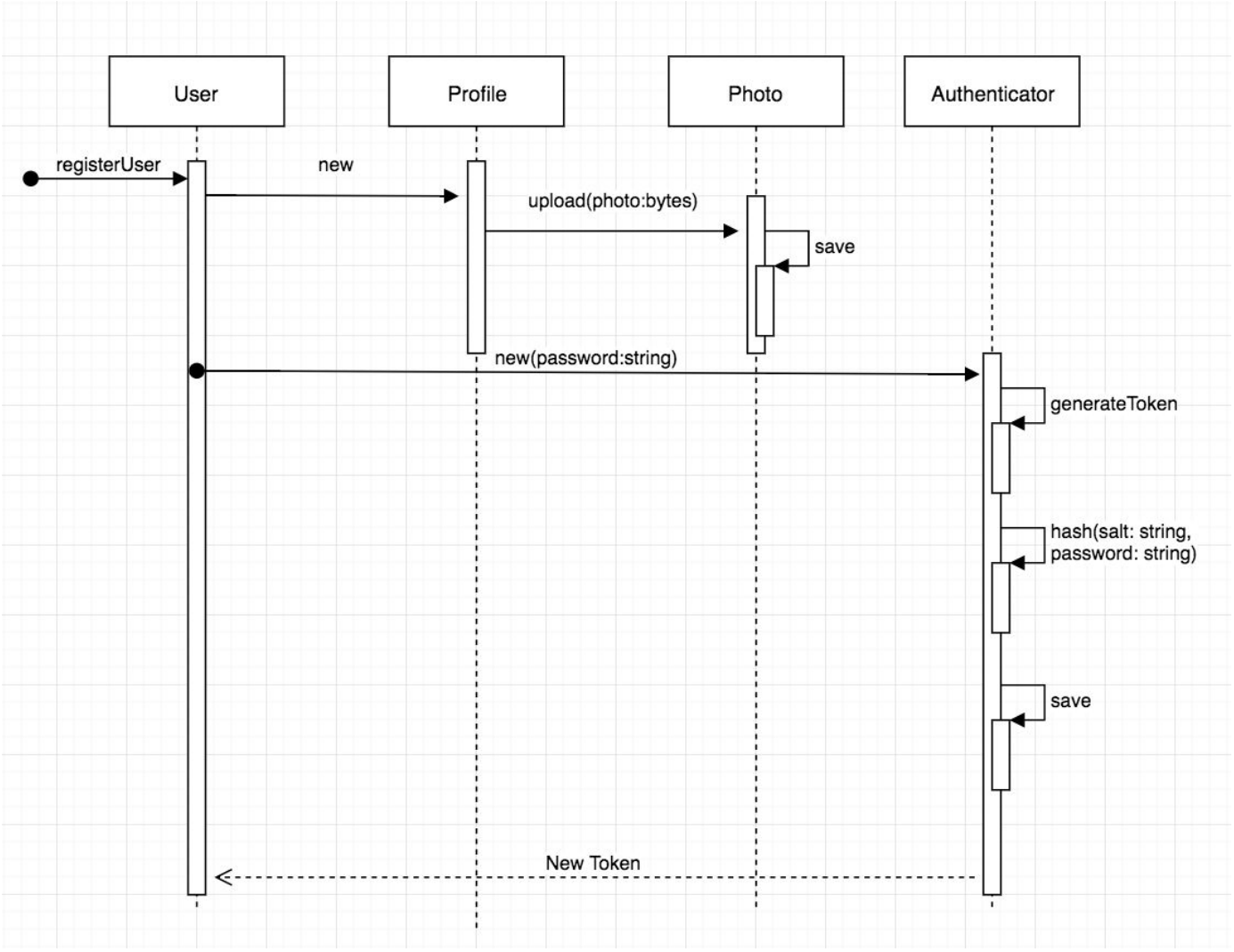
Login/Registration

These diagrams show the classes as well as the sequence of actions for login and registration. The user registers with a username and password. The password is hashed with a

salt and stored. If registering, a profile is made for the user which stores user information and activity. An authentication token is generated for the user and returned on registration and login.



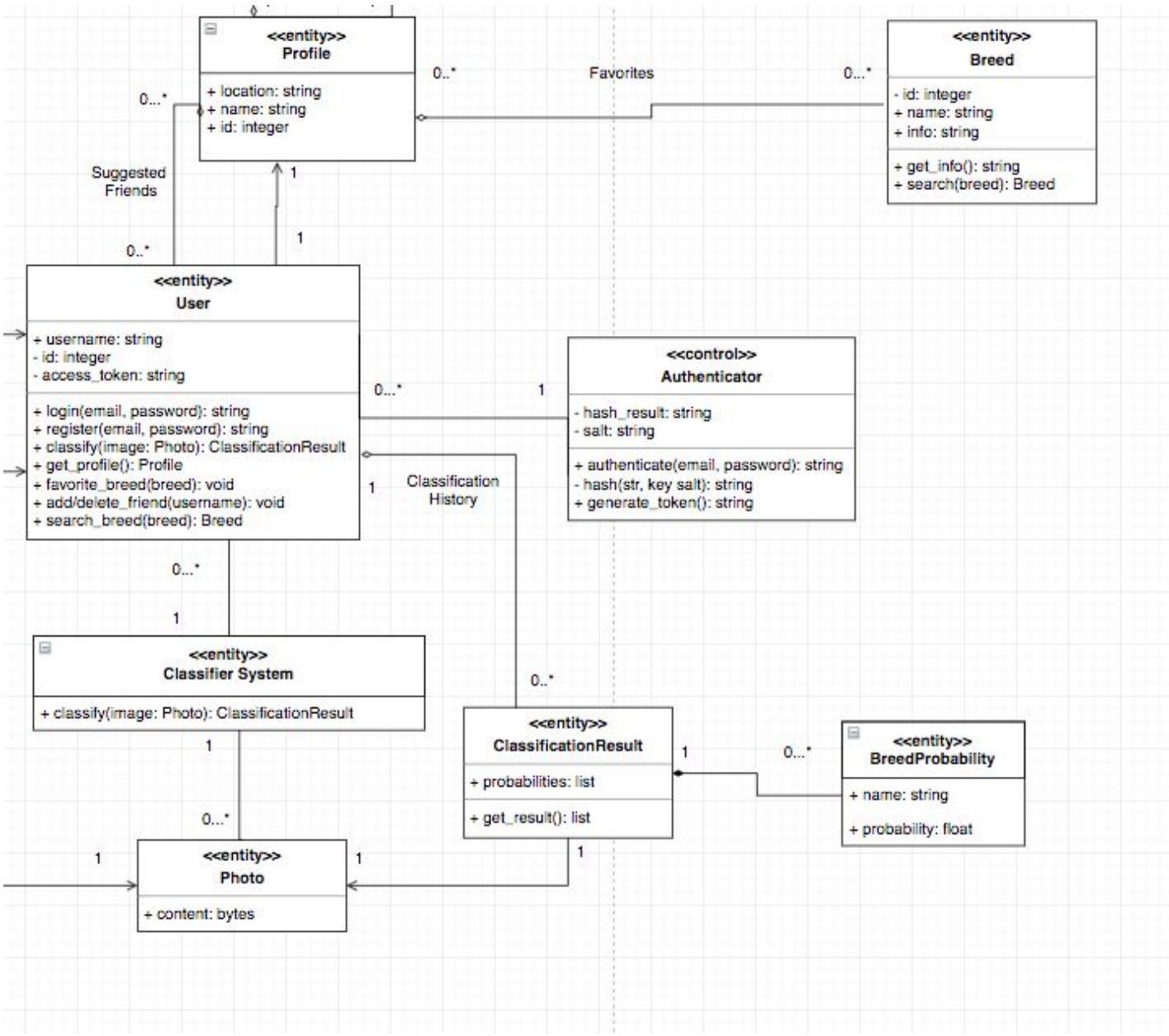
(User Class Diagram: User registers and logs in with the authenticator. Profile stores auxiliary information about the user and the profile picture)



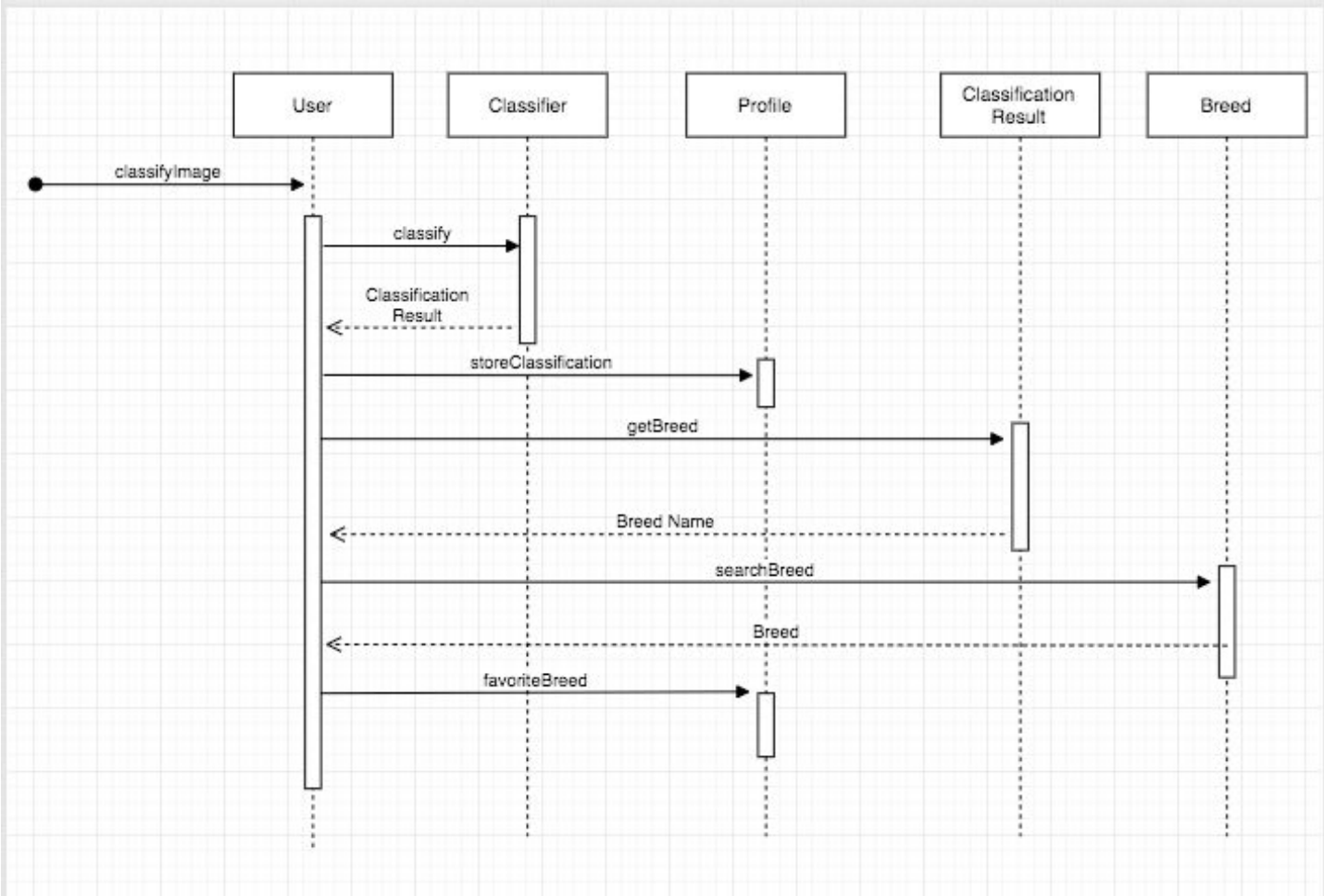
(User Sequence Diagram: User creates profile, uploads a profile image, and then authenticates using the authenticator.)

Breeds

These diagrams show the classes for all breed related features as well as a common sequence of actions. The user can classify an image of a dog and get a set of most probable breeds for the dog in the image. The user can then search for these breeds and find more information about their traits and characteristics. They can then choose to favorite the breed if they would like to save it for later. These favorites as well as the classification history is stored with the profile.



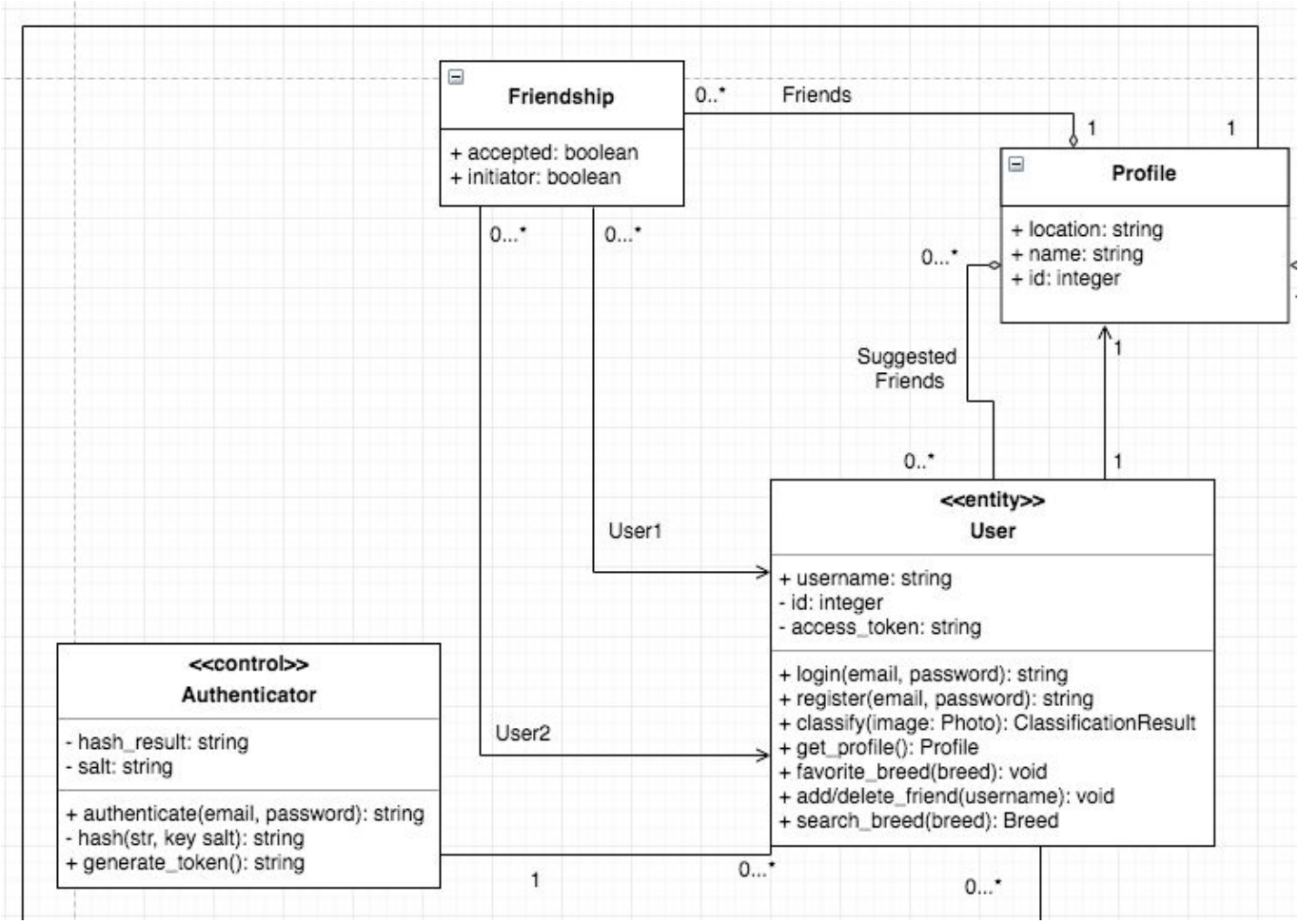
(Breed Class Diagram: Central User makes all requests. The user classifies an image with Classifier System and a Photo. ClassificationResults made of BreedProbabilities are stored with the user. Profile stores all favorited breeds for the user)



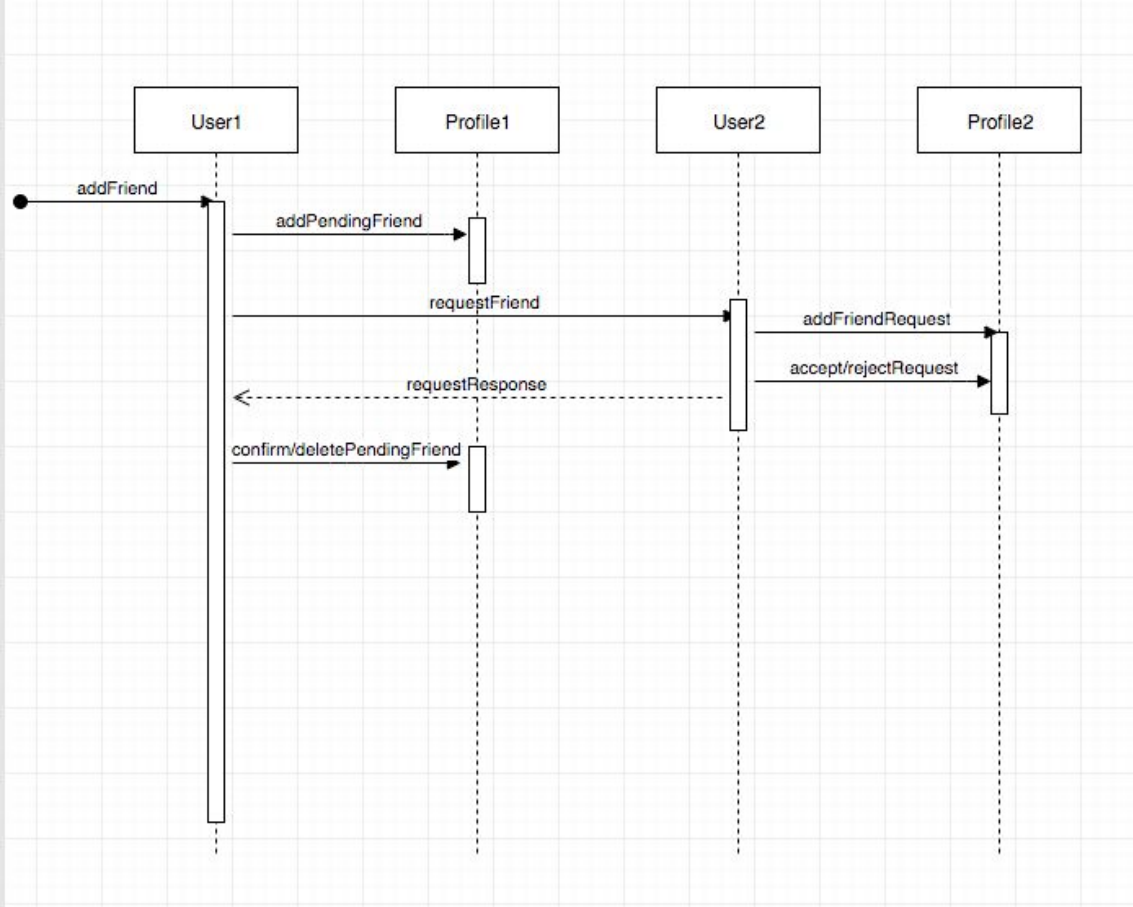
(Breed Sequence Diagram: User classifies an image with the classifier and gets a result and stores it. The user can then search for information about breed results using the breed name. If we have information about the breed we'll return characteristics about the breed. The user can then choose to favorite it.)

Friends

These diagrams show classes for user friendships and the sequence of actions for user friending. User1 will send a friend request to User2 and this pending request will be added to User1 and User2's profiles. User2 will then either accept or reject the pending on their profile and this response will cause User1's profile to either add User2 as a confirmed friend or remove the pending request.



(Friends Class Diagram: Friendships are represented with two Friendship objects. Each user involved with the friendship has their own friendship object indicating whether they are the initiator or not. These friendships are all stored in the Profile as well as Suggested Friends.)



(Friend Sequence Diagram: User1 adds a User2 as a friend. This saves User2 as a pending friend in User1's profile and User1 as a friend request in User2's profile. User2 can then accept or reject the friend request which will then appropriately confirm/delete the friendship in each profile appropriately.)

Reflections and Lessons Learned

Dillon Davis

I learned about the principles of extreme programming. I learned about project management by leading planning, organization, and delegation of iteration tasks. I also learned about the importance of constant communication and accountability on a team. On the technical side, I learned more about the pitfalls of fine tuning pre-trained CNNs for specific tasks and how finicky the process can be. I also learned about productionizing PyTorch models so that they can be easily used by a service for a client facing product. Finally, I learned more about API Design and Django, a framework that I hadn't worked with before, as well as writing reliable web scrapers for production use.

Tiangang Chen

Throughout working with Django and CI on the backend, I realized the importance of properly handling various environments in a development process. During the semester, we migrated the CI server and application server two times. This raised countless problems for

installation and configuration. When the setup is complex, it is important to use abstractions such as virtualenv and Docker instead of “bare” or “global” setups. Process such as XP naturally calls for at least a dev environment and a production env. So troubles can be avoided if these envs and setup appropriately from the beginning.

Yuxian Zhu

In this project I got more hands-on experience with Django and learned about how to design API interfaces between server and client. I also had chance to experience CI/CD testing & deployment of Gitlab on my campus cluster VM, as well as some trivial maintaining tasks, like setting up server dependencies etc. As far as our team experience goes, my teammates and I spent time working together on iteration planning, and writing user stories, which was not exercised enough from CS 427.

Jinhoo Jeung

I realized how important a software development process is. As we are developing a program as a team, planning and scheduling made large difference in terms of speed and quality of our development. Writing user stories and planning seemed trivial at first, but I realized the importance of extreme programming throughout the project. On the technical side, I learned much about Android. Especially I never wrote tests in Android environment before, but through the project, I learned how to do it. Lastly, I learned more about continuous integration as well.

Duncan Shepherd

I learned the importance of a software development process. Planning user stories ahead has made things feel more viable and more organized. On the technical side, I was new to Android and through the project, I learned much about it. Especially, I became familiar with connecting the application and the server. Handling intents was good learning experience too. Overall, it was very different experience as we had to start the project from scratch.

Jian Du

I learned a lot about the process of software development. I learned that story planning, iterations and meeting checking points are very important and effective ways to sync the process and get things going. I also learned the importance of ownership and holding one person accountable for a specific task. Communication is always one of the central elements of successful teamwork. I also learned the benefits of pair programming, especially when you work with a team remotely. On the technical side, I learned Django, which is a handy and organized tool to integrate with database and server; I learned UI design in Android studio and basic Android app testing framework.

Code Documentation Instructions

Server/Model

Make sure Sphinx is installed. If not, run `pip install sphinx` inside your Python environment. To generate documentations for the server and model, first `cd` into the appropriate directory: `cd server/pupai_django`. Then execute `make html`. The documentations will be output to the directory `server/pupai_django/_build/html`. Use an HTTP server to serve this directory in order to view the rendered HTML documentations.

Client

To generate documentations for the client, you need to install Android Studio. Open the project with Android Studio. On the top, there is a tab called 'tools'. Under the tab, click 'Generate JavaDoc' option. Choose 'Whole project' and 'Include test sources'. Specify 'output directory' and then press 'Ok'. Inside the output directory, open `index.html` to view the HTML documentations.

API Reference

Authentication

APIs that require authenticated privilege require a mandatory HTTP header:
Authorization: Token ...
Failure
HTTP 4XX client error
HTTP 5XX internal server error
In case a response is needed:

```
{
    "error": "<error message>"
}
```

User related

Register

Request:
POST /user/register

```
{
    "email": "<email>"
    "password": "<plain text password>",
    "name": "...",
    "location": "..."
}
```

Response:

```
{
    "token": "..."
}
```

Login

Request:
POST /user/login

```
{
      “email”: ”<email address>”
      “password”: “<plain text password>”
}
```

Response:

```
{
      “token”: “...”
}
```

Get Your Own Profile

Request:
GET /user/profile
Authorization: Token ...
Sample Response:

```
{
    "name": "default name",
    "location": "default location",
    "image":
"https://pupai.blob.core.windows.net/profile/default",
    "favorite_breeds": [],
    "friends": [],
    "suggested_friends": []
}
```

Edit Your Own Profile

Only fields appear in request body are patched, others are left unchanged
Request:
POST /user/profile
Authorization: Token …

```
{
      “name”: “1234”,
      “location”: [“foo”, “bar”]
}
```

Edit Your Own Profile Picture

This will overwrite the requesting user’s old profile picture if it exists

Request:

POST /user/profile_pic

Content-Disposition: attachment; filename=profile.jpg

Authorization: Token …

Response:

```
{
    "url": "profile picture url..."
}
```

Edit Your Own Favorite Breeds

Authorization: Token …

Add a favorite:

`POST /user/fav_breed`

Remove a favorite:

`DELETE /user/fav_breed`

Request body:

```
{
    "name": "breed name"
}
```

Response:

HTTP 204 success

HTTP 418 with body "invalid breed'' if breed name not valid (not in our breed name database)

Get Someone Else's Profile

Request:

`GET /user/profile/:username(i.e. email)`

Authorization: Token ...

Change Password

Request:

PATCH /user/password

Authorization: Token …

Request Body:

```
{
    "old": "<old plain text password>"
    "new": "<new plain text password>"
}
```

Response:

HTTP 200

```
{
      "token": "new token"
}
```

Friending

Request:

`POST, DELETE /user/friend_user`

Authorization: Token …

Request Body:

```
{
      "username": <username>
}
```

Response:

HTTP 200 Success

HTTP 400 Bad Request

Image related

Classify a picture

Request:

`POST /classify`

Content-Disposition: attachment; filename=upload.jpg

Response:

```
{
      "isdog": true/false
      "data": ...
}
```

Search breeds

Request:

`GET /breeds/{breed_name}`

Response:

```
[
      {
            "name":"Golden Retriever"
            "greed_group": "Sporting Dogs"
            "Height": "1 foot"
            etc
      }
]
```