Polyamanita

Group 30

Amelia Castilla
Tyler Clarke
Jan Darge
Layne Hoelscher
Nawras Rawas Qalaji
Ethan Woollet

Table of Contents

Overview

Executive Summary

As technology becomes more and more ingrained in our society, it is very easy to allow oneself to sit inside all day and ignore the world around them, viewing and interacting with it via a screen. There have been several apps that try to encourage people to go outside and exercise more such as Pokémon Go and more recently Pikmin Bloom. Polyamanita is a project with a similar goal and also is an attempt to help users learn more about the natural world, specifically mushrooms.

This project has four major components: A Map for tracking mushroom locations, the ability to take pictures of mushrooms (“capturing” them) and determine what species they are, the ability to share those captured mushrooms with friends who use the app, and a journal for keeping track of found mushrooms as well as holding information about them. These four parts are the essence of our application. Together, they will provide the user with the information and encouragement necessary to start their journey on the path of mycology and of mushroom collecting.

Features of the App

- The map aspect of this application is a cluster map of their immediate area indicating where other users have found mushrooms. This will help to engage users on their current trek and help them to plan out future paths that they would like to explore. Users would be able to view the map and see nearby mushroom locations found by previous users as well as being able to add their own information to share with others about mushrooms that they have found.
- Users will be able to identify mushrooms by simply taking a picture of a mushroom and by using a machine learning algorithm that we have trained, the application will determine the type of mushroom in the picture.
- Once identified, the user can be directed to the journal page for that particular type of mushroom so that they can learn more about it. Users can also navigate to a mushroom’s page without needing to be directed to it once they have been discovered. The journal page would show the taxonomy, key identifiers, any fun facts, whether or not it is poisonous, and possibly recipes for the species where applicable.
- Users will be able to add other users on the app as friends and share captured mushroom data.

Polyamanita is structured technically into five separate fields:

- The front end app and web designs, including design and UX goals, aesthetic design choices, and technical limitations
- The API structure and functionality,
- The database foundation and schema,
- The machine learning models being used and their implementation, and
- Continuous Integration / Continuous Development infrastructure for streamlined app development and release

Front End

The front end documentation has two major components, the design specification and the technical specification. Beginning with the design specification, it is discussed about how the front end team wanted to make the application interesting and engaging to our users, and more importantly why. Then the document moves on to then how it will use those ideas to design and implement the app. This UI specification will make it clear for any front end developer to jump to and see how to implement the UI. As for the other component, the technical specification will deal with the software intricacies involved during development.

Referring back to the design specification, the process of developing the front end began with understanding who and why someone would want to use Polyaminita. By gaining insight on this user experience, the resulting application can be engaging and interesting for our users. Someone who would pick up the application is someone looking for an adventure, someone who wants to get out of their routine to appreciate and examine nature around them. The application is also for people who want to be proud of their journey; their journey of finding and “collecting all” the mushrooms relative to their region. By taking these ideas along with us into the design process; users with these goals will form habits and re-engage themself with the application.

The next part of the design specification deals with the UI and how each of the components UI elements should be designed. An example of the importance of this section would be about how the mobile will be developed around the React Native framework. As React Native is a component based JSX framework each component should be detailed on how it should be designed and where it will be used. This will help future developers to understand how and when to implement new UI features while matching current design standards.

Lastly, the technical specification for the front end explains the major implementation details and requirements. On a larger scale, the main component of the project, the mobile app, will be built using React Native with TypeScript, and the smaller web component will use React with TypeScript. These choices were made based on prior familiarity with the frameworks/languages and desire for easier deployment to multiple platforms/device types.

Also included in the front end technical specification are specific implementation details for different “functional” aspects of smaller areas or the front end; these will explain various instances of navigation specification, technically-complex screens and components, and anything else that we felt required in-depth detailing, most often for our own use/reference.

API

Due to the nature of Polyamanita the app will likely be used in areas of low or no internet connectivity. In anticipation of this the API calls will primarily be structured around the idea of communicating with the server in large batches rather than constant communication. This way the application handles all required communication with the server at home or places with strong internet connection so the user may freely utilize the main features without worrying about internet connectivity. The main calls that require access to the server would be account login, registration and updating a user’s journal with any new mushrooms discovered; none of which require constant access to the server and can be done in batches when user is connected to wifi or, with the user’s permission, has a signal for data.

Database

The Database will be created and accessed through Amazon DynamoDB. Several tables will be defined to store user credentials and metadata, to provide support for sharing stats with other users, and data sync across devices. Amazon Simple Storage Service is used alongside DynamoDB to store larger user data, for mushroom image syncing across devices.

Machine Learning

The machine learning algorithm we will implement is a Convolutional Neural Network (CNN), which is very commonly used in image classification problems like this. Since we are using a deep learning model, we will require a decently large amount of data to train the model on. Training data is a key challenge that we need to overcome. We plan

to collect and build our own database of images gathered from the internet, but this poses some challenges. We'll need to curate the images ourselves to remove any unrelated or outlier images. We want to gather one hundred images per species, with one hundred species in our list. With a two to one training and testing split, we should be able to train our model enough to gain a high accuracy. A major consideration is the content of the data themselves. We want to gather images that are similar enough to the real world data, however, we won't have enough time to be as thorough, so that will be left up to future expansions of the data and model.

For the purposes of machine learning, there will be a few key APIs we will end up using. The first, and most important API, is TensorFlow. This is the API that allows us to train a neural network to identify mushroom species simply by providing an image. However, TensorFlow is designed for use on a stationary computer or server, not mobile. Therefore, we will also need to implement the use of TensorFlow Lite, an extension of TensorFlow specifically for mobile devices. Additionally, to have the CNN work at all on different devices, TensorFlow Serving must be used. This is the final API we will be using from the TensorFlow team, and it allows us to use our trained model for production. Some of the other APIs required are Selenium, GBIF, GeoPy, Pandas, and NumPy. Selenium will be used to scrape the web for images to use for testing. GBIF is an API where we can get accurate and reliable mushroom data, as well as images. This is particularly important because we need to make our own mushroom dataset. GeoPy will be used to get information on where the user currently is located, with the intention of providing a better user experience (i.e. "heat map" of where mushrooms were found).

Continuous Integration / Deployment (CI/CD)

Polyamanita's Continuous Integration and Deployment structures are set in place to create developer environments that cause minimal friction between the team's work, code correctness, and an up to date product to users.

Polyamanita implements automatic workflows using GitHub Actions to verify code correctness and manage its distribution to production environments. Contributions to both front and back end codebases trigger actions to automatically run tests to thoroughly verify that the code is up to standard. This includes simple unit tests to capture component functionality, integration tests to ensure all systems of the code base function correctly together, and end-to-end tests to validate stability during high traffic loads. Additionally, containerization is utilized to create build images for the app, with functionality in place for automatic deployment to AWS environments with S3 and Elastic Beanstalk.

Broader Impacts

The team has felt that the rise of mobile phone popularity and the attraction of social media popularity has caused a decline in the need of being physical in the *real world;* being social and being around people. We hope Polyaminita allows our users to collaborate and have fun by going out into nature. Appreciating the environment becomes an important factor moving forward into the following decades because of the rise of concern regarding climate change. This is our second impact our app hopes to achieve. Lastly, we hope to educate those not familiar with the topic of mushrooms. Mycology is a broad and interesting field that we feel that any novice or expert should participate in learning about the vast phenomenon of mushrooms.

Terminology

Capturing a mushroom

Capturing entails the process of using the mushroom analyzer model to take a picture of a mushroom and log it into the database. Similar to a PokeDex, these mushrooms are similarly “captured” and logged in the journal.

Shroomalyzer

The Shroomalyzer is a shorthand form of describing the mushroom species analyzer feature that Polyamanita provides.

Trek / Trekking

Instead of the common phrase, “going out into the forest” or “taking a journey”, a concise word for these actions is with the word trek. To go on a trek, for example.

View

A view is an instance of content that the phone has rendered to the screen. Each view is distinct from one another. For example, a Profile view has an entirely different set of content from the Map view.

Motivations

Jan

My motivations for this project are quite simple: I want to make something that interests me. While pitching different Senior Design projects with a couple friends, I stumbled

upon a really neat website: mushroomcoloratlas.com. The website is simple, given some mushroom species, it will explain how one can make various shades of pigment or dye from said mushroom. Prior to the start of the semester, I was looking for a way to make a good, thin, dye to use in a fountain pen. This website definitely captivated me. I have always been fascinated with mushrooms, the sheer variety of colors and shapes spirals me into a world of wonder. So, when a teammate mentioned an idea of an app to identify different birds, I almost immediately thought of a mushroom identification app. This would also be more feasible to do than bird recognition. Since a lot of birds are quite timid, it would be hard to get a decent picture of one without an expensive camera. Additionally, in the heat of the moment, pulling out an app to do such a thing sounds quite difficult.

When this idea was first pitched, I was thinking about making a journal that shows you different use cases per mushroom the user finds (with an emphasis on the dye portion), but that seemed too easy and perhaps a little unimaginative. So, after tossing some ideas around we finally settled on the idea of a mushroom identification app that works kind of like a Pokedex; but, with some other features to make it more interesting. The idea we have been brewing for this project will not only test everyone's skills, but allow us to flex our muscles and create something we're all proud of.

Of course, to identify a mushroom through an image, you're gonna need to create a machine learning algorithm that recognizes the seemingly minute details of each individual mushroom. Awesome! I have always wanted to try my foot in machine learning, and that's part of the reason I am currently taking an algorithms in machine learning course here at the University of Central Florida. I will not only be preparing for the project as a whole in Senior Design 1, but gaining invaluable knowledge for what is to come soon in Senior Design 2. Additionally, we'll need to have a huge picture collection to train our model on. Luckily for my team, I have already dabbled with tools such as Selenium. Upon doing research, we found it to be increasingly more daunting to collect a solid and trustworthy bank of images strictly for mushrooms. All of them seemed to have critical issues. Personally, I have used Selenium for a previous job and thought it was quite interesting, I am just glad I finally have a proper use case for it.

Ethan

From a young age, I've always been interested in making things. I used to spend hours building things with all sorts of toy blocks, Legos, etc. By the time I graduated high school, I knew I wanted to get my bachelors. At the time, I was actually interested in

mechanical and aerospace engineering, but those plans were quickly crushed when I came to UCF and took my first engineering course. I immediately realized that I couldn't even pay attention because I was just disinterested in the material. It wasn't fun enough to even consider spending years of my life doing those kinds of things. In that same semester, however, I took my first introductory programming course in C, which completely captivated me. I was instantly hooked and changed my major the following semester. I haven't looked back since.

This project stemmed from an idea I had about my favorite group of animals, birds. I love birds, but it isn't easy to know what to look for when it comes to identifying them, so I thought "what if we made an image recognition program to identify birds?" I brought this idea up to Jan and Nawras when we were brainstorming for student project ideas for this class, and they reminded me that birds are fast and small, thus making it very difficult to get a good picture of one. Jan proposed a slight change to something quite a bit more stationary, and that's how we got to mushrooms.

While I don't particularly like mushrooms, I do love machine learning and using technology to aid in learning about the world. I think that many tasks can and should utilize machine learning as an aid or replacement for humans in tasks that humans are not suited for, and I would love to help with some of that development. This project will be my first deep dive into the field of image recognition and Artificial Intelligence (AI) as well as working on a project of this size. It may be relatively small in scale when compared to industry projects, but this is the largest project I have yet to work on and I hope that it will be a great learning experience and chance to develop my programming and team communication skills.

Amelia

I've really loved computer science from a young age, having started by writing Gamemaker games in high school and simple Java apps before having started at the University of Central Florida. Going through the undergraduate program for Computer Science had been thoroughly engaging and enjoyable on that part, creating a system in which I could express my ideas and learn significantly new material that I ever have before, and Senior Design was no exception.

My recent time I've been studying however has painted me jaded, however, as I had started working internships since last summer and throughout the last few semesters and became frustrated with the sheer discongruence present between academic classes and concepts discussed through there as opposed to the problems and skills I

had to develop to actually be a Software Engineer (SWE). I understand that Academia is not inherently obliged to teach students SWE skills, but my time in the industry has made me wish I had other avenues to harness and sharpen my SWE skills.

This is why I take Senior Design so seriously, as it's the only set of classes in the Computer Science curriculum that give exposure into the kind of work that most people getting a Computer Science degree will actually encounter for most of their lives. From specific software development ideologies like Agile methodologies, to common workplace technologies like Jira or Google Cloud Platform / Amazon Web Services, to even broader soft skills like teamwork and ownership, Senior Design allows me the opportunity to work hard to develop those skills and learn more about the fields of work I want to do full time.

I love working in back end technologies and using software design principles to orchestrate and structure solutions to large problems - Polyamanita provided me a really great opportunity to work with an effective team with near full control over the part of the project I'm working on, the API server. Given the complexity of the project I can also exert my skills learned in DevOp teams to create Continuous Integration workflows on a larger scale than any personal project I'd work on, allowing for multi-step processes that alleviate pain points my team and I would face when working in an environment that doesn't have structure in place to stop tedious parts of SWE work, like merge conflicts or building/deployment of our app.

Nawras

I had two main reasons for my interest in this project. The first reason was for its incorporation of machine learning, which I personally feel is a very promising field. This project allows me the opportunity to explore an amazing tool that holds so much potential for predictive software. Collecting and analyzing data is so valuable because it helps us understand what features influence different results,however going through massive volumes of data is very difficult for humans to do alone. Yet machine learning allows us to create technologies that make sifting through such huge reservoirs of data not only possible but valuable. Our project will also incorporate image recognition which is a subdivision of machine learning I am excited to learn more about as opposed to the more quantitative machine learning algorithms I am used to.

The second main reason was the app itself. A younger version of myself might have found the idea of walking and reading about mushrooms a bit boring but now it seems

quite fun since it gives an already enjoyable activity like hiking more purpose. While hiking through trails can be fun on its own, now I can also be on the lookout for interesting mushrooms. I'm not exactly sure why but the collecting phenomenon can become quite addicting, whether it's stamps, coins or cards for some reason people seem to really enjoy collecting things. Mushrooms themselves are also the ideal thing to find since they are quite varied and unique, not to mention once you find them they won't try to run away or bite you. It's also a nice way for someone like me who has no experience in something like this could easily download an app that not only journals all my findings in one place, but the app itself would also already contain information about what I find. This eliminates the need for any prior information or searching through online resources to try and figure out what I have actually found.

Tyler

I've always been a design and creative focused individual. It always seemed at a young age that I always honed in on the fine details of my articulations. Long ago and almost 10 years from now I took note of my abilities and graduated highschool to begin pursuing a degree in the Fine Arts with an emphasis in painting. It was there, I was able to study some incredible topics about light, color and composition. I am proud of my past experience studying fine art because I am a firm believer that they have helped me to be where I am today. It is now three years ago, I came across programming with an interest in creating and designing websites. As a result, I appreciated the coding part; more specifically it was the process and steps to take to express to a machine to do some action that interested me the most. It felt like magic. At this same time I was also pursuing an associates in a general studies program, and when I graduated, I immediately transferred to UCF in 2020 to pursue a degree on becoming a better technical problem solver.

I did not anticipate how much I have grown and learned at my time at UCF. Now heading into senior design I wanted to express to my peers and the world of my abilities completing the magnum opus project at UCF. I came across Ethan, from the machine learning team, pitching the idea of a mushroom collecting application; neat indeed. It piqued my interest and sent him some of my previous work, one involving a website built in vanilla JavaScript and another a desktop application built using the React framework on top of Electron. Ethan alongside Jan and Nawras liked my work, and I was brought into the team.

With my past experiences and interests, I am happy and confident alongside Layne developing the front end of the Polyamnita application. I think with the whole team's combined skills and chemistry, Polyaminita will be successful. This is what motivates me the most; being a part of this team, my past experience, desires, and its mission as an application.

Layne

My involvement in this project, like most things to which I resolve to commit this much time, thought, and energy, is motivated by a few different factors. First, having friends also working with me on an app is certainly a huge boost to the motivation I would have for contributing to any project. I often have difficulty finding initiative to sink time into projects like this, mostly because of a lack of real short-term motivation for making progress; in some previous experiences, the people and organizations I worked with were either relatively uninterested or simply too monolithic for motivation to be derived naturally. However, while working on this team, which consists of previous friends and acquaintances and new ones, I have found that it’s far more rewarding to make meaningful progress as it’s easier to get a real sense for how the work impacts the project as a whole when our communication is great and it doesn’t feel like a chore to reach out to my teammates.

As far as the actual role I have in the development of Polyamanita, I elected to participate in creating the front end based on my prior experience building, maintaining, and experimenting on UIs for various pieces of software, and especially from the experience working with Meta’s tech stack (including React) in my internship this summer. Since personal and school projects often involve close proximity to all aspects of a given application (back and front ends), it can be difficult, in my opinion, to get a feel for which aspects of the process you like and dislike. Having the chance to put a lot of work into just the front end of a website gave me a different perspective of the process, and let me know that I particularly enjoy implementing and experimenting with visual designs. When I had the chance to showcase my newfound abilities and interests for this project, as well, it was an easy decision. Additionally, the “aesthetic”, so to speak, of making a mushroom-based application is not lost on me.

Legality, Ethical and Privacy Issues

Several legal concerns surround Polyamanita, especially around the mushroom analyzer which serves as the focal point of the application. Detailed below are all

liabilities and ethical concerns around the scope of the Polyamanita, and the solutions we're employing to address those concerns.

Mushroom Analysis Reliability

The most critical legal concern around Polyamanita revolves around the usage of the Shroomalyzer to make predictions on mushrooms. The Shroomalyzer is primarily meant to serve as a guide that can easily integrate mushroom finding with an interface to track stats and provide quick information. As with most machine learning models, however, the Shroomalyzer is **NOT** perfect, and mushrooms will occasionally be marked as species that are different from the species they actually are. Without clarity on this detail, users may assume the app is always accurate on species matching and so could rely on it for making decisions on whether a mushroom is edible or safe to pick.

To ensure that the Shroomalyzer will be easier to understand for its intended use, the user will be reminded after capturing any mushroom that the captured mushroom may be incorrectly identified. Additionally, Polyamanita will use phrases that imply a non-absolute ability of the Shroomalyzer, so as to imply that the results aren't to be trusted for situations that could threaten the users health. Examples include:

- Using words such as "predicts" over words like "determines"
- Using phrasing that avoids asserting authoritatively over the user, such as describing the Shroomalyzer as a predictor or presenting the app as a softer, more informal style.

The machine model is in its efficiency and reiterate that eating and using a mushroom lies with the user decision alone; ***ALWAYS*** consult an expert and we are not responsible for maleffects.

User Location

The application will end up using the user's location to put mushrooms on a map. This way people can understand further where mushrooms are. Letting the user know about the location data collection will need to be understood on the user end.

Financing

The primary source of charges with Polyamanita will be with the back-end needs of the application. This includes using:

- DynamoDB as NoSQL database for the back-end server

    - Since DynamoDB is on-demand, costs are insignificant.
- Amazon S3 to store mushroom images and artifact image versions of the back-end server
    - Amazon S3 will cost approximately $0.05 a month per gigabyte of storage
- Elastic Beanstalk to host and maintain the application
    - Cloud run will cost approximately a minimum of $8.21 per month
- The total estimated cost per month would be $8.26 per month. Depending on how much storage we need or don't need we may reduce or increase the amount which will alter the cost.

These charges scale with increased use of the services, and at an introductory level server costs will be small.

Other financing that might be up for consideration is the use of outsourcing icons for art assets. These art assets will be mushroom icons to associate each mushroom on the application.

Design Specification

The design specification entails two topics of discussion, the UX Style and the UI Style of the Polyaminita application. In these two sections, their primary goal is to standardize the approach on solving design problems. In the UX Style, the topic of discussion is why the application is being built. Another takeaway with the UX Style is to show how the Polyaminita has grown from an idea to a proposed prototype. The other topic of discussion is the UI Style. With the decisions and feedback made in the UX Style, a standard can be made on what the application will look like. Its goal is similar to following a strict recipe in a cookbook where any person jumping to the front end to begin development can reference these *recipes* on how to implement these choices.

UX Style

User experience plays a huge role in understanding how and why the application will be used. Gaining insight on these questions, acts as a clairvoyance to the final goal of the project. This insight acts as a tool to gain empathy for our users. For instance what if some functionality or content was missing, how would they react or feel? By doing this research, we can gain insight on our users to meet their interests and needs. Without this discussion, it becomes difficult to understand the end goal of the application. Let’s begin with that research on who would pick up the application.

Research

The importance of user research is vital for a product's success as it essentially provides a foundation for designing a product. This research includes analyzing and observing what to create and why helps the product become relevant to the consumer. If the product is relevant to a consumer, then there is an increased chance for them to check out the app. A potential user checking out Polyaminita is looking for a need to be fulfilled. Perhaps they want to get outside and seek adventure; or keep a collection of mushrooms they come across. Understanding those needs of the consumer is imperative for product success, as it is the key of maintaining and growing a user base. Looking inward into the realm of psychology; we can design the product to meet the needs of the consumer.

User Personas

The development of user personas helps articulate and bring value to any feature wanting to be implemented. It also helps understand a specific set of users who could be using the application and meeting their needs on how and why they can use Polyaminita.

Down below is a list of user personas with their respective attributes: Persona, Motivation, Investment Level, End State. Their Persona is a title or label to provide context. The Motivation of the persona would be reasons why they would be using the app. The Investment Level is an arbitrary estimated scale on how the persona will be engaged with the application. Lastly the End State is what that persona will get out of using the application; fulfillment wise.

Persona	**Motivation**	**Investment Level**	**End State**
Mushroom Maniac	- Competing with other players because they enjoy it. - Experiencing a sense of progress.	High	- Competitive - Self Fulfillment
Social Explorer	- Getting the experience to go out and explore new places. - Be a part of something their friends are doing.	Low	- Social
Scholar	- They find mushrooms interesting enough to pursue acquiring knowledge regarding mycology. - Uses the application to enforce their interest.	Low	- Self Fulfillment
Social Circle	- A bunch of friends want to do something together and find new pursuits together. - Comparing each other's progress competitively.	High	- Competitive - Social

Feedback

Throughout the applications prototyping and development, collecting feedback from peers, friends, or even family can prove vital to creating an engaging application. Create time to meet with someone over the course of half an hour to an hour and discuss how they felt about certain features and design choices of the application. This feedback can also generate interesting ideas; simply asking features they would like to see integrated or to see how something was to change how they would feel about it.

User Flow

User flow diagram to indicate how a user can navigate across the application. Each “column” of each section can be treated as nesting deeper into the application.

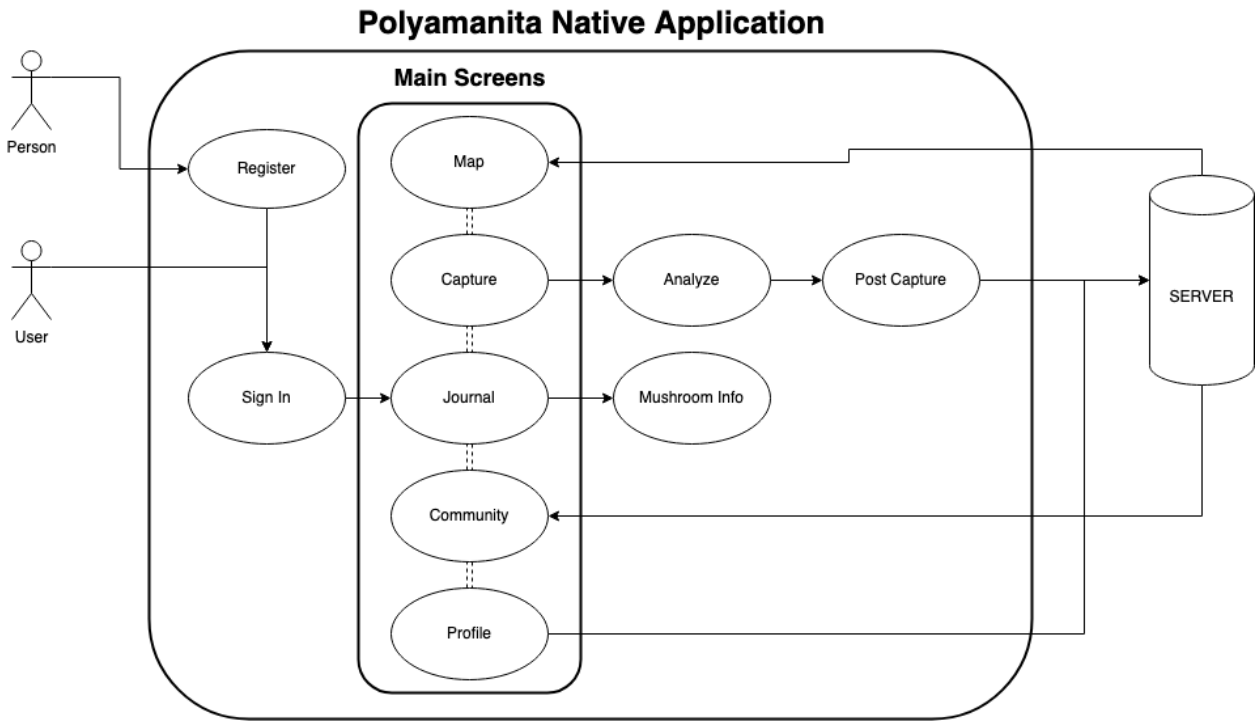


Figure 01: Mobile Navigation Sitemap

UI Style

In the above section, wireframes are used to show the user flow of the mobile application to see how a user can navigate across the application and understand why they are there. Now the layout is ready, it’s time to build on top of that. With the use of Figma and its tools, the branding and general appearance of the application can begin to be defined. Creating a seamless and packaged aesthetic is a priority. The goal of this section is to convey the following functions: consistency, shared vocabulary, onboarding and standardization in code and accessibility.

Accessibility

While building color patterns and components for the application, the topic of accessibility was in the team's interest. More specifically on vision impairment and perceivability. Color can convey subtle yet powerful information, but through vision impairments; that information is not received the same by everyone. Two examples could be color blindness and ocular distortions. Polyaminta expects their users to be on trek with the possibility of low light conditions from the phones screen, either a result of the bright sun, or the phones brightness is turned down to low to conserve battery.

Learning more about these disabilities and conditions can help increase the useability of the application and should be noted to front end developers. This isn't to restrict the creativity involved building a front end application but to add more value to it. There are standards to follow that are set with the Web Content Accessibility Guidelines (WCAG). These standards were developed through the W3C, to meet the needs of individuals worldwide.

WCAG Standard

One particular interest in Polyaminita design choices was to follow the WCAG AA standard at a minimum regarding color contrast. With the low light condition example that was previously mentioned, a high contrasting application can help mitigate any visual perception difficulty using the application. A helpful tool to determine if a design passes a color contrast test can be found using a contrast checker. The WCAG standard has algorithms to run a comparison of two RGB values but this website allows quick checking for this comparison. An example of this contrast checker:

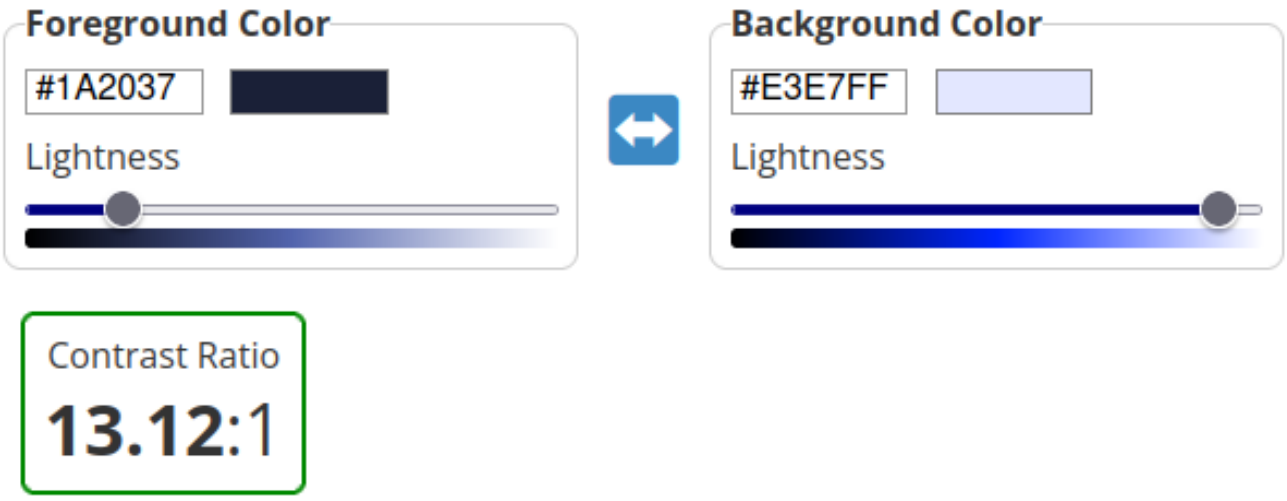


Figure 02: Contrast between text color and a background color.

A striking 13.12:1 ratio between these two colors, enough to pass WCAG AA and WCAG AAA requirements for readable contrast for: normal test, large text and graphical objects and user interface components. *Perfect*.

For making decisions on new colors, this table can be used as a quick reference to check if two colors contrast well:

	WCAG AA	**WCAG AAA**
Normal Text, 13px	>= 4.5:1	7:1
Large Text, 18px	3:1	4.5:1
UI Elements	3:1	

Further inspiration for web accessibility guidelines such as WebAIM, the World Wide Web Consortium, and others can be found here:

https://www.w3.org/standards/webdesign/accessibility
https://webaim.org/
https://web.dev/learn/accessibility/
https://webaim.org/resources/contrastchecker/

Color Impairment

Being diligent about color choice is also important. Those with visual impairments and color blindness can have a difficult experience distinguishing feedback with the use of color.

For a person without an atypical color vision, distinguishing between reds and greens are natural; but because of these impairments; the use of these color combinations to indicate something might be confusing or would not be as obvious.

For example, a warning is usually indicated as red and a success indicated as green.

Figure 03: Red-green colorblind example.

To compensate for these visual impairment problems, the front end can integrate some iconography to indicate further what the message or feedback is trying to convey. For example, input fields when signing up:

Figure 04: Using iconography alongside color

Color

This color section is a quick reference guide for the front end to reference color when needed. Instead of copying color codes in the codebase, the document standardizes these color choices with the goal to keep a continuity. Tables are layed out to help gain color information in a concise manner. Color IDs and labels are given for quick identification. While the colors properties are laid out for the developers favorite color channel numeric system: HEX, RGB, and HSB. Lastly, an alpha is given to define how transparent these colors will be as the current theme revolves around transparency between elements.

Brand

The Brand color palette involves the colors of the brand of Polyaminita, colors will be used in our logo and for marketing purposes. This palette also acts as a source to create other colors to apply to the UI elements.

Figure 05: The fly agaric mushroom, the *Amanita muscaria.*
The color influence of Polyaminita's brand.

ID	LABEL	COLOR	HEX	RGB	HSB	ALPHA
A01a	Primary		#E5176D	229, 23, 109	334.9, 89.9, 89.9	1.00
A01b	Secondary		#E4E8FF	228, 232, 255	231.1, 10.6, 100.0	1.00
A01c	Accent		#FAA54B	250, 165, 75	30.8, 70.0, 98.0	1.00
A01d	Positive		#008F6B	0, 143, 107	164.9, 100.0, 56.1	1.00
A01e	Warning		#D21947	210, 25, 71	345.1, 88.1,	1.00

Theme

The theme color palette involves the colors used for UI elements across the website and mobile app. These colors are in their raw form and often are mixed with either a set of layered low opacity colors or gradients.

A note on the ink “primary” and “secondary”. The use of these colors interchange bases on light mode and darkmode. Obviously, for a dark mode setup, we will use the set of ink secondary colors for lighter borders and text.

ID	LABEL	COLOR	HEX	RGB	HSB	ALPHA
A02a	Background A		#E3E7FF	227, 231, 255	231.4, 10.9, 100.0	1.00
A02b	Background B		#FAFBFF	250, 251, 255	228.0, 1.9, 100.0	1.00
A02c	Primary A		#e53981	229, 57, 129	334.9, 75.1, 89.8	1.00
A02d	Primary B		#E56D66	229, 109, 102	3.3, 55.4, 89.8	1.00
A02e	Ink Primary		#1A2037	58, 68, 228	233.1, 54.7, 50.2	1.00
A02f	Ink Secondary		#FAFBFF	250, 251, 255	228.0, 1.9, 100.0	1.00
A02g	Field Background		#D8DBF2	216, 219, 242	233.0, 10.7, 94.9	0.65
A02h	Field Disabled		#171A33	23, 26, 51	233.5, 54.9, 22.0	0.18
A02i	Snap Element Background		#434559	67, 67, 89	234.5, 14.1, 30.6	0.54
A02j	Link Color		#515FE1	81, 95, 225	234.7, 70.6, 60.0`	1.00
A02k	Link Hover		#DB0042	219, 0, 66	341.9, 100.0, 42.9	1.00
A02l	List Visited		#9C4F9C	156, 79, 156	300, 32.8, 46.1	1.00

Typography

This section is dedicated to set a standard in the use of fonts across the application to help reduce extra time replicating past designs and prevent font inconsistencies.

Downloading

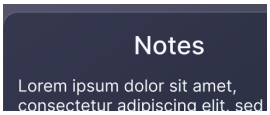
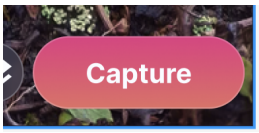
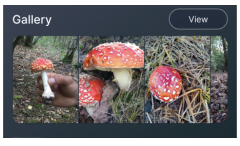
The font used within Polyaminita is the Inter font, and can be quickly checked out and downloaded from Google fonts:

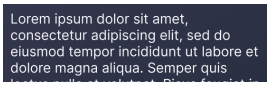
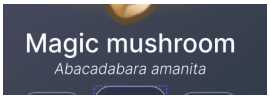
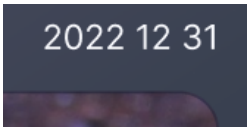
Google Fonts: https://fonts.google.com/specimen/Inter
Direct DL: https://fonts.google.com/download?family=Inter

Whereas a common understanding of these rights and freedoms is

Figure 06: Inter example text.

Defined Type Styling

Title	Size	Variant	Description	Example
Heading	28px	Semi-Bold	Use no more than once per view. This is only used for header navigation text.	
Title	24px	Medium	Used when titling content to help the user parsing the view. Also used titling new modal views.	
Call to Action (CTA)	18px	Bold	CTA text used on capsule buttons.	
Body-Large	18px	Medium	Section titles to help divide visual space.	

Body	16px	Regular	Body text, text for users to read in paragraphs or in lists.	
Titte-Sub	14px	Italic	Subtitle underneath the title. Useful when extra information is needed in relation to the Title.	
Body-Sub	13px	Medium	Auxiliary information, a small date for example within the section header.	

Use of Bolds and Italics

The use of bolds and italics should be restricted to only using the style or type of font from the table above. Going outside those bounds should be limited to prevent inconsistent type styling across the application. A use case for bolding and italicizing outside the restriction is within a body paragraph when it warrants it.

Iconography

The icon pack used across the Polyaminita brand will be the Google Material Icon Pack. Google's resource can be found here: https://fonts.google.com/icons

This icon pack comes with a large array of filled and unfilled icons, to also match our target mobile device operating system, Android.

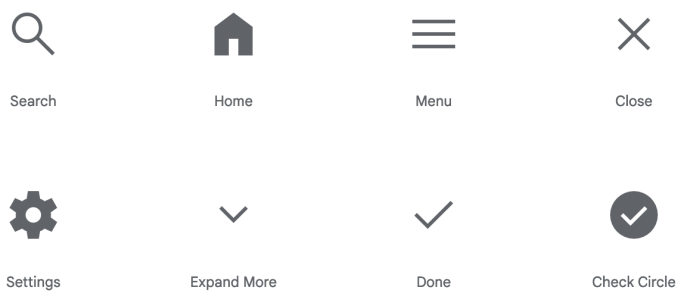


Figure 07: Google Material Icon Example

Mobile Components

The components needed for the mobile application will be contained here. Any front end developer can quickly reference the look and feel of these components. This is to both help both in development time and for creating new elements to match current design patterns. This section will also detail different variants of the same UI element; to help onboarding understand use cases when to use a certain element.

For more in depth styling implementation and components, all nuanced details can be found on the Figma document:
https://www.figma.com/file/ksepm92BnLnPDa7oryeea3/ui-design-(Copy)?node-id=212%3A834&t=jsqDKc1XHnvs9Wuj-0

Buttons

Anything dealing with pressable actions. On the left are capsule buttons which will be the “call to action” primary buttons and on the right are just buttons for functionality; we call them auxiliary buttons.

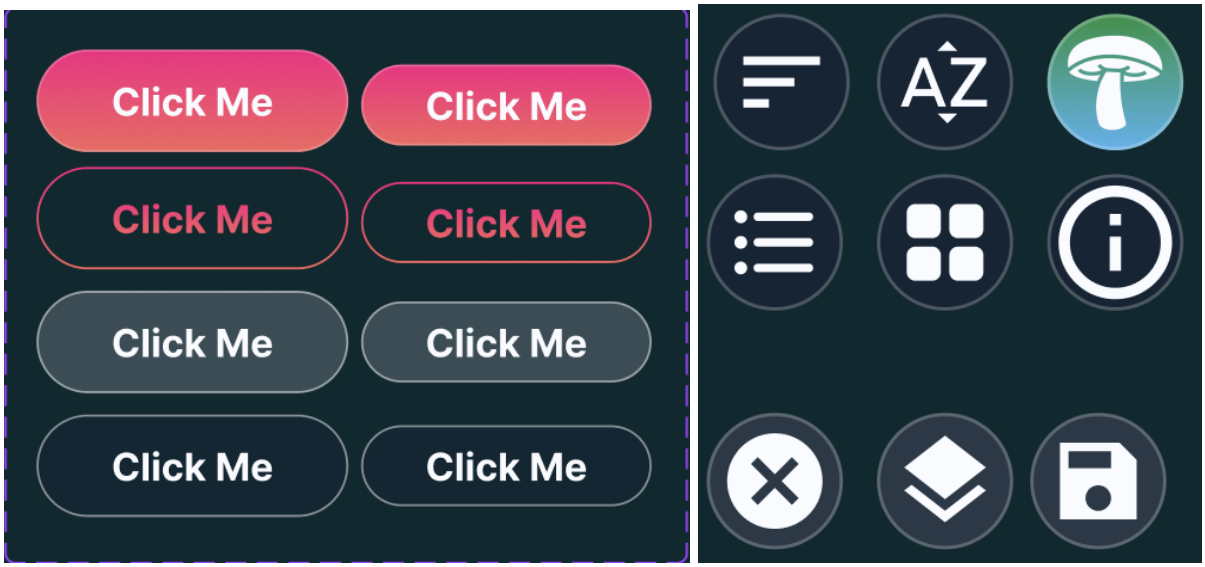


Figure 08: Capsule buttons (left), auxiliary buttons (right)

Below are the button’s limitations on where and when to use such buttons.

Type	Limitations
Capsule button (Primary)	**Only one** per page, and to be used only for actions you WANT the user to perform.
Capsule button (Primary-Outline)	**Only one** per page, and give the user an option if they don’t want to perform the action of the Primary capsule button.
Capsule button (Glass)	Glass variant used on top of an active camera or an image.
Capsule button (Default)	Use these by default if action is not required, a secondary option to perform on a view.
Auxiliary button (Default)	Give users further options to select and perform actions. These are actions beyond the MVP of the application.
Auxiliary button (Avatar)	Avatar icon representing the user's account, which will be visible across the application.
Auxiliary button (Glass)	Glass variant used on top of active camera or image, to provide users auxiliary options to perform.

Input Fields

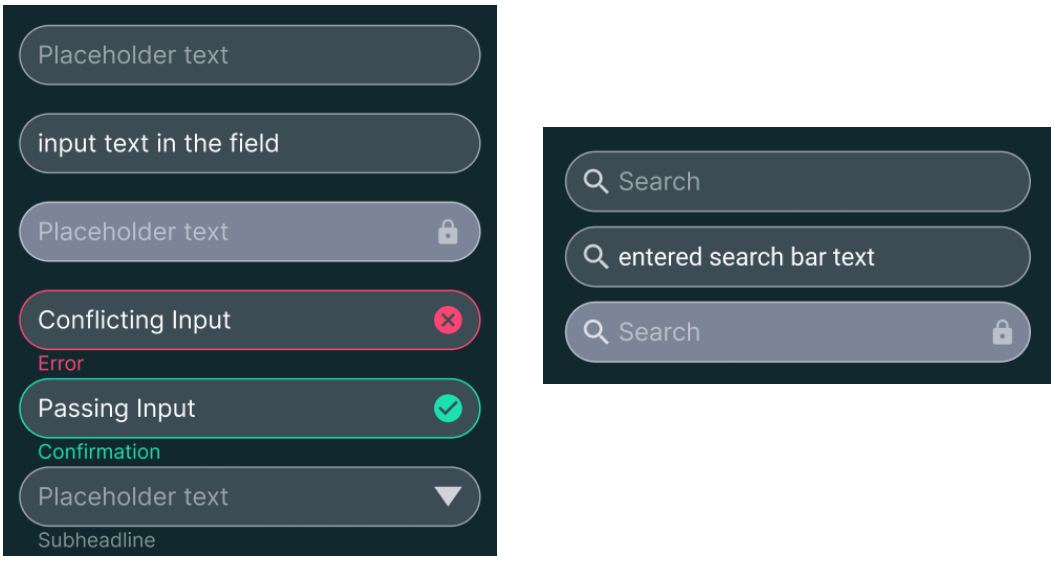


Figure 09: Input Field (left), Search Field (right)

Below are the input fields for different versions and where to use them.

Type	**Limitations**
Input Field (Default)	Default input field for users to provide info into a form.
Input Field (Input)	Removed placeholder and now displays inputted content.
Input Field (Locked)	For any instance we want to indicate a field is not ready for input, we have this option.
Input Field (Error)	Instance when feedback is returned to the user, red coloring and error icon appears with “broken” text field.
Input Field (Confirm)	Opposite of the error field, this is to give the user a confirmation that the field passes all confirmed cases that the field needs.
Input Field (Drop Down)	For an instance when a field can be typed in or alternatively can tap on the triangle to display a list of options.
Search Field (Default)	Default search field for users to type in a search.
Search Field (Input)	Text inputted into the search.
Search Field (Locked)	For a case to indicate to a user that searching is not available.

List Items

Figure 10: List Item (Left), Follower List (Right)

To provide context of how lists will look. They provide a way to insert an icon on the left of the field with a provided title. On the left of the figure, this is part of the mushroom journal where Common is just any mushroom title. While on the right is a follower list to showcase how else the list can be implemented.

Camera Elements

Figure 11: Camera elements.

Camera elements are unique from the application because their styling is built around the instance where when the camera is open the user should be able to still see the elements. A high contrast transparent elements provide a modern yet useful interface to work with. When adding and designing further elements, keep in mind to prevent adding too many elements. This provides the user more context to see the mushroom they are capturing.

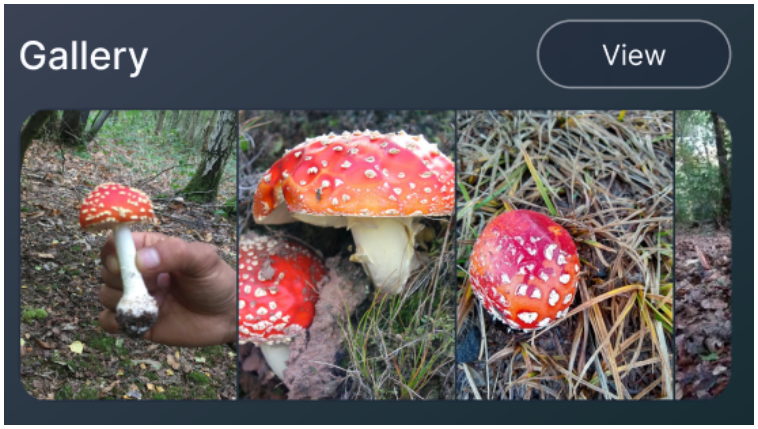


Figure 12:

Sections

Lastly possibly the beefiest component to deal with is the section. Sections are the application's way of dividing information across a view. On a profile view for instance, a user will have a section with a preview of their gallery of images (image above), another case is a section where to display a heatmap of mushrooms found in a specific region.

Front-end Specification

Platform-independent Technologies

Installation

The following are the steps necessary to set up a new development environment containing Node.js and TypeScript, in case a developer doesn't already have them installed. These are globally installed so that they can be used for multiple repositories/projects.

Install nvm (Node version manager)

```
curl -o-
https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.1/install.sh
| bash
```

Install latest version of Node

```
nvm install node
```

Assign the default Node version for new shells (in project folder)

```
echo "node" > .nvmrc
```

Install Yarn (alternative Node package manager)

```
npm install --global yarn
```

Install TypeScript

```
yarn add global typescript
```

JavaScript

JavaScript is an object-oriented programming (OOP) language designed and supported for website development/functionality, and is the single most popular programming language on both GitHub and StackOverflow. It is used in virtually all websites for client side behavior, and all major web browsers run JavaScript natively, making it a poor choice to avoid when developing web-based applications.

Though Polyamanita is primarily a mobile app, JavaScript is still the basis of all devices/platforms for our frontend UI functionality. There are many technologies and frameworks written for JavaScript that make it easier to write and enable its cross-platform usage, as will be discussed in later sections.

Node.js

Node.js (Node) is a runtime environment for JavaScript that enables the execution of scripts and programs divorced from a website/web browser. This is an imperative component of mobile applications, as although it is possible for mobile applications to simply run websites through some built-in browser, the results would be extremely tacky and disappointing. Node is also used by certain JavaScript frameworks even on website applications; in general, using JavaScript for a project implies that Node is also being used to some degree.

Additionally, Node comes with a sort of “package manager” called npm. npm enables the easy import and usage of third-party libraries/frameworks, and provides a universal platform for testing/running/building the application(s). npm is what is usually used to download and load the stack components that sit on top of JavaScript/Node. For Polyamanita, we are using the package manager Yarn instead, which provides some minor benefits and nicer syntax for our preferences. Acquisition of the necessary Node packages can be done using the commands outlined in the “Installation” portions of this section.

More simply, Node is needed to run and test the front end on our own systems during development, as relying solely on a web browser’s JavaScript engine to run and debug is not remotely conducive to efficient programming (though certainly some bugs and errors can be more easily diagnosable this way).

TypeScript

One major downside (and, indeed, a common complaint) of choosing to write code in JavaScript is its weak/un-typed nature. It can be difficult to know exactly what data types variables/objects will take until late into runtime with JavaScript scripts. For this reason, debugging/diagnosing the source of logical errors can be a nightmare and require heavy use of an IDE debugger or else manual tracing/debugging.

TypeScript is a language developed by Microsoft that offers a solution to this problem. It sits “on top” of JavaScript (and is compiled down to it upon building) and provides the ability to strictly specify data types upon declaration of variables/parameters, as well as create new object types and interfaces. If the TypeScript engine detects that something has been used incorrectly according to its specified type, it will produce an error message and prevent the script from compiling. This is similar to how other languages treat data type mismatches, which is a feature that makes development much easier, since we can now diagnose many problems via an IDE code linter before having to execute our code.

Essentially, TypeScript offers some benefits at little to no developmental cost, especially since our front end members are already intimately comfortable with using strictly-typed JavaScript.

Jest

Unit and end-to-end testing are part of the development of Polyamanita. Writing and running tests for the front end assists in attaining a smoother workflow as it becomes less important to manually test every change to ensure that all parts of the application still function as expected with no unexpected changes, or to simply ensure that individual UI components look and act correctly.

Jest is a testing framework (maintained by Meta) designed to offer a comprehensive, easy solution for writing and running tests. Though our experience with Jest is limited, the lack of truly viable alternatives virtually necessitates that we use it. Being used as the primary testing framework at Meta, Jest has configurations that work seamlessly with the UI frameworks we will be integrating; namely, React and React Native, which are both also maintained by Meta.

Jest also provides a kind of unit test called a “snapshot test” which allows for purely visual detection of changes to the UI across source code versions via comparisons between automatically-saved internal screenshots. While this test isn’t always particularly useful to determine whether components are displaying properly or not, it is an easy way to monitor for any unwanted changes.

Our Jest configurations/tests are included in Polyamanita’s GitHub repositories. To run the bundled test suite for a given platform, run the following command in the root directory of the appropriate repository:

```
yarn test
```

JSX

Since JavaScript is designed to run and operate on webpages, its standard libraries contain built-in classes, functions, and methods to allow for direct manipulation of the layout of webpages (referred to as the “Document Object Model” or “DOM”). However, dealing with the DOM directly in these ways can be cumbersome and produce hard-to-read code. Additionally, keeping the HTML and JavaScript separate (as is common practice with applications that use “vanilla” JavaScript) can prove detrimental to optimally organizing a web codebase.

Thankfully, the UI frameworks that Polyamanita use offer a syntax extension to JavaScript known as JSX. This allows HTML-like object layouts to be referenced and “called” directly from the JavaScript, using the same familiar HTML syntax. These HTML elements are then converted into traditional DOM object calls. The ability to simply generate DOM code with function-like syntax isn’t particularly important by itself, but emulation of the simple element hierarchy manipulation that HTML (and other markup languages) provides is an invaluable feature for organizing UI and staying sane.

However, JSX is not usable on its own. The syntax extension is bundled with React and React Native, and only in the context of those frameworks (or similar ones) can JSX syntax be used instead of normal DOM calls. React and React Native (and their relevance to Polyamanita’s front end) will be discussed in following sections.

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>This is an example of a JSX element.</h2>
  </div>
);
```

Figure 13: Example of JSX syntax

Mobile Stack

For the mobile app codebase, we employed a third-party “boilerplate” repository setup. This was in order to allow us to begin actual development sooner, and not have to focus as much on setting up folder structures, tool configurations and compatibilities, and prebuilt constructs and configurations for frameworks.

The boilerplate provided by GitHub user “WrathChaos” at https://github.com/WrathChaos/react-native-typescript-boilerplate is the specific one we utilized. The full list of included configurations and features can be found in the “What’s Included?” section of the repository’s README, but the main reasons we selected this boilerplate (as opposed to the default create-react-native-app, or another of the multitude of third-party ones) were its prebuilt navigation setup (using React Navigation, whose use is discussed later), premade light- and dark-mode theming system, and list of preinstalled packages/dependencies, including Axios, Flipper support, and custom icons.

The usage/setup of this boilerplate is not discussed in this document, as once the repository was created and initialized, it no longer required any more maintenance or consideration.

Installation

Required for new development environments.

Android Studio

Android Studio is the official Android development environment. The use of this software here involves the use of the emulator which comes with the SDK. This will provide context of how the application will operate without need for a physical phone to always be used, with the caveats and restrictions that come with that.

1. Android Studio can be downloaded from https://developer.android.com/studio
2. Agree with terms and download prompts.
3. Open Android Studio, and follow the installation wizard. Recommended settings are fine.
4. Finalize the wizard and wait for components to be added. When done, hitting Finish will open up Android Studio Automatically.
5. Keep it open, shift over to downloading the mobile application's repository to begin a working environment.

Local Repository

To get the mobile application development repository, open the terminal and go into your desired directory. We will clone the environment from the organization's set of repositories.

```
git clone https://github.com/Polyamanita/Polyamanita_Mobile.git
<project directory name>
```

After the repository is on a local system, we will need to install all npm packages into the project directory.

```
yarn
```

Now that the project is fully downloaded onto the machine, we need to get an emulator going to see the changes being made.

Emulation

Emulation provides context of how the application is handling itself.

Return back to Android Studio. Click on the **More Actions** dropdown, and select **SDK Manager.** This will open up the **Preferences** window. Under **Appearance & Behavior > System Settings**, click on **Android SDK.**

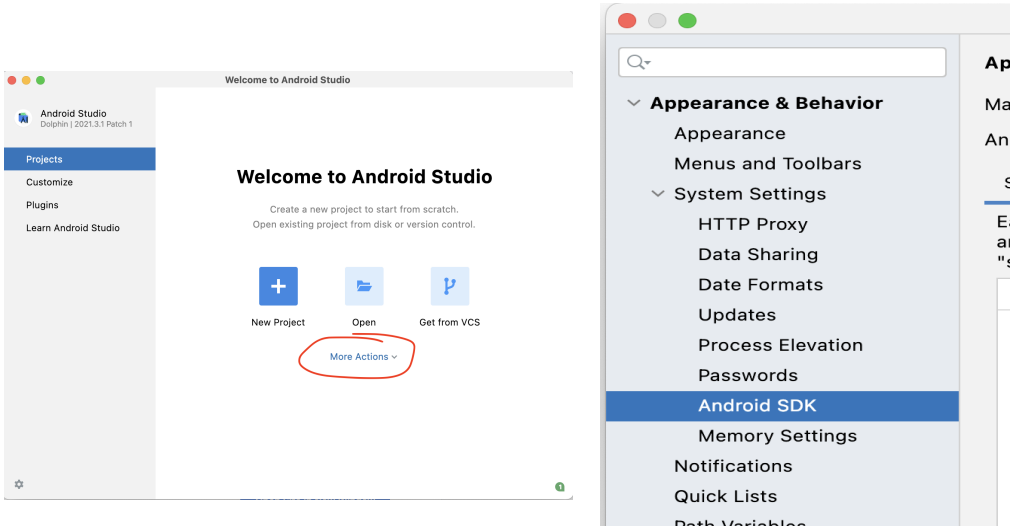


Figure 14: Selecting Android SDK

In **Android SDK** there are three tabs, **SDK Platforms, SDK Tools,** and **SDK Update Sites**. Under **SDK Platforms** is a list of platform versions that can be installed. We will need to download **Android 12.0 (S)** for its relatively new yet long term support.

This will also be your opportunity to take note of your **Android SDK Location**; save this directory path because we will need this later.

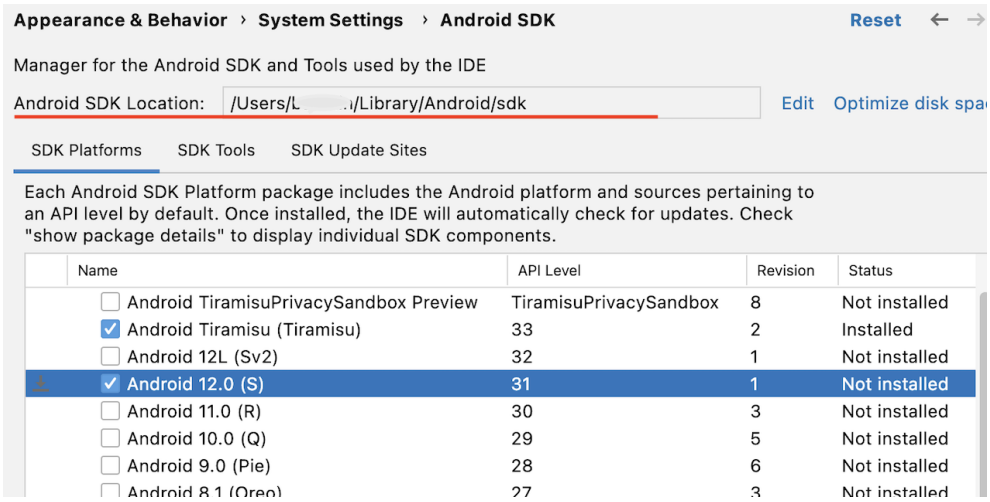


Figure 15: Adjusting Android SKD location

With the SDK now installed, we need to link up the working project directory with the emulator. This can be done via the following actions.

First we need to go into the android directory with the following command.

```
cd android
```

Find where the Android SDK is in your machine's filesystem: Android Studio displayed it earlier under the **Android SDK** window.

We use this command to echo this information into **local.properties**.

```
echo sdk.dir=<Android SDK Path> >> local.properties
```

One step to perform is to get the actual emulation device setup. Return back to the Android Studio and go to the welcoming window (the one which opens on initial startup of the application). Selected the **More Actions** again and now go to **Virtual Device Manager**. Create a new device with the following properties:

Device Definition: Pixel 5
System Image : S (API Level 31)

With these two settings we have an Android emulator. This device management window is how you can start the emulator.

Now with file path saved in local.properties and an emulated device, we begin a Metro instance. Metro is a bundler which will combine all files into one and translate any JavaScript code that the device won't be able to understand, such as JSX, and is already a part of the project as a Node module. We can begin this bundler with the following command:

```
yarn start
```

With the Metro bundler running, it is required to open another terminal **in the same project directory** and begin a build for the emulator to run. **Make sure Android**

Emulator is running before running. The next yarn command checks for an existing instance of an emulator to which it can attach; if one is not found, it may not work.

This creates a build, which requires the **Java Runtime**. For React Native to be "compiled" with Kotlin, **Java versions < 12.x** are good. Versions higher are not supported for Gradle to handle. For more information about installing Java SE Development Kit, see https://www.oracle.com/java/technologies/downloads/#java11.

```
yarn android
```

React Native

For the development of Polyamanita's mobile application, a slightly different toolset must be used. Obviously, using "vanilla" React is out of the question, as the DOM manipulation performed by standard React function calls cannot be applied to mobile device interfaces/renderers. Additionally, writing the UI from scratch using purely native device APIs would be a nightmarishly tedious and more involved endeavor for the scope of this project, as these can vary a lot between devices and operating systems and hinder us from effectively designing Polyamanita with cross-platform functionality in mind.*

A solution that serves to convert familiar and intuitive code into these lower-level native device calls/patterns is then clearly something we require for reasonably-paced development. Fortunately, there are a few existent frameworks/libraries that provide exactly this for mobile developers. The one we've chosen to use for Polyamanita is React Native, which, as the name would suggest, is derived directly from the web-based React.

React Native is indeed very similar to React in many ways. It uses the same JSX syntax with component separation and functional hooks, the code design paradigms are also essentially the same where applicable, and less importantly it is also maintained by Meta. However, React's default component library is obviously mostly unusable, and controlling navigation within and between screens/components is also a meaningfully different process on the developer's side.

As an example of React Native's main differences, as opposed to something like a in React, which is "compiled" and transformed into JavaScript DOM calls, a <View> (and any other component) in React Native is translated into native mobile UI

calls depending on the platform; this isn't done through the source code in real time, but rather done through pre-compiled Android and iOS code generated during React Native's build process.

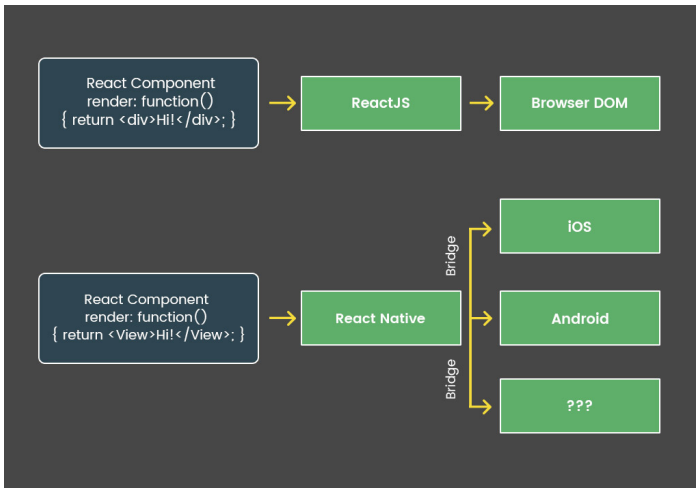


Figure 16: Diagram comparing the generic rendering order of React and React Native. Credit Naiya Sharma, dzone.com

Following is a discussion of some important React features used in Polyamanita's code, as well as React-Native-specific libraries used by Polyamanita's mobile app for various bits of UI functionality and features.

Components

The JSX syntax allows not only the use of standard HTML elements, but also imported or user-defined elements, called "components". These components can be called in the same way as the base elements and will nest their content within the tree; basically, the complexity of the underlying code doesn't change, but the visual complexity can be easily decreased. This is similar to using functions in code to reduce the complexity; in fact, one way to define a component (which is also currently the preferred way) is by declaring a function that returns a JSX element.

```
const MyButton = () => {
  return (
    <button>
      Push me :)
    </button>
  );
}
```

Figure 17: Example creation of a component “Button”

```
export default const App = () => {
  return (
    <div>
      <h1>Hello!</h1>
      <MyButton />
    </div>
  );
}
```

Figure 18: Example use of the new component

Appropriately segmenting Polyamanita’s app and website into basic, “atomic” components will greatly reduce the size of source files in its codebase, at the cost of increased size of the codebase itself due to the greater number of individual source files. However, the organization of components (and other source code files) within the project file directory can be made acceptably systematic and hierarchical.

Props

React components further adhere to this comparison with traditional functions via the usage of “props” (short for “properties”). Props allow for values to be dynamically passed to the called components and used within the component’s code, so that even portions of code that include variables or values that need to change can be condensed. In this sense, they are akin to function arguments/parameters in that they allow for dynamic reusability of UI code. This is especially helpful when it becomes a requirement to render an unspecified number of element trees (i.e. an array of components) with different data for each.

```
export default const Profile = () => {
  return (
    <Avatar
      person={{ name: 'Lin Lanying', imageId: '1bX5QH6' }}
      size={100}
    />
  );
}

const Avatar = ({ person, size }) => {
  // person and size are available here
}
```

Figure 19: Example use of component props

Something to note about creating and using components is that React currently has two “competing” code design ideologies: whether to use “class components” or “functional components”. As can be imagined, this means that it is possible to either use OOP classes to represent single components, or functions as we have shown in this section so far (specifically arrow functions, an alternative syntax for functions in JavaScript) to do the same. Each has advantages and disadvantages, but the front end of Polyamanita will be implemented using functional components, for the sake of familiarity and ease; additionally, the current latest version of the React documentation (at https://beta.reactjs.org/) uses functional components in all its code and examples, so consulting the docs is made easier by use of functional components.

Hooks

Since February 2019, React has had a type of top-level function called a “hook”. Today, there are 15 of these hooks provided by default in the React library, and developers have the ability to write their own custom hooks, either using combinations of those default hooks or their own implementations. Hooks provide ways to maintain component states, trigger re-renders, and perform other important tasks necessary for more complex UI functionality. Code involving hooks usually comprises the majority of the logic in React applications.

```
const MyButton = () => {
  const [count, setCount] = useState(0);

  const handleClick = () => {
    setCount(count + 1);
  }

  return (
    <button onClick={handleClick}>
      Clicked {count} times
    </button>
  );
}
```

Figure 20: Component example with the `useState` hook

React Navigation

Since React Native is indeed working on a fundamentally different UI basis than React, a different set of interfaces for inter- and intra-screen navigation is necessary. React Navigation is a third-party library built for React Native that provides a variety of such native-compatible navigation components, containers, and patterns, and is the one Polyamanita's front end uses; like a majority of our front end stack technologies, it was chosen for its popularity, support, and familiarity to us.

A brief overview of each major concept type of navigation container follows, for the sole purpose of providing context for other sections of this document in which screen-specific implementation details are given. Code examples may be given in those sections to introduce exactly how we are using each tool for a given app screen/page, but will mostly just be quick, general references for how we planned to implement the navigation patterns.

Just as React and React Native allow for sharing state and other information between components in an app hierarchy through the use of component props, React Navigation extends this capability to allow a subset of what can be passed through props to be passed via "route params" when navigating to other screens. Though this method of cross-component sharing is indeed a bit more restrictive, it can still be a powerful tool for rectifying hierarchical issues, or serve to reduce complexity if used correctly.

The Stack Navigator and Native Stack Navigator place contained screens in a stack (i.e. the data structure), which allows for the native phone "back" control (this is different between different phone models) to navigate back out of entered screens as opposed to relying solely on custom in-app buttons to return to previous screens. This type of navigator is used for moving "deeper" into a set of related screens.

Drawer Navigators provides an easy way to have a hidden-by-default side “drawer” with links to a few different pages, usually accessible from multiple screens. Polyamanita (and other apps) use this type of navigation to provide quick access to multiple common but possibly unrelated sections of the app.

A Tab Navigator is used to render an always-visible (usually) set of horizontally-laid tabs (on either the top or bottom of a screen) for arguably the simplest and easiest navigation method between a group of screens. Given the screen real estate taken up by this navigation UI, this is designed to be used to group related screens, or to simply provide a constant way to get to the most important pages of the app.

Redux

For certain actions, such as storing user information on sign-in or updating the Journal across app screens, manipulating a global application state is an incredibly important task. While React’s Context API is technically a way to achieve this, it still relies heavily on component hierarchies in a traditional DOM sense, and does not easily allow for state updates. Redux is instead the popular choice for managing state, and is the library we use to do so for Polyamanita.

Our Redux setup allows us to perform two basic action templates on the global state (called the “Redux store”): “selecting” and “dispatching”. Selecting involves querying the Redux store for state data that is currently stored within it, and is done via the `useSelector` hook. Dispatching involves sending a signal (and any necessary data) to the Redux store to update a state value, and is accomplished through use of the function returned by the `useDispatch` hook. Dispatching also requires specific action tags and functions to be predefined in the code.

The Redux code itself is regrettably somewhat complex and verbose; see the src/redux directory for the code to refer to when designing new global state actions.

Project Structure

This structure is dedicated to file structure and where things are located within the project directory. Alongside any useful descriptions for aforementioned directories and methods from libraries like Axios.

File Structure

`index.js` is the entry-point for our file, and is mandatory.

`App.tsx` is the main-point for our application. We can store global variables here, thus the Redux store and lightmode/darkmode variables wrap the entire app in this file.

- `/android` - contains native code specific to the Android OS.
  - `build.gradle` - Top level build file where configuration options to all modules of the application.
  - `/app` - Native Android Polyamanita code
    - `build.gradle` - Specific build modular configuration for Polyamanita.
    - `/src/main`
      - `/assets` - Context of files not native to Android are placed here. The tensor-flow model, fonts, and mushroom labels reside here.
      - `/java/com/polyamanita` - native modules to run on the front end. Shroomalzyer code sits here.
      - `/res` - native application icons and splash screen handling.
- `/docs` - as the name suggests - any docs.
- `/node_modules` - React Native packages added with yarn package manager.
- `/src` - contains rendered JS and style code. This is the front-end.
  - `/api` - generic network handling with API constants
  - `/assets/` - non-code files, like icons, images, fonts, etc.
  - `/navigation` - react navigation handling, components and constants.

`/redux` - redux store, reducers and constants.

`/screens` - contains all screens/pages of the application. (detailed section in next section)

`/shared` - whole app-wide shared

`/components` - app-wide shared components

`/constants` - app-wide shared constant variables

`/localization` - app-wide localization

`/theme` - app-wide theme which contains `color palette` and `fonts`

`/wrappers` - app-wide component wrappers

`/storage` - anything dealing with file input/output, searching files, etc.

`/utils` - generic util functions

`index.js` - the starting place for our app

`App.tsx` - the main place for our app

`.babelrc` - local configuration file for importing modules across the application. Cleans up import statements.

`.commitlintrc.json` - a git commit linter

`.eslintrc.js` - a set of rules fixes syntactic problems across applications.

`.prettierrc` - Prettier rules that formats code to enforce code styling consistency.

`babel.config.js` - instantiates context from native module plugins to use on front end (like the Shroomalyzer).

`package.json` - Node modules and versioning.

`README.md` - Remote repository text document.

`tsconfig.json` - TS rules and babel filepath context for TS

src/screens Structure

For each screen on the application, the current structure has it so where each unique screen gets its own directory. The only case of not following that convention is if there is a complete similarity to screens to prevent clouding up ./screens with similar screens.

Each screen can have the following sub-directories.

`screen.tsx` - screen component

`screen.style.ts` - styling for screens sub-components

`utils.ts | /utils` - module functions that run on the screen.

`/components` - organizational sub directory to put uniquely created components.

`/wrappers` - organizational sub directory to put uniquely created component wrappers.

Axios

All of Axios's functions/methods can be found at `./src/api`. This section will be useful to get a general and quick idea on what the front end's communication is doing towards the API.

Axios instance

Axios.create is an easy way to instantiate axios and create a consistent configuration for your API calls.

in `./requests.ts`

```
const instance = axios.create({
 baseURL: BEANSTALK_URL, // secrets found in ./secrets.ts
 timeout: 7500,
})
```

We can also get creative by declaring an object that handles all Fetch API call methods:

GET, **POST**, **PUT** and **DELETE**.

```
// Set of general Axios HTTP request functions.
const requests = {
 get: (url: string, params?: unknown) => {
   return instance
     .get(url, { params: params })
     .then(result)
     .catch(handleError);
 },
 post: (url: string, body: unknown, params?: unknown) =>
   instance
     .post(url, body, { params: params, headers: contentJSON })
     .then(result)
     .catch(handleError),
 put: (url: string, body: unknown, params?: unknown) =>
   instance
     .put(url, body, { params: params, headers: contentJSON })
     .then(result)
     .catch(handleError),
 delete: (url: string, params?: unknown) =>
   instance.delete(url, { params: params }).then(result).catch(handleError),
};
```

API constants

List of interfaces and classes to use for API calls.

```
// Credentials to verify a session. If valid, API returns a session token.
interface Session {
 email: string;
 password: string;
}

// To authorize a user a username, email, and pass required.
interface AuthUser extends Session {
 username: string;
}
```

```
// To confirm a user has signed up by typing 4-digit verification, we can create a new user officially.
interface NewUser extends AuthUser {
 code: string;
}
```

Methods

List of API calls and their use. All methods created here are of `Promise<any>`. So the developer may create a decorator to extend these methods if they wish to do so.

doRegister [POST]

@params: `AuthUser.` Server side handles objects to send an email verification to email.

doAuthorize [POST]

@params: `NewUser.`Object that confirms the code sent to users email. Upon success, this creates the account.

doSignin [POST]

@params: `Session.` Create a session/cookie for user sign in.

doSignOut [DELETE]

Delete current session and cookie.

doGetUser [GET]

@params: `userID: string.` Useful when we want to get user information of any user by providing their userID.

doUpdateColor [POST] <experimental>

@params: `userID: string, colors: [color1: string, color2: string].` Updates the color for the current user.

doGetCapture [GET]

@params: `userID: string, captureID: string.` "TODO"

doGetAllCaptures [GET]

@params: `userID: string, captures: Captures.` "TODO"

doGetUploadLinkAndS3Key [POST]

@params: `userID: string.` Retrieve a new upload link to post the image and its s3key.

doUploadToS3 [PUT]

@params: `imageURI: string, uploadLink: string.` With an image URI from the phone, converts it to a blob, then uploads blob as the body of PUT request.

Here's a direct current version of our Axios requests for the mobile application. It provides an easy interface to setup communications to endpoints.

```
const instance = axios.create({
  baseURL: BEANSTALK_URL,
  timeout: 7500,
});

const result = (res: AxiosResponse) => res;
const contentJSON = { "content-type": "application/json" };

const handleError = (err: AxiosError) => {
  if (err.response) {
    return err.response;
  } else if (err.request) {
    return err.request;
  } else {
    return err.message;
  }
};

// Set of general Axios HTTP request functions.
const requests = {
  get: (url: string, params?: unknown) => {
    return instance
      .get(url, { params: params })
      .then(result)
      .catch(handleError);
  },
  post: (url: string, body: unknown, params?: unknown) =>
    instance
      .post(url, body, { params: params, headers: contentJSON })
      .then(result)
      .catch(handleError),
  put: (url: string, body: unknown, params?: unknown) =>
    instance
      .put(url, body, { params: params, headers: contentJSON })
      .then(result)
      .catch(handleError),
  delete: (url: string, params?: unknown) =>
    instance.delete(url, { params: params }).then(result).catch(handleError),
};
```

Figure 21: Axios custom request functions setup

VisionCamera

The mobile front end for Polyamanita requires the ability to utilize a device's camera to take photos of mushrooms. This is also immensely difficult to accomplish with stock React Native components/capabilities, so we are again using a third-party library to provide this functionality.

react-native-camera was once a popular third-party choice for accessing native phone capabilities but has been deprecated due to lack of maintenance and increased code complexity; VisionCamera is instead our library of choice for taking photos. It offers a vast array of new APIs with better performance and improved stability. This library requires **iOS 11 or higher**, and **Android-SDK version 21 or higher**.

To begin using the camera, the Polyamanita app needs to gain native device camera permission. VisionCamera provides a very easy way to prompt the user for this permission, simply using a static asynchronous method from its library.

As modern mobile phones have increased their photographic potential, more separate physical cameras are added to the devices themselves; one needs only to look at the back of any iPhone released in the past few years to confirm this. The existence of multiple cameras is handled expertly by many mobile software developers, on both native and third-party applications, and many end users are likely unaware at a given point in time as to which camera is actually being used by an application (for example, stock camera apps can use all of the cameras depending on the photo setting, but users may only have general info about what kind of photo they're about to take, such as zoom level, exposure, FPS, etc. and aren't made explicitly aware that a specific lens corresponds to a specific setting). However, it is often useful (and/or necessary) for an app to force the utilization of a specific physical or virtual camera. This, too, can be noticed in a number of mobile apps, where there are slight differences between their non-modifiable camera interfaces. VisionCamera fortunately provides methods for querying the list of available camera devices, listing and analyzing their properties, and selecting one that the app will forcibly use for the best results. Currently, the app uses this mechanism to simply query and launch the main back camera, which defaults to an appropriate zoom level.

All normal camera functionality is accessible fairly simply with VisionCamera's main camera component via a set of hooks and methods. Further documentation and examples can be found at https://mrousavy.com/react-native-vision-camera/docs/guides and https://mrousavy.com/react-native-vision-camera/docs/api.

react-native-maps

react-native-maps is a popular choice for implementing a map and its components for iOS and Android. The added ability of rendering custom markers makes it a good choice for implementing the user interface on the map.

More information can be found at https://github.com/react-native-maps/react-native-maps.

Mobile Screens

This section details implementation specifics for each screen of Polyamanita's mobile application. It will enumerate exactly which components and techniques are being used to achieve the main functionality within the screens, as well as discuss any other concerns (apart from previously-discussed design or UI concerns) that needs to be addressed.

This section also served as reference for those "building blocks" that are utilized. If something is mentioned for the first time, a more detailed breakdown of exactly how it is used (and how we plan to use it) will be given, such that anything that needs further explanation is guaranteed to get it. The alternative to this method of documenting important implementation techniques is to instead use contrived examples in the previous section(s) and provide incredibly general examples and instructions; we originally chose to document it this way instead partly to avoid oversaturating the previous sections of the document, but mostly to assist ourselves in actually writing the code that we need to.

Start Screen

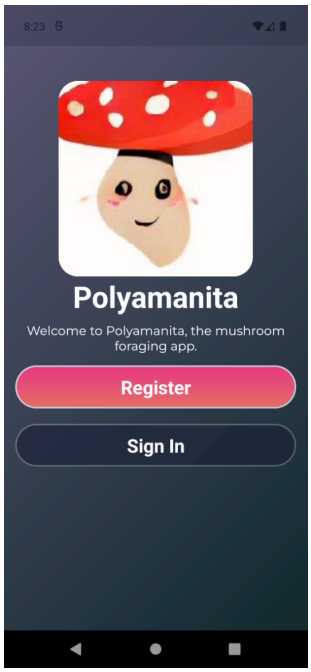


Figure 22: Screenshot of Start screen for reference

This screen is the first one a user will see if they haven't logged in yet (or their session token has expired and they need to generate a new one), and allows them to navigate quickly and easily to the sign-in and register screens using the two buttons. The nuances of the requirements for this screen (alongside the sign-in, register, and confirmation screens) are discussed here.

The code responsible for the Start screen can be found in `src/screens/initial-screens/start/start-screen.tsx`.

Navigation

Predictably, the button labeled "Sign Up" brings the user to the registration screen, and the one labeled "Sign In" brings them to the sign-in screen. Because we wanted the user to have the ability to easily and consistently navigate back out of those two screens to this one (and thus have the ability to travel freely between the two screens), a Stack Navigator is used. Usage information and code examples regarding this Stack Navigator are included here.

```
import StartScreen from "@screens/initial-screens/start/start-screen";
import RegisterScreen from "@screens/initial-screens/register/register-screen";
import SigninScreen from "@screens/initial-screens/signin/signin-screen";
import ConfirmScreen from "@screens/initial-screens/confirm/confirm-screen";
import { SCREENS } from "shared/constants/navigation-routes";

const Stack = createStackNavigator();

// This navigation stack contains signing in and registering related screens.
export const InitialStack = () => {
  return (
    <Stack.Navigator screenOptions={{ headerShown: false }}>
      <Stack.Screen name={SCREENS.START} component={StartScreen} />
      <Stack.Screen name={SCREENS.REGISTER} component={RegisterScreen} />
      <Stack.Screen name={SCREENS.SIGNIN} component={SigninScreen} />
      <Stack.Screen name={SCREENS.CONFIRM} component={ConfirmScreen} />
    </Stack.Navigator>
  );
};
```

Figure 23: Code for the initial screen navigation stack, found at `src/navigation/stack-navigations.tsx`

In this snippet, `StartScreen`, `SigninScreen`, `RegisterScreen`, and `ConfirmScreen` are the components responsible for rendering each of the respective screens, and `START`, `REGISTER`, `SIGNIN`, and `CONFIRM` are the imported constant strings we supply to React Navigation for internal use. To navigate to another screen, the `navigate` function, which is contained in each component's "navigation prop" (a prop that is passed automatically to components nested inside navigator components like this), is called using the `name`s provided to the navigator component. Usage of this function will be discussed later.

Note that there is no explicitly-stated "initial" or "first" screen in this navigator's code. This is because while React Native must indeed always prioritize one screen in a navigator, it does provide default fallback behavior if an initial route is not specified via the `initialRouteName` prop on the `Navigator` component: the first child of the `Navigator` becomes the initial route. For our purpose, we want the Start screen to be the initial route anyway, so simply having it be the first one in the list of `Screen`s within the `Navigator` is sufficient.

By default, a native Stack Navigator will utilize the "native" behavior of device navigation and render a default header bar with a back button, though a device's hardware back

button or universal screen gesture will also work to navigate back to the original screen. However, it is possible (and indeed necessary to achieve visual parity between the design mockups and application prototype) to easily bypass this behavior using the boolean `headerShown` option, as in the example. Placing this parameter within the `screenOptions` prop of the outer `Navigator` causes the option to be applied automatically to each screen nested within. This prevents React Native from placing a header on those screens, though indeed also removes access to the back button within it; this inability to natively navigate back is remedied by the design of the Sign-in and Register screens, which feature “Cancel” buttons that behave identically. We also still allow the hardware (or virtual) back button or gesture on a user’s phone to be used to return to the landing page in the same way.

Other Stack Navigators used in the application may have different props or options supplied, depending on the requirements informed by the mockups or other specifications. Those differences are discussed as is appropriate, and all of the application’s various Stack Navigators are found in `src/navigation/stack-navigations.tsx`.

Permissions

Even before the landing/start screen appears, on app launch and if the necessary permissions for the app (camera, location, and storage) are not yet granted, the app will prompt for these permissions. This is done via the `PermissionModal` component defined in `src/screens/initial-screens/permissions/permission-modal.tsx`.

To accomplish the native permission requests, the React Native `PermissionsAndroid` module is used. We feed the `requestMultiple` function a `Permission` object array, which we have defined as a constant to include the three aforementioned permissions, and deal with the resultant promise resolution and rejection to determine whether the permissions were successfully given. Screenshots and a code sample are given below.

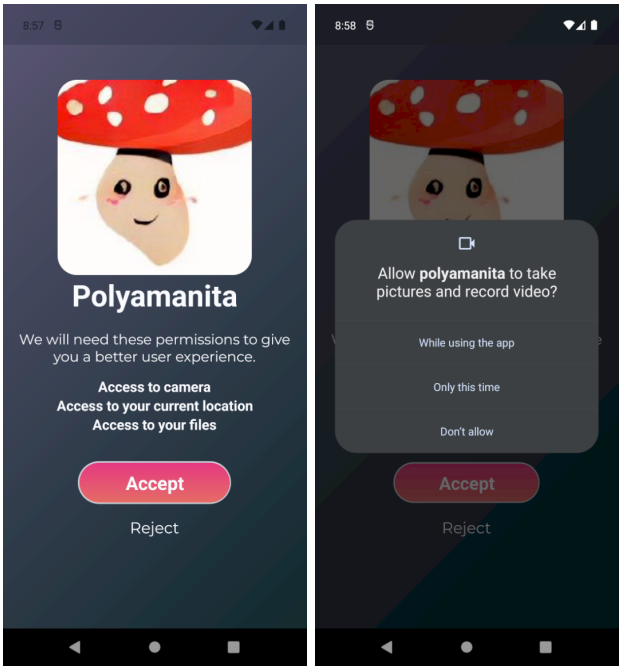


Figure 24: Screenshots of the permission request popup

```
PermissionsAndroid.requestMultiple(
  permissionFinalizer as Permission[],
)
  .then((_) => {
    // Check if all were set to true.
    navigation.navigate(APPSECTIONS.INITIAL as never);
  })
  .catch((rejectOnGivingPermissions) => {
    console.error(
      "Something really went wrong giving permissions.",
      rejectOnGivingPermissions,
    );
  });
```

Figure 25: Code for the initial permission request

Sign-in

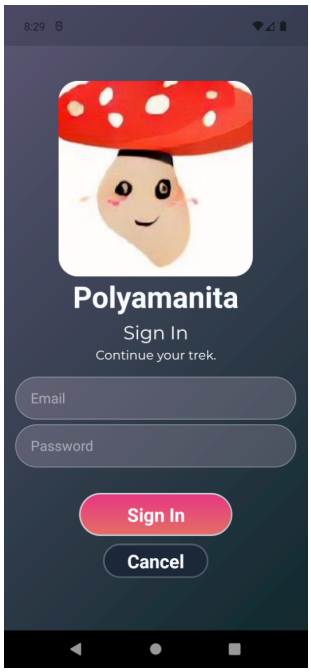


Figure 26: Screenshot of Sign-in screen for reference

The code responsible for the Sign-in screen can be found in `src/screens/initial-screens/signin/signin-screen.tsx`.

Navigation

This screen is reached by tapping the “Sign In” button from the Start screen. The way this is done is through use of the `navigate` function, as previously mentioned. Example usage of the navigation prop and this function is shown below.

```
interface StartScreenProps {
  navigation: StackNavigationProp<ParamListBase, string>;
}

const StartScreen: React.FC<StartScreenProps> = ({ navigation }) => {

  // ...

  return (
    // ...
      <Button
        title="Sign In"
        onPress={() => {
          navigation.navigate(SCREENS.SIGNIN);
        }}
        size="full"
      />
    // ...
  );
};
```

Figure 27: Sample code for the “Sign In” button on the Start screen

The above code illustrates baseline functionality for the red “Sign In” button previously located on the Start screen. Annotating the component as `React.FC<StartScreenProps>` allows us to properly force a type on the screen’s props, which in this case only includes the `navigation` callback. This pattern is, in general, how navigation between screens is performed in React Native.

Once on the Sign-in screen, there are two ways to return to the previous screen:

- using hardware or virtual buttons or screen gestures, or
- pressing the “Cancel” button. In this case, the navigation must be handled manually. However, it is not desirable to use the above code pattern exactly, as the Stack Navigator will put new screens on top of old ones by default and allow for exploitation of the indefinite size of the navigation stack. Instead, the `popToTop` function is used in a custom `CancelButton` component as follows.

```
const CancelButton = ({ navigation }: CancelButtonProps) => (
  <View style={{ paddingBottom: 10 }}>
    <Button
      title={localString.cancel}
      onPress={() => navigation.popToTop()}
      size="small"
    />
  </View>
);
```

Figure 28: Code for the Sign-in/Register screens' "Cancel" button

Manually calling `popToTop` in this way allows us to ensure consistency when navigating within the Stack Navigator by more strictly controlling the number of possible navigation paths for users; this is a recurring design paradigm for the front end.

Credential Validation

When actually logging in (done by pressing the "Sign In" button from the Sign-in screen), the credentials typed in need to be checked against the database and verified, and in the case of success the user's information is returned and the rest of the application is loaded with the retrieved user. This is simply done using the Login User API endpoint, detailed in the API Specification section of this document. Below is the code for the corresponding Axios request.

```
export interface Session {
  email: string;
  password: string;
}

export const doSignin = (credentials: Session) =>
  requests.post("/session", credentials);
```

Figure 29: Code responsible for the Sign-in screen's API call

Register

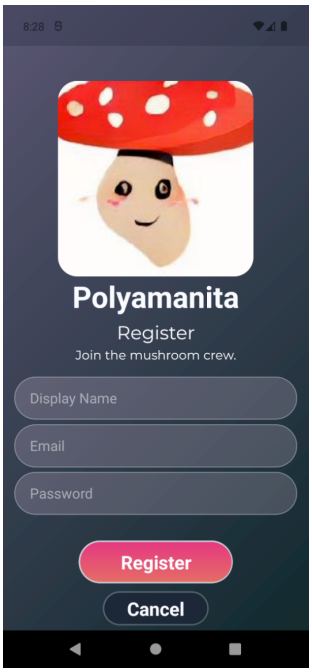


Figure 30: Screenshot of Register screen for reference

The code responsible for the Register screen can be found in `src/screens/initial-screens/register/register-screen.tsx`.

Navigation

Navigation between the Start and Register screens is essentially identical to that between Start and Sign-in. Code samples are given below for the sake of completeness.

```
const StartScreen: React.FC<StartScreenProps> = ({ navigation }) => {

  // ...

  return (
    // ...
      <Button
        title={localString.register}
        varient="primary"
        onPress={() => {
          navigation.navigate(SCREENS.REGISTER);
        }}
        size="full"
      />
    // ...
  );
};
```

Figure 31: Code for the “Register” button on the Start screen

```
const RegisterScreen: React.FC<RegisterScreenProps> = ({ navigation }) => {

  // ...

  return (
    // ...
      <CancelButton navigation={navigation} />
    // ...
  );
};
```

Figure 32: Code for the Register screen and its “Cancel” button

Input Validation (Registration)

As shown in the mockup image for the Register page, the text fields of the Register screen can have red or green borders on top of the default border color; this predictably indicates that a given field is either valid or invalid. The checks on the validity of these fields are performed upon the cursor leaving a specific box:

- the length requirement for the username,
- whether the email address has a valid format, and
- whether the password adheres to our password requirements.

The exact requirements for these can be found under the “Register User” API endpoint header in the API Specification section. The actual implementation of these

Uniqueness of the user’s email address and desired username are checks that must be performed online, via the Register User API call. This API call is performed very similarly to the one for Sign-in (see above), and returns an error if any of the fields do not adhere to the requirements there. If the front end ensures that the format requirements, enumerated above, are met, then an error implies either the username or email are not unique, which is communicated to the user.

Email Confirmation

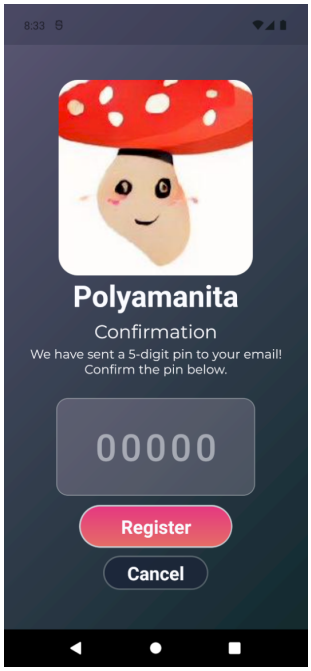


Figure 33: Screenshot of Confirmation screen for reference

This screen simply exists to provide a nicer look and feel to the second stage of user registration, and involves inputting the required confirmation code sent via email after successfully hitting the “Sign Up” button from the Register screen; the code generated from a Generate Auth Code API call is used and the actual call to the Register User endpoint is made upon successful input and pressing the “Confirm” button.

The code responsible for the Confirm screen can be found in `src/screens/initial-screens/confirm/confirm-screen.tsx`.

Navigation

The implementation of the navigation between this screen and the Register screen is essentially identical to that of the previously-discussed screens; refer to those for details.

Snap/Capture

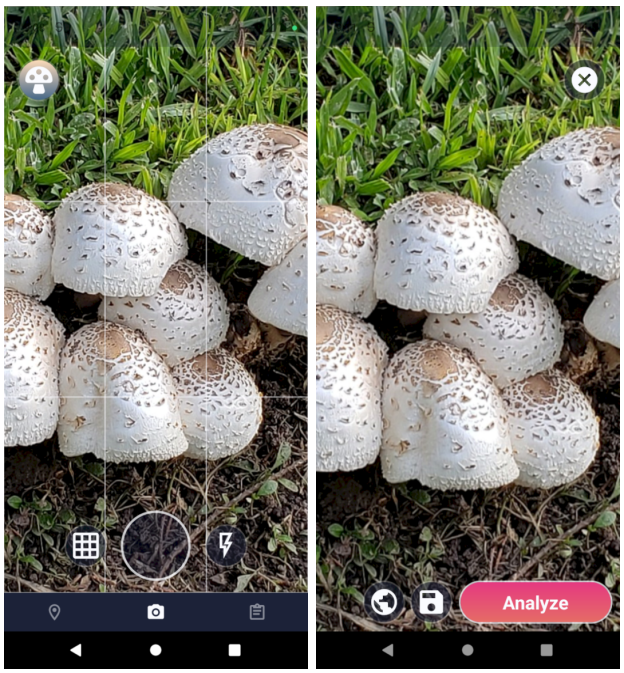


Figure 34: Screenshots of Snap and Capture screens for reference, respectively

These two screens are responsible for the photo-taking portion of the app. The Snap screen, technically speaking, is fairly simple. A VisionCamera camera instance is rendered, and upon snapping a photo the user is navigated to the Capture screen. The flash and grid toggles work as intended; the former once again courtesy of VisionCamera builtins, and the latter a simple toggle for visibility of the custom grid overlay component.

The Capture screen has more to discuss. The main point is the “Analyze” button, which is responsible for running the classification model (which is bundled with the mobile app); this is discussed below.

After successfully capturing a mushroom, the user is navigated to the Post-Capture screen, which is a

The code responsible for the Snap screen can be found in `src/screens/snap/snap-screen.tsx`. The code for the Capture and Post-Capture screens is located in the same folder, in `src/screens/snap/capture-screen.tsx` and `src/screens/post-capture/post-capture.tsx`, respectively.

Shroomalyzer

As defined earlier, the Shroomalyzer is the name given to our custom mushroom classification model. This model is bundled with the app as a TFLite (TensorFlow Lite) model file, and classification is performed via a native Android plugin. This plugin is executed from the helper file `src/screens/snap/utils/shroomalyze.ts`, and its actual Java code is found at `android/app/src/main/java/com/rntypescriptboilerplate/ShroomalyzerPlugin.java`.

Within this plugin, the necessary image manipulation and resizing is performed, then the image object is passed to TFLite Android library methods in order to feed it as an input to the model. A softmax regression and category labels are then applied to the resultant output vector and it is passed back to the main app via a callback method as a key-value set (a JSON object on the React Native end). In case of some error (which should never happen in normal usage of the app), a similar error callback is used. One of these callback methods is shown here as an example.

```
private void passResults(Callback cb, Map<String, Float> results) {
  WritableNativeMap bridge = new WritableNativeMap();
  for (Map.Entry<String, Float> entry : results.entrySet()) {
    bridge.putDouble(entry.getKey(), entry.getValue());
  }
  cb.invoke(bridge);
}
```

Figure 35: A Shroomalyzer callback method

On the React Native side of things (back in the main codebase, in `shroomalyze.ts`), basic promise handling is performed via the result and error callbacks; it is also here that a simple threshold is applied to the results to determine whether the model has identified a mushroom with enough confidence to reliably report.

Navigation

Upon entering the Camera screen, the user will also be presented with an ever-present set of tabs along the bottom of the screen, which they can use to navigate to the rest of the major screens of the application. Usage of a Tab Navigator is actually very similar to that of a Stack Navigator, though less care needs to be taken as the lack of an actual navigation stack means that infinitely stacking screens is no longer possible; in other words, we can allow the user full freedom to move between the set of 4 screens.

The necessary code for the rendering and behavior of the tab navigator is too long to reasonably show here; instead, refer to its source file location at `src/navigation/tab-navigation.tsx`.

Journal

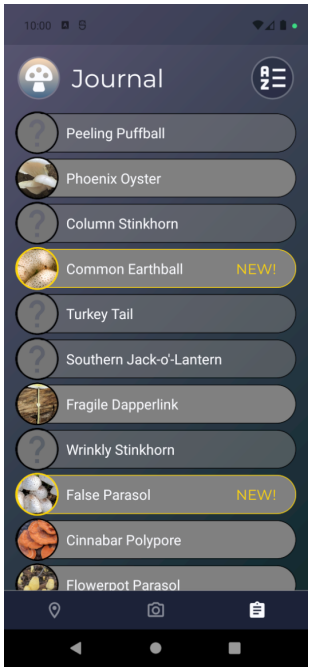


Figure 36: Screenshot of main Journal screen for reference

The code responsible for the Journal screen can be found in `src/screens/journal/journal-screen.tsx`.

Capture List

Most prominently rendered on the Journal screen is the list of mushrooms, with previously-captured mushrooms displayed differently and newly-captured ones with “unread” badges. At its core, this is actually just a modified list of *all* supported mushroom species’ IDs (which is found in `src/shared/constants/mushroom-names.ts` and exposed to the rest of the app as a single JSON object `MUSHROOM_IDS`). Whether a specific mushroom has been captured before by the user, as well as its capture data in the case that it has, is stored differently.

The type `CaptureMap`, from `src/screens/journal/utils.ts`, defines an object type with mushroom IDs as keys and values containing their corresponding capture objects and a flag for whether the mushroom has new “unread” capture instances.

```
export type CaptureMap = {
  [shroomID: string]: {isUnread: boolean; capture: CaptureInstance};
};
```

Figure 37: The `CaptureMap` type

A JSON object is used as a map here in order to make it easy to check whether a given mushroom ID has associated captures; this check is performed in `journal-screen.tsx` when rendering each list item to determine whether to render a completed or undiscovered item.

Additionally, to save loading time, the `CaptureMap` object is cached via storage in the application’s global Redux state. The user’s captures are only fetched on initial visit to the screen, and on subsequent visits only after a capture has been made. Consult the above utility code file and the files in the `src/redux` directory to refer to how this was implemented.

Navigation

The only navigation concerns for this screen are the bottom Tab Navigator and a Journal Stack Navigator, which have been discussed already; see above.

Mushroom Details (and Undiscovered)

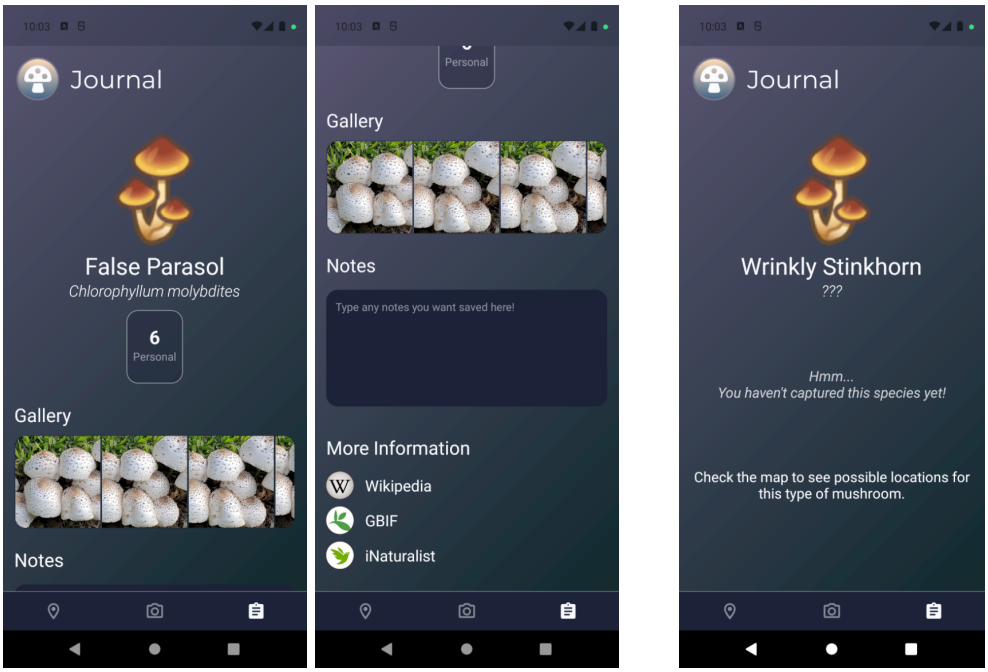


Figure 38: Screenshots of Mushroom and Undiscovered screens for reference

These screens are reached from individual list items on the Journal screen. If a mushroom has been previously captured by the user, the completed Mushroom screen is rendered, with a gallery list of instances, the notes on that mushroom, and links to external sources. The screen receives the appropriate props for these items and renders them, using subcomponents as necessary.

The gallery items are scrollable and navigate to Image screen instances when pressed, through use of a mapped array of `TouchableHighlight` elements rendered within a custom `Gallery` subcomponent. The notes box is an editable text entry box, and they are saved to the database when the box is unfocused (i.e. the user closes the keyboard or navigates away from the screen).

Alternatively, if the user has not captured the mushroom, the mostly-generic Undiscovered screen is rendered, which contains only the mushroom's common name and some messages.

The code for the Mushroom and Undiscovered screens can be found in `src/screens/mushroom/mushroom-screen.tsx` and `src/screens/journal/notfound-screen.tsx`, respectively.

Navigation

The only navigation concerns for this screen are the bottom Tab Navigator and the Journal Stack Navigator, which have been discussed already; see above.

Image

Figure 39: Screenshot of Image screen for reference

The Image screen is navigated to from both the Journal and Map screen, and contains the image associated with a particular capture instance, as well as information about the capture in text. This screen is fairly simple; most of its code is just UI styling, and it only requires the minimum necessary capture information as component props.

As previously mentioned, a “copy” of the Image screen is rendered as the Post-Capture feedback screen.

The code for the Image screen can be found in `src/screens/image/image-screen.tsx`.

Navigation

The only navigation concerns for this screen are the bottom Tab Navigator and the Stack Navigators that include this screen, which have been discussed already; see above.

Map

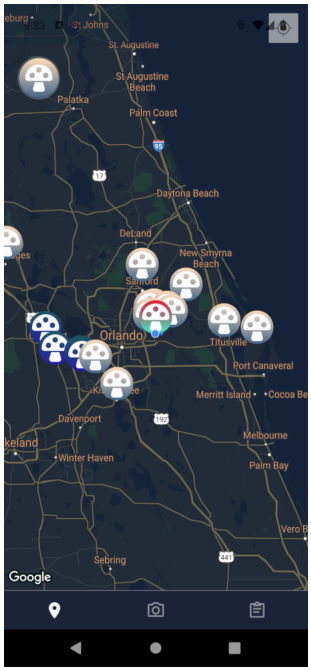


Figure 40: Screenshot of Map screen for reference

Like the Camera screen, use of a third-party-library makes creating this screen far easier than from scratch. The react-native-maps API is fairly straightforward and provides simple ways to control the region, zoom, and pins we render. When tapped, the pins on the map (organized prior to rendering as an array of objects that include user IDs, capture IDs, and capture instances) open up corresponding instances of the Image screen to show details for the capture instance.

The code for the Map screen can be found in `src/screens/map/map-screen.tsx`.

Navigation

The only navigation concerns for this screen are the bottom Tab Navigator and a Map Stack Navigator, which have been discussed already; see above.

Community

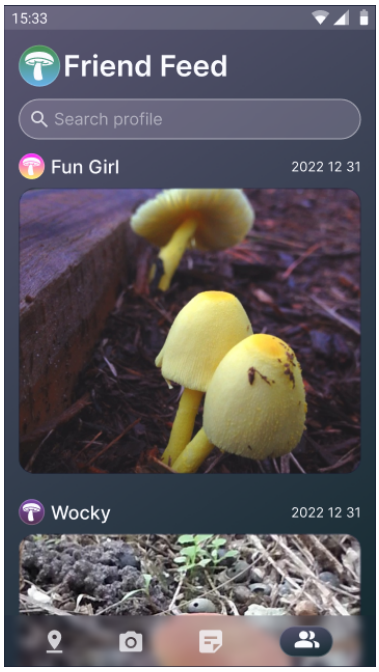


Figure 41: Mockup of Friend Feed screen for reference

This screen's features were moved to a stretch goal early on in development. That stretch goal was not met, and thus this screen and its features were not implemented on the app or API server. Implementation details have thus been excluded for the sake of brevity.

Database Specification

Polyamanita manages two types of databases in the backend, with various functions for each. For user-based data and statistics storage, Polyamanita uses AWS' DynamoDB and S3 as a relational database, for several reasons:

- Able to utilize other AWS tools in conjunction such as S3, Elasticbeans and others.
- NoSQL database makes it easier to setup and eliminates the need to manage tables, only entries.

Captured mushroom images are stored on S3 as a file store, for several reasons:

- Images have large file sizes, which can be expensive if stored on a relational database
- AWS S3 has many built in features that make it easy to use, such as S3 replication, storage management and monitoring
- Images can be stored in mass for each user's captured mushrooms in a more intuitive way through file hierarchy

Detailed below are the specifications for both the relational database tables for user data, and the file store structures for captured mushroom image store and mushroom notes.

User Account Table

The User account Table is used to display all information relevant to a user's personal information and social statuses. Detailed below are the use cases of each field in the user table.

- **UserId** is the primary way in which users are referenced in the back-end app. Mushroom statistics are retrieved using the User ID, and are unique to each user. This will stay static upon creation and cannot be changed after a user is created.
- **Username** is used primarily to find other users on the application. They are unique to and set by each user, and cannot be changed after a user is created. Usernames are also used when logging a user into the application
- **Color1** and **Color2** are used to store custom defined colors to be shown in various aspects of a user's account, primarily in their account icon. They are set by the user. They are stored as a 8 character Hex value that correlates to their color preceded by a hash (**#**).
- **Email** is used primarily to authenticate user operations in the cases where a username and password are insufficient to confirm user identity (e.g. in the case of users who forgot their password) by storing a confirmed email that the user confirmed access to during account creation.
- **Password** is used for the purpose of confirming user identity alongside the user's username.
- **Follows** contain a list of User IDs that represent which users the current user has followed. This is used when retrieving a user's follow post board, or seeing user leaderboard statistics.
- **Followers** contain a list of User IDs that represent which users follow the current user. This is used when retrieving a list of users that follow the user.

```
CREATE TABLE Users (
 UserID      STRING(1024) NOT NULL,
 Username    STRING(1024) NOT NULL,
 Color1      STRING(8),
 Color2      STRING(8),
 TotalCaptures INT64,
 Email       STRING(1024) NOT NULL,
 Password    STRING(1024) NOT NULL,
 Follows     ARRAY<STRING>
 ) PRIMARY KEY (UserID);
```

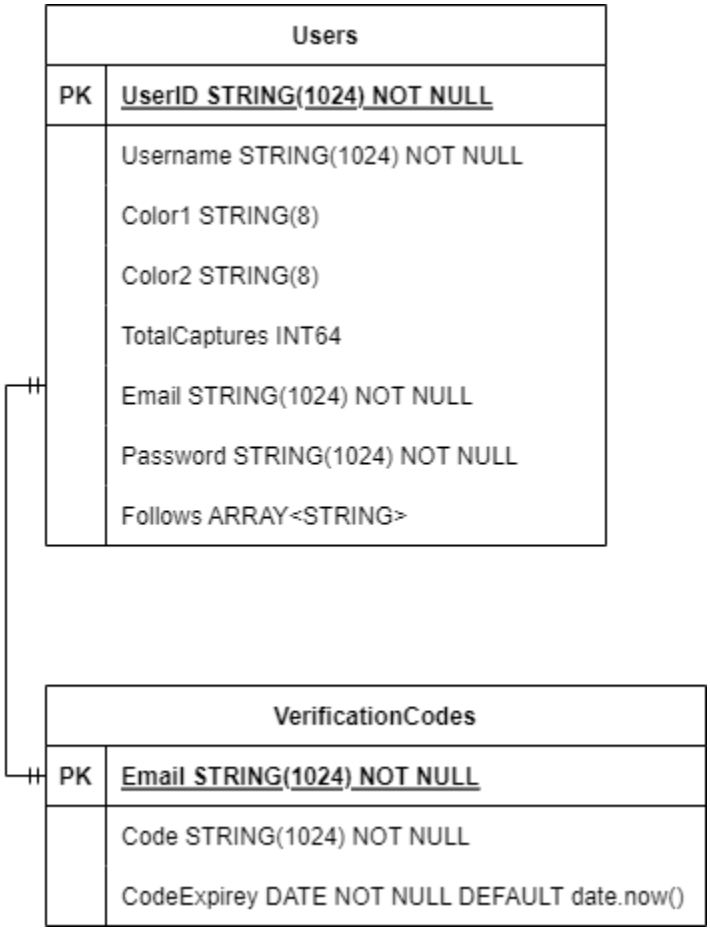
Verification Code Table

The verification code table is used to store information about the verification codes used in verifying a user for registration, resetting their password and deleting their account. This method allows us to authenticate a user before they take any important or “potentially dangerous” actions, for example deleting an account is non reversible and if someone other than the user resets their password they would gain control of their account and lock the intended user out. It’s important to note that these entries will be temporary and will be deleted one hour after creation, this along with unique emails allows us to avoid the spam creation of verification codes which could clog up the database. Below are the fields and descriptions that make up the table.

- **Email** is the primary key used to identify which each verification code is associated with what user. The email is unique and there cannot be multiple entries with the same email, so essentially each user will only be allowed one verification code at once. This means that if another one is generated the previous one will be replaced.
- **Code** is the main reason behind each entry, each code is a randomly generated four digit pin from zero to 9999. The user will submit their code and once that code is verified, the user will be allowed to execute whatever action they were trying to take.
- **CodeExpiry** is used to track when a verification code has been created. Each code will only last one hour upon creation before being deleted in order to prevent the spam creation of codes.

```
CREATE TABLE VerificationCodes (
```

```
Email       STRING(1024) NOT NULL,
Code        STRING(1024) NOT NULL,
CodeExpiry  DATE NOT NULL DEFAULT date.now()
) PRIMARY KEY (Email);
```



Captured Mushrooms Table

The table captured mushrooms will contain all the information related to the mushrooms captured by a user. The table works so that there is only one entry for each mushroom species per user. So each entry will be identified by a primary key consisting of the user's ID and the mushroom species' ID. Each entry will also contain a struct array called instances that holds information about each mushroom of that type the user has found. Each instance will hold general statistics about the mushroom, primarily the location derived from the longitude and latitude and the S3 key used to retrieve the image of the scanned mushroom. The captured mushrooms table will also be interleaved under each user, so when a user account is accessed this also tells the

database to access any mushroom information related to that user allowing for faster lookup times.

- **CapturedMushroomID** stores the ID for each different type of mushroom. This ID is the primary way in which mushroom types are referenced in the back-end app. The user may not see each exact ID, instead each mushroom may correlate to a different more recognizable name. Each type of ID is a unique primary key and cannot be used more than once, however combining two primary keys, UserID and CapturedMushroomID, allows us to identify which user has discovered which mushroom type without unnecessarily clogging up the database with redundant information.
- **UserID** is the primary way in which users are referenced in the back-end app. In this table the UserID is a foreign key and primary key that is from the user account table. It allows us to associate a user account with the different types of mushrooms they've found and works in conjunction with the CapturedMushroomID.
- **TimesFound** is a field used to store the number of times each type of mushroom is found. This is a simple integer field used to store mushroom statistic information.
- **Notes** is a field used to store the custom notes a user wishes to create for each type of mushroom, alongside the information already provided by the machine learning algorithm.
- **Instances** is an array of type struct used to store each mushroom of a particular type a user has found. The struct type allows us to store information specific to that particular mushroom.
- **Longitude and Latitude** are used to store the exact location a mushroom has been found. When a mushroom is scanned its longitude and latitude are stored in these fields to be used in the map graphic.
- **Location** is used to store the approximate location a mushroom has been found. When a mushroom is found it gets an approximate area centered around the longitude and latitude
- **DateFound** is a field used to store when a scanned mushroom has been found. This is a simple date field used to store mushroom statistic information and can be shown to the user or used within graphics included in the app.
- **S3Key** is how each image taken to scan the mushroom is stored. Each key is used to represent a file path within S3 storage and is needed to retrieve each object which in this case is our image file.

```
CREATE TABLE Captures (
 CaptureID          STRING(1024) NOT NULL,
 UserID             STRING(1024) NOT NULL,
 Species            STRING(1024) NOT NULL,
 TimesFound         INT64 NOT NULL,
 Notes              STRING(1024) NOT NULL,
 Instances          ARRAY<STRUCT<
                     Longitude      FLOAT64,
                     Latitude       FLOAT64,
                     Location       STRING(1024),
                     DateFound      DATE NOT NULL DEFAULT date.now(),
                     S3Key          STRING(1024)
                    >>
 ) FOREIGN KEY (UserID),
   PRIMARY KEY (UserID, CapturedMushroomID),
INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```

Captures	
PK	CaptureID STRING(1024) NOT NULL
PK, FK	UserID STRING(1024) NOT NULL
	Species STRING(1024) NOT NULL
	TimesFound INT64 NOT NULL
	Notes STRING(1024) NOT NULL
	Instances ARRAY<STRUCT>
	Longitude FLOAT64
	Latitude FLOAT64
	Location STRING(1024)
	DateFound DATE NOT NULL DEFAULT date.now()
	S3Key STRING(1024)

Mushrooms Image Store

Mushroom species that a user captured will be stored in a file-storage structure, with the help of AWS S3.

Buckets

Polyamanita will use AWS' S3 for storing user-taken images of mushrooms, so only one bucket will need to be created to store all data, per each environment. In total, 3 buckets will exist:

- `Polyamanita-image-store-dev`, for local app development and testing.
- `Polyamanita-image-store-staging`, for generated data store and stress testing.
- `Polyamanita-image-store`, for production level data.

Keys

Object keys will be named to and structured as so:

- At level 0 (base of file name), image objects are labeled according user ID
- At level 1, image objects are labeled according to mushroom dex ID
- At level 2, image objects are labeled according to mushroom capture ID

For example, a user with id `mushroom_hunter` capturing their 7th mushroom of dex ID 1234 would have their image get stored under the object name `/mushroom_hunter/1234/7`

This hierarchy allows for filtering according to each user's collection of mushrooms found, with more granular filtering for pictures a user took of a specific mushroom.

Technologies

AWS DynamoDB

DynamoDB is a NoSQL database offered by Amazon which offers streamlined management, unlimited scale, and almost constant availability. Creating databases with DynamoDB is straightforward, allowing for setup using AWS's Console. Polyamanita bases its database on the foundation of DynamoDB for several reasons:

- DynamoDB provides a Golang SDK, offering intuitive hookup to the API layer of Polyamanita for faster development and increased understandability. The SDK

implicitly supports callback-oriented use with the database, allowing for API interactions to be built effectively with scalability in mind.
- The most prominent feature of DynamoDB is its application of version control on data. This creates strong confidence in data consistency and integrity, preventing any ambiguities in issues our database data could face when performing user information fetches or syncing from multiple devices.
- DynamoDB is part of the AWS, unifying our technologies' sources for handling credentials across AWS service accounts, providing usage metrics and suggested optimizations to services, and billing from our service activity.
- DynamoDB's out of the box table interleaving allows for mushroom data to be closely located to each user's data, allowing for faster read times and ultimately a response from the API layer

DynamoDB will be accessible only to the API layer, meaning a data transfer from the DynamoDB DB to the polyamanita API, then to the front end will be necessary to perform any operations. This sacrifice is necessary for the purposes of keeping critical API keys stored out from the front-end, to mitigate security risks.

Schema

DynamoDB has a schemaless database setup since it's NoSQL, allowing for the database to be easily modifiable and updateable in the case of new features.

AWS S3

Amazon's Simple Storage Service (S3) is a data storage service provided by Amazon that allows for storing unstructured data in a manner that is easily administrable and cost-effective. Data is organized into buckets, with metadata keys that represent paths to a virtual file-path system.

S3 will be accessible only to the API layer, meaning a data transfer from the bucket to the polyamanita API, then to the front end will be necessary to perform any operations. This sacrifice is necessary for the purposes of keeping critical API keys stored out from the front-end, to prevent any security risk.

Buckets

Buckets are the highest level storage component in S3, representing completely separate file stores to store objects under. They are used to organize and control access to objects, and cannot be nested.

Objects

Objects are the representation of any piece of data uploaded to S3. All objects are stored in buckets and are uniquely identified with the help of its object key.

Keys

Objects have key paths that correlate to a virtual file-system. Depending on object name, they can be organized according to different virtual folders and filtered as such. The “/” character is used to signify breaks in the object name, creating a hierarchy as such.

Changes made

Google to AWS

We decided to move from using google cloud to using AWS’ cloud services. Our primary reason was one of our API members at the time had recently started working with AWS and thought it was better to work with and a good opportunity to try something new. We traded google cloud spanner and storage for AWS’ DynamoDB and S3.

API Specification

Representational State Transfer (REST)

Representational State Transfer (REST) is a series of structural constraints on an API that influence the way it ends up being interpreted and used. Polyamanita’s server side implements RESTful practices, creating an intuitive and consistent environment in which calls can be made and recorded in the back end.

In order for Polyamanita to conform to RESTful standards, it conforms to these RESTful practices:

- All of Polyamanita’s server-client communications are stateless, creating endpoints that are completely independent from one another
- Clear distinctions between the structure of data sent to the server and the responses sent back to the client
- A layered routing system that clearly distinguishes between different types of resources and actions performed on those resources.

SwaggerHub

Swagger hub is a useful tool used to plan out and create API documentation. It allows us to easily design an API before actually implementing code. With Swaggerhub we can organize and plan out which endpoints we think we need, what their request bodies, responses and response bodies will look like without actually having to program the whole endpoint. Utilizing tools such as Swaggerhub helps speed up the design process and avoid long and costly mistakes, such as spending a large amount of time developing an endpoint only to realize we don't actually need it or it doesn't work.

Volume of API Calls

Our server will need to process two main groups of API calls, the first one being related to account CRUD operations. The volume of CRUD operations will likely be less than the total number of users we have registered, since the most common CRUD operation will likely be logging in and we want to keep our users logged into their accounts on

their devices unless they log out in order to limit the number of api calls in lower internet connectivity areas required. Users will also only realistically update their profiles a few times throughout their usage of the app as those details will have minimal impact on the rest of the app. This leaves creating and deleting which on average will only be done once per user.

The second group of API calls will be related to keeping track of a user's journal, so which mushrooms they've found and any personal notes or images taken. For this group the highest volume of CRUD operations will be add and update, since we will need to add every new mushroom discovered. It's also very likely the average user will make more than one edit to any previous discoveries in order to add any new pictures of mushrooms of the same type or any notes they may not have thought of right away when initially scanning the mushroom.

The delete operation will likely only be used in cases where a user experiences an error or a user wanting to rediscover a mushroom since deleting will re-mark a mushroom as undiscovered.

The retrieve operation will only be used when a user logs into their account and needs to retrieve their journal and their personalized notes and images. Again since we want to limit the number of API calls required in the event the user is in an area of low connectivity we intend to keep the user signed into their devices unless they choose to log themselves out. While keeping the volume of API calls and database storage in mind we've chosen to utilize AWS' Elastic Beanstalk to host our server

Decreased Reliance on Internet Connectivity

Due to the likelihood that the app will primarily be used in areas of weak to no internet access, we aim to make as few calls to the server as possible and keep the need for internet access as low as possible. In order to accomplish this different parts of the app may consume more storage in order to omit the need to talk to the server. Primarily this will be done in a way so that either the information is either already in the app, will be stored within the app locally instead of server side or the app will bundle API calls together in batches and talk to the server when the app has an internet connection. However certain calls would not be bundled and stored until the user has an internet connection since they might be changing the app in a way that the user would not be able to use the app until the endpoint has been processed.

Authentication Token Endpoints

Detailed below are all endpoints related to access token and refresh token authentication and generation.

Login User (POST /session)

After successfully registering the user will be automatically logged in and brought to the snapview. Alternatively a returning user can login through the login view by submitting the correct credentials.

This is a POST endpoint that matches the correct email and password with an entry within the database and retrieves a user's associated account information.

- Retrieves associated account information located in the user account table, captured mushrooms table and friends table.

Login would be another endpoint that requires immediate internet connection since it would be too high cost and complex to bundle and queue it up with others and it requires an immediate response from our server. We can omit the need for constant internet connectivity keeping the user logged in until they logout or are logged out

- If the user's password is changed they will need to log back in on all previous devices for security reasons.
- If a user's email is reset they will also need to log back in on all previous devices for security reasons.

Request Body

```
{
```

```
    "email": string,
    "password": string
}
```

Response

- 200: successfully login
- 400: Incorrect login credentials
- 500: Any of the database services returned errors.

Response Body

```
{
    "id": string,
    "accessToken": string
}
```

Logout (DELETE /session)

Once logged in the user will be able to log out by navigating to their profile view and simply clicking the logout button.

This is a DELETE endpoint that deletes all of the user's data from the session.

- This will let the app know to clear any associated account information stored after a login.
- This endpoint can not only be manually called by the user but also will be called in the event of a password reset or email reset which both require the user to log back in for security reasons.

No internet connectivity is required for this endpoint as it is simply forgetting data.

Request Body

```
{

}
```

Response

- 200: Successful logout

- 500: Any of the database services returned errors.

Response Body

```
{

}
```

Generate Registration Auth Code (POST /auth)

After the user submits their account information they will be taken to another view where they can submit their personal verification code. An email will have been sent containing the code they need to enter. Once they enter their personal verification code correctly the user will be automatically logged in.

This is a POST endpoint that creates an entry within the verification code table after the user completes the account information portion of registration.

- The verification code entry will consist mainly of a user's email and the generated verification code. These are the two fields that will be checked to verify a user within the Register User endpoint.
- The verification code entry will last one hour before being deleted in order to prevent spam creation, afterwards the user will need to restart the account creation process.

This would be an endpoint that requires immediate internet connection since it would be too high cost and complex to bundle and queue it up with others and it requires an immediate response from our server.

Request Body

```
{
    "email": string,
    "password": string,
    "username": string
}
```

Response

- 200: Code was successfully created
- 400: Body was incorrectly formatted or email is invalid.

- 500: Any of the database services returned errors.

Response Body

```
{
    "codeExpiry": Time
}
```

Generate General Auth Code (POST /authGen)

This is a generalized version of the auth code endpoint that can be used anytime we require authorization from the user. An email will have been sent containing the code they need to enter. Once they enter their personal verification code correctly the user will be automatically logged in.

This is a POST endpoint that creates an entry within the verification code table after the user completes the account information portion of registration.

- The verification code entry will consist mainly of a user's email and the generated verification code. These are the two fields that will be checked to verify a user within the Register User endpoint.
- The verification code entry will last one hour before being deleted in order to prevent spam creation, afterwards the user will need to restart the account creation process.
- This endpoint will also be used to authenticate a user during password reset and account deletion

This would be an endpoint that requires immediate internet connection since it would be too high cost and complex to bundle and queue it up with others and it requires an immediate response from our server.

Request Body

```
{
    "email": string
}
```

Response

- 200: Code was successfully created
- 400: Body was incorrectly formatted or email is invalid.

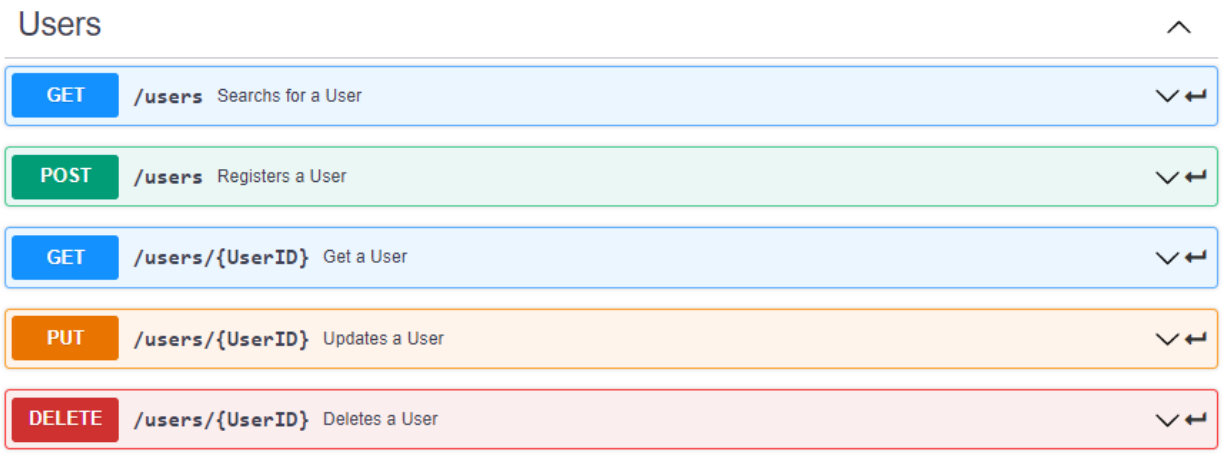
- 500: Any of the database services returned errors.

Response Body

```
{
    "codeExpiry": Time
}
```

User Account Endpoints

Detailed below are all endpoints related to user account management that will be active upon running Polyamanita's back end server, including use case and restrictions for use. This spans from user creation to user data updating and account deletion



Search User (PUT /users)

As a stretch goal we want our users to interact with others, so a search user endpoint allows them to find and view other peoples accounts. There they can add them as friends or view their feed and so on.

This is a GET endpoint that retrieves the information of multiple users filtered by whether their usernames contain the inputted string

-

Searching for users would be another endpoint that requires immediate internet connection since it would be too high cost and complex to bundle and queue it up with others and it requires an immediate response from our server.

Request Body

```
{
    "username": string
}
```

Response

- 200: Results found
- 500: Any of the database services returned errors.

Response Body

```
{
  "users": [
    {
      "TotalCaptures": int,
      "color1": string,
      "color2": string,
      "email": string,
      "follows": [string],
      "mainSort": string,
      "userID": string,
      "username": string
    }
  ]
}
```

Register User (POST /users)

Once the user arrives at the account registration view they will be able to fill in the required information to create their own account. After submitting their user information a verification code will have been sent to their email which they can submit in the next view to complete the registration process. Only a username, email, and password are required to register an account and begin using the app.

This is a POST endpoint that creates an entry within the user account table using the information submitted on the registration screen and the verification code generated by the Generate Registration Auth Code endpoint.

- The user must fully complete the registration process for their account to be saved within the database, if the app is fully closed midway their registration information will be lost and they will have to re-generate another verification code.
- In order to prevent login complications users will not be able fto register an email already in use
- To successfully be accepted the new password must match the security criteria; must be at least 8 characters, including a capital letter, a number, and a special character. For security there will be checks on both the frontend and backend to verify the criteria.
- The user can only submit the code that is associated with their account which is verified via the same email and same code entered. This way a user cannot accidentally or purposefully steal someone else's verification code.

This would be an endpoint that requires immediate internet connection since it would be too high cost and complex to bundle and queue it up with others and it requires an immediate response from our server. While the account registration portion of the app requires internet connectivity it's unlikely that a user would have the internet connection strength to download the app but not enough to create an account. We can also include a tip within the download page for any users to create their accounts before using the app in areas where they have any internet access.

Request Body

```
{
    "code": string,
    "username": string,
    "password": string,
    "email": string,
}
```

Restrictions

- `password` must be at least 8 characters, including a capital letter, a number, and a special character (@, #, $, %, !)
- `username` must not exist in the user database.
- `email` must not exist in the user database.
- `code` must be correct to the code entered for the associated email in the verification code table.

Response

- 200: User successfully created.
- 403: User code was incorrect.
- 400: Restrictions on body fields were not met or body was incorrectly formatted.
- 500: Any of the database services returned errors.

Reset Password (PUT /users/<user id>/resetPass)

In the event a user has misplaced their password or needs to reset it for security reasons, they can begin the process by following the forgot my password button on the login view or by navigating to their profile and following the reset password button. Once at the reset password view a user will be able to generate another verification code similar to the one sent during registration. After verifying the code found within their email they can create a new password to use for their account so long as it meets the security criteria

This is a PUT endpoint that updates a user's old password with a new one they have created, as long as it matches the security criteria. The Generate General Auth Code endpoint will be called to generate the code required to complete this endpoint.

- The verification code is required in order to make sure nobody other than the user is able to change their password. We strongly advise against sharing the password to your account in order to prevent your password from accidentally being leaked
- The user must fully complete the password reset process for their account to be saved within the database, if the app is fully closed midway their new password will be lost and they will have to re-generate a new verification code.
- To successfully be accepted the new password must match the security criteria; must be at least 8 characters, including a capital letter, a number, and a special character. For security there will be checks on both the frontend and backend to verify the criteria.

Resetting the user's password would be another endpoint that requires immediate internet connection since it would be too high cost and complex to bundle and queue it up with others and it requires an immediate response from our server. Since a user would require internet access to check their email for the verification code we can assume that a user will likely already have internet access when completing this step; however the app will alert the user that an internet connection is required if they attempt to reset their password without one.

Request Body

```
{
    "code": string,
    "password": string,
    "email": string
}
```

Restrictions

- `password` must be at least 8 characters, including a capital letter, a number, and a special character (@, #, $, %, !)
- `code` must be correct to the code entered for the associated email in the verification code table.

Response

- 200: User password was successfully reset.
- 403: User code was incorrect.
- 400: Restrictions on body fields were not met or body was incorrectly formatted.
- 500: Any of the database services returned errors.

Response Body

```
{
    "message": string
}
```

Update User Information (PUT /users/<user id>)

Once logged in the user can navigate to their profile view and choose to update their profile information.

- Personal information such as username, email and password can be changed provided the requirements for each is met, for example the new username or email cannot already be taken by another user.
- Any boxes left empty will signify to the user account table that the particular field can remain the same.

This is a PUT endpoint that updates the fields associated with the current user within the user account database.

- The endpoint will overwrite fields in the user account table with the new fields submitted by the user, any fields left blank will remain the same and not be overwritten.
- Email or username cannot be updated with ones that are already in use
- To successfully change a password associated with a user account they must complete another email verification generated by the Generate General Auth code endpoint and a separate reset password endpoint

Updating the user's profile would be another endpoint that requires immediate internet connection since it would be too high cost and complex to bundle and queue it up with others and it requires an immediate response from our server. While one solution could be to queue up the change until internet access is secured, it's significantly less complex to create a pop up asking the user to secure an internet connection before making any changes.

Request Body

```
{
    "username": string,
    "email": string
}
```

Response

- 200: successfully updated account
- 400: Incorrect password
- 500: Any of the database services returned errors.

Response Body

```
{
    "message": string
}
```

Delete User (DELETE /users/<user id>)

A User may delete their profile by navigating to the very bottom of their user profile view located under a submenu and following the onscreen directions.

- Since an account cannot be recovered once deleted from the user account table a multiple step process is used to ensure a user is confident they want to delete their account and it is not accidental or spur of the moment.
- The user will be required to complete another verification email similar to registering their account and resetting their password
- They will also be prompted to retype their password for verification along with the code they received

This is a DELETE endpoint that searches the entire database for all entries associated with a particular user account and deletes them. The Generate General Auth Code endpoint will be called to generate the code required to complete this endpoint.

- Requires a user's correct password and the correct newly generated code associated with their account
- The user must fully complete the delete user process for their account to be deleted from the database, if the app is fully closed midway they will have to redo the process and re-generate a new verification code.
- Frees entries within the user account table, captured mushrooms table and friends table. This would free the email to be used in the creation of another account and would remove all friends associated with a particular user and vice versa

Deleting the user's profile would be another endpoint that requires immediate internet connection since it would be too high cost and complex to bundle and queue it up with others and it requires an immediate response from our server. If a user deleted their account they would not be able to do anything with the app until the account is deleted and they logged into a new account.

Request Body

```
{
    "code": string,
    "email": string,
    "password": string,
    "refreshToken": string
}
```

Restrictions

- `code` must be correct to the code entered for the associated email in the

verification code table.

Response

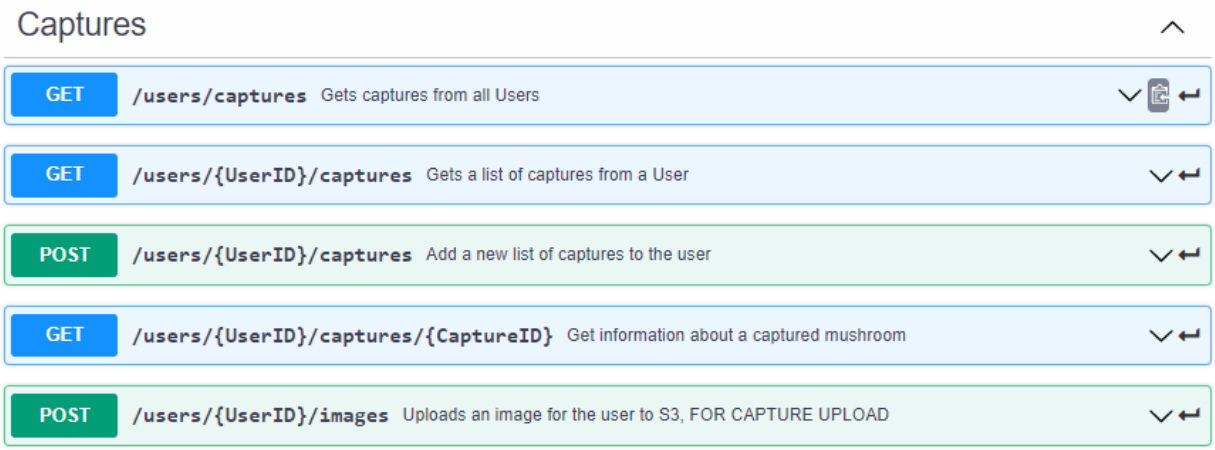
- 200: Account successfully deleted.
- 403: User code or password was incorrect.
- 500: Any of the database services returned errors.

Response Body

```
{
    "message": string
}
```

Mushroom Statistics Endpoints

Detailed below are all endpoints related to mushroom data for a user that will be active upon running Polyamanita's back end server, including use case and restrictions for use. This spans from retrieving data about a user's mushroom statistics to adding new captures.



Get All Captures (GET /users/captures)

The map feature will require information on all the captures from users in the area in order to properly display them. Longitude, latitude and the capture id are particularly important here as these are the main things used to place the mushrooms on the map.

This is a GET endpoint that retrieves all the captures in the captured mushroom database. It includes all the required information needed to display them in our map feature and allow the users to inspect them for more information.

Get all captures would be another endpoint that requires immediate internet connection since it would be too high cost and complex to bundle and queue it up with others and it requires an immediate response from our server. If a user continued to use the app without syncing up with our database they would have to overwrite the discoveries either stored in the cloud or locally on their phone, since there would be new information in both places. While we could hypothetically create a way to merge both storage spaces by adding together unique mushrooms and their instances, it would be a high cost implementation and would only be considered after other more high priority features were added.

Response

- 200: mushrooms successfully retrieved
- 500: Any of the database services returned errors.

Response Body

```
{
  "captures": [
    {
      "captureID": string,
      "instances": [
        {
          "dateFound": string,
          "imageLink": string,
          "latitude": int,
          "location": string,
          "longitude": int,
          "s3Key": string
        }
      ],
      "mainSort": string,
      "notes": string,
      "timesFound": int,
      "userID": string
    }
```

```
  ]
}
```

Get Captures (GET /users/<user id>/captures)

Upon login the app will need to retrieve all the captures a user has made and saved within the database. This effectively works to sync up what is locally stored and what has been uploaded to our database. The user will largely not notice this endpoint as it will be done mostly within the background during different parts of the app, primarily during login.

This is a GET endpoint that retrieves all the mushrooms in the captured mushroom database associated with a particular user.

Get captures would be another endpoint that requires immediate internet connection since it would be too high cost and complex to bundle and queue it up with others and it requires an immediate response from our server. If a user continued to use the app without syncing up with our database they would have to overwrite the discoveries either stored in the cloud or locally on their phone, since there would be new information in both places. While we could hypothetically create a way to merge both storage spaces by adding together unique mushrooms and their instances, it would be a high cost implementation and would only be considered after other more high priority features were added.

Response

- 200: mushrooms successfully retrieved
- 500: Any of the database services returned errors.

Response Body

```
{
  "captures": [
    {
      "captureID": string,
      "instances": [
        {
          "dateFound": string,
          "imageLink": string,
          "latitude": int,
```

```
            "location": string,
            "longitude": int,
            "s3Key": string
          }
        ],
        "mainSort": string,
        "notes": string,
        "timesFound": int,
        "userID": string
      }
    ]
}
```

Add Captures (POST /users/<user id>/captures)

The journal contains the full dictionary list of all the available mushrooms that the app has information on.

- From the journal the user is able to access the gallery and more details about a specific mushroom in the journal.
- When a user finds a mushroom that discovery will first be logged locally in the app and then in our captured mushrooms table for that user, along with any extra information or images they wish to include.

This is a POST endpoint that creates a new entry within the captured mushroom database. This endpoint is primarily for when the user discovers a new mushroom type not currently associated with their account, so a new captured mushroom ID must be created for the user. Afterwards any subsequent discoveries of that mushroom type can be added by updating the array of instances in the same entry

- Can be used on the same species of mushroom to append new captures to the instance array
- Our database should always either be behind or in sync with the local storage. In the event it is not the user will have the option to choose from which source to overwrite from.

Add captures is an API endpoint that does not need immediate internet connection as the user can continue to use the app as normal since they are not waiting on a response from the database. The user is also able to interact with the new changes

immediately on their local device. Since this call essentially backs up the user's changes it's easy to simply wait until the user can reach the database before doing it.

- Data will only be uploaded to our online database after a stable internet connection is established
- The app will consider whether the user has a significant internet connection before attempting to send information so as not to drain the battery, slow down the phone or consume large amounts of data.
- When no connection is detected it will wait to call the endpoint until sufficient internet connection is established where it will then create new entries for all the discovered mushrooms
- The only thing the user must be made aware of is to establish an internet connection before logging out, that way the app can update the database and save the user's progress.
- In order to prevent the loss of progress the app will create pop ups to alert the user of any potential loss of progress.

Request Body

```
{
    "captures": [
       {
          "captureID": string,
          "instances": [
             {
                "longitude": string,
                "latitude": string,
                "location": string,
                "dateFound": date,
                "ImageLink": string,
                "S3Key": string
             }
          ]
       }
    ]
}
```

Response

- 200: New mushroom successfully added
- 500: Any of the database services returned errors.

Response Body

```
{
    "message": string,
    "accessToken": string
}
```

Get Specific Capture (GET /users/<user id>/captures/<captureID>)

There may be times when we want to get more information about a specific capture when we only have the capture ID and the user ID of who we got the capture from. This way if we want to get all the mushrooms of a particular species and their stats from a user we can do so using this endpoint

This is a GET endpoint that retrieves all the information about a specific capture. This includes its stats and instance array that includes every mushroom found of this species.

Get specific capture would be another endpoint that requires immediate internet connection since it would be too high cost and complex to bundle and queue it up with others and it requires an immediate response from our server.

Response

- 200: mushroom successfully retrieved
- 500: Any of the database services returned errors.

Response Body

```
{
    "capture":
       {
          "captureID": string,
          "instances": [
             {
                "longitude": string,
                "latitude": string,
                "location": string,
                "dateFound": date,
                "ImageLink": string,
                "S3Key": string
```

```
                }
            ]
        }
}
```

Upload image to S3 (POST /users/<user id>/images)

In order for us to store our images we first need to create a spot within S3 to put them. Users will snap an image, this endpoint will be called and then an S3 provided endpoint will be called using the location of the picture to upload it to the correct place.

This is a POST endpoint that will create spots within our S3 database for us to upload our images to. Then a separate post endpoint must be ran to actually upload the images to S3, this endpoint is provided by S3

- Requests the number of spots to create and whose userID to associate them with.
- Returns a number of links to the spot to upload the image to and the S3 primary keys that's used to locate the spot in the S3 database. The amount is dependent on how many were requested.

Upload image to S3 is an API endpoint that does not need immediate internet connection as the user can continue to use the app as normal since they are not waiting on a response from the database. The user is also able to interact with the new changes immediately on their local device. Since this call essentially backs up the user's changes it's easy to simply wait until the user can reach the database before doing it.

- Data will only be uploaded to our online database after a stable internet connection is established
- The app will consider whether the user has a significant internet connection before attempting to send information so as not to drain the battery, slow down the phone or consume large amounts of data.
- When no connection is detected it will wait to call the endpoint until sufficient internet connection is established where it will then create new entries for all the discovered mushrooms
- The only thing the user must be made aware of is to establish an internet connection before logging out, that way the app can update the database and save the user's progress.

- In order to prevent the loss of progress the app will create pop ups to alert the user of any potential loss of progress.

Request Body

```
{
    "numLinks": int,
    "userID": string
}
```

Response

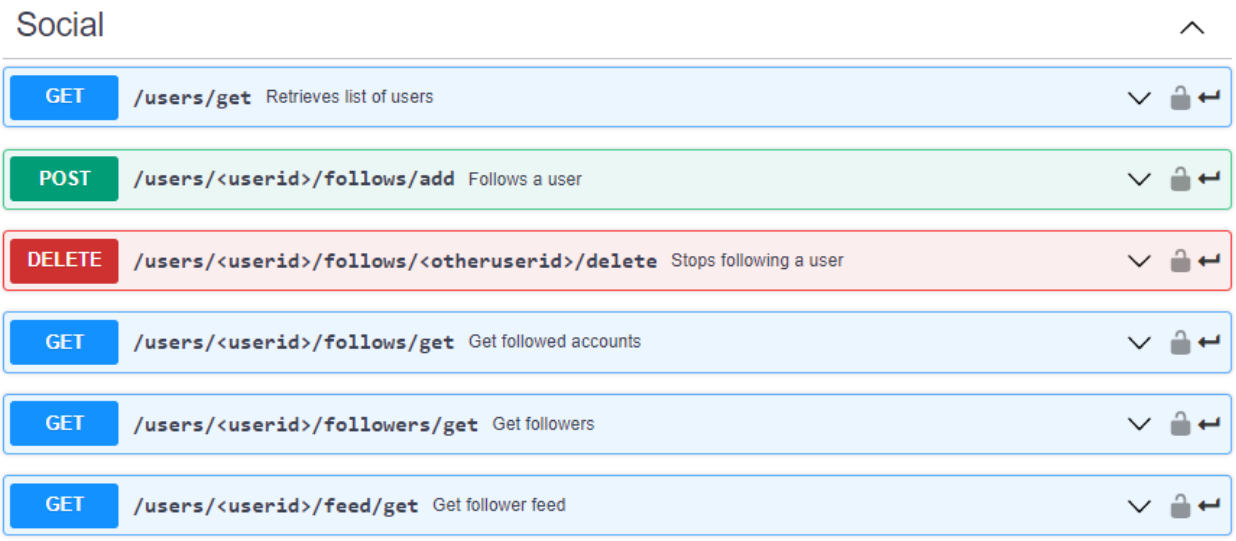
- 200: Mushroom uploaded
- 403: Bad Request
- 500: Any of the database services returned errors.

Response Body

```
{
    "images": [
       {
          "imageLink": string,
          "S3Key": string
       }
    ]
}
```

Social Endpoints

Detailed below are all endpoints related to a user's social functions that will be active upon running Polyamanita's back end server, spanning from following system functionality and user search.



Search User (GET /users)

Searching for users is the primary avenue for finding other users to follow to facilitate the social features provided by Polyamanita. This endpoint searches users according to their username, returning a list of stubs to reference for further information retrieval of a user.

Since this endpoint requires the user database in order to query other users, it requires immediate internet connection on use and thus isn't stored in batch when waiting to connect back to the internet.

Parameters

`query`: a string query username to search the user database with.

Response

- 200: Query is valid and users were successfully fetched.
- 404: Query is valid but no users matched the query.
- 401: Access token is invalid.

Response Body

```
{
    "results": [
        {
            "username": string,
```

```
            "userID": int64,
            "color1": string,
            "color2": string,
            "totalCaptures": string
        },
        {
            "username": string,
            "userID": int64,
            "color1": string,
            "color2": string,
            "totalCaptures": string
        },
        ...
        {
            "username": string,
            "userID": int64,
            "color1": string,
            "color2": string,
            "totalCaptures": string
        }
    ]
}
```

Add a Follow (POST /users/<user id>/follows)

When a user wants to follow another user to see posts in their feed or information on the leaderboard, this endpoint is used to update the users follows array to include the new account to follow.

Since this endpoint requires the user database in order to follow other users, it requires immediate internet connection on use and thus isn't stored in batch when waiting to connect back to the internet.

Request Body

```
{
    "userID": int64
}
```

Response

- 200: User successfully followed.
- 404: User does not exist.
- 401: Access token is invalid.

Remove a Follow (DELETE /users/<user id>/follows/<other user id>)

When a user wants to no longer follow another user, this endpoint is used to update the users follows array to remove the new account to follow.

Since this endpoint requires the user database in order to follow other users, it requires immediate internet connection on use and thus isn't stored in batch when waiting to connect back to the internet.

Response

- 200: User successfully unfollowed.
- 404: User does not exist.
- 401: Access token is invalid.

Retrieve Follows (GET /users/<user id>/follows)

When a user wants to look at a collection of their follows or intermediately retrieve information of their follows to find further information about them, this endpoint is used to acquire a truncated account of all of a user's follow data.

Since this endpoint requires the user database in the immediate moment in order to find other users, it requires immediate internet connection on use and thus isn't stored in batch when waiting to connect back to the internet.

Response

- 200: Follows were successfully fetched.
- 404: No users were found.
- 401: Access token is invalid.

Response Body

```
{
    "follows": [
        {
            "username": string,
```

```
            "userID": int64,
            "color1": string,
            "color2": string,
            "totalCaptures": string
        },
        {
            "username": string,
            "userID": int64,
            "color1": string,
            "color2": string,
            "totalCaptures": string
        },
        ...
        {
            "username": string,
            "userID": int64,
            "color1": string,
            "color2": string,
            "totalCaptures": string
        }
    ]
}
```

Retrieve User Followers (GET /users/<user id>/followers)

When a user wants to look at a collection of their followers or intermediately retrieve information of their followers to find further information about them, this endpoint is used to acquire a truncated account of all of a user's follower data.

Since this endpoint requires the user database in the immediate moment in order to find other users, it requires immediate internet connection on use and thus isn't stored in batch when waiting to connect back to the internet.

Response

- 200: Followers were successfully fetched.
- 404: No users were found.
- 401: Access token is invalid.

Response Body

```
{
    "followers": [
        {
            "username": string,
            "userID": int64,
            "color1": string,
            "color2": string,
            "totalCaptures": string
        },
        {
            "username": string,
            "userID": int64,
            "color1": string,
            "color2": string,
            "totalCaptures": string
        },
        ...
        {
            "username": string,
            "userID": int64,
            "color1": string,
            "color2": string,
            "totalCaptures": string
        }
    ]
}
```

Retrieve Follow Feed (GET /users/<user id>/feed)

This endpoint retrieves and compiles a list of the most recent captures from every user that the current user follows. This is used primarily when users want to look at the feed page of their followers, as it performs multiple operations of several endpoints into a concise operation to alleviate logical load on the front end.

The response body contains a list of the last 50 captures in sorted order by latest capture first. Each entry contains the user’s username, profile colors, and key path string to the captured image.

Since this endpoint requires the user database in the immediate moment in order to find other user's captures, it requires immediate internet connection on use and thus isn't stored in batch when waiting to connect back to the internet.

Response

- 200: Feed was successfully fetched.
- 404: No captures were found.
- 401: Access token is invalid.

Response Body

```
{
    "feed": [
        {
            "username": string,
            "userID": int64,
            "color1": string,
            "color2": string,
            "captureTime": Time,
            "captureImage": string
        },
        {
            "username": string,
            "userID": int64,
            "color1": string,
            "color2": string,
            "totalCaptures": string
        },
        ...
        {
            "username": string,
            "userID": int64,
            "color1": string,
            "color2": string,
            "totalCaptures": string
        }
    ]
}
```

Technologies

Golang

Go is a compiled static type programming language. The language was created by and maintained by Google, who supports it as an open source project. It was created with high readability and intuitive concurrent processing as a focus of design, allowing for easily readable and understandable concurrent code with high process speeds. The language is syntactically similar to C, maintaining C's simplicity while streamlining harder to read syntaxing with more contemporary conventions.

Polyamanita bases all of its server side code on Go, for several reasons:

- Intuitive concurrency processes allow for routing libraries that provide the ability to process multiple requests concurrently out-of-the-box.
- Simple syntax lets not just all members of the API team understand others' work quickly, but can be intuitively understood by team members writing other facets of the app like the front end.
- Golang has fast compilation times, reducing build times in continuous integration workflows
- Golang maintains fast execution speeds, allowing for faster response times when receiving requests.

Gin

Gin is an HTTP web framework written in Go. It provides simple developer interfacing, easy unit testing accessibility, and exceptional performance when compared to other http web frameworks.

Syntax

Gin provides a simple way to parse calls to separate endpoints by tying a developer-defined function to each endpoint, with syntax rules to allow for variable parameters in the path or header and body parsing. A router struct is created and used to attach different endpoints to each function that processes the request. Every endpoint method (router.POST/GET/etc) accepts a function parameter that must be a signature `func (c *gin.Context)`, wherein all call information can be retrieved by `c`.

```
router := gin.Default()
router.POST("/some/path", func (c *gin.Context) {
    // Process request
})
```

```
router.Run(":3000")
```

Testing Framework

Testing through Gin is particularly streamlined, as functions passed into each router request can be tested separately allowing for unit testing that doesn't rely on starting a server to mock endpoint calls. On an integration level, gin can also provide well by simply running the endpoints like normal and making HTTP requests using Golang's `http` library

Performance

Gin is one of the most efficient web frameworks available for Go, ruling at up to 40 times faster than the second most common Go-based HTTP web framework, Martini.

Swag

Swag is a documentation tool and converter that transforms Go annotations into Swagger Documentation. Swag includes an easy to use CLI to format annotations and generate documentation files. Swag also includes integration support with Gin to allow for easy integration with the gin-based frameworks.

Polyamanita will be using Swag to automatically generate documentation. This will greatly help bridge the gap between front and back end communication, as documentation is faster to describe in-line rather than on a separate service (like SwaggerHub) in YAML.

Annotations

Every route in the Polyamanita API will in general contain and use the annotation types listed below.

- `@Summary` - Summary statement for a swagger endpoint.
- `@Description` - A more descriptive detail of the endpoint in question, usually involving parameter restrictions or related endpoints.
- `@Tags` - The category to lump a route in with others.
- `@Accept` - The data type that the route accepts in its body (e.g. json, multipart/form-data).
- `@Produce` - The data type that the route returns in its body (e.g. json, multipart/form-data).
- `@Param` - a type of data that it passed into the route request. Can vary from body structure input, header values, or dynamic route parameters.

- `@Success` - the code and potential objects returned on a successful execution of the application.
- `@Failure` - the code and potential objects returned on a failed execution of the application.
- `@Router` - the URI route and method type to send the request to.

For example, a request to the login endpoint to the project be annotated as below:

```
// Login godoc
// @Summary     Create a new user session
// @Description Gets user credentials to auth and return tokens
// @Tags        Accounts
// @Accept      json
// @Produce     json
// @Param       request body routes.Login.LoginInput true "Login"
// @success     200 {object} routes.Login.LoginOutput "JWT Token"
// @Failure     400
// @Failure     401
// @Failure     500
// @Router      /session [post]
```

Commands

Detailed below are useful commands used to generate documentation and stylize annotations using swag.

swag init

Reads all annotations in all files routed by the gin router and driver application. Then, generates a `/docs/` folder that contains a `docs.go`, `swagger.json`, and `swagger.yaml` file that details all endpoints annotated. This includes descriptions, parameters, body expectations, and responses.

swag fmt

Formats all annotations with spacing to make reading and editing route annotations easier.

Testing

Since the back end of Polyamanita is the effective lowest level of our application logic, instability and incorrect behavior in logic found at this level can lead to devastating

problems that cascade beyond just the back end of the program. Unlike the mobile app or web app, in which any failure local to their codebase would be independently contained and unable to seriously affect each other's interactions with the back end, both are dependent on the back end server and any problem with it lead to ambiguous erroring and difficulty finding bugs for all the other parts of the app.

For this reason it's paramount that testing thoroughly covers as much logic as possible on the API, with extensive testing ranging from Unit to E2E tests. Polyamanita will contain comprehensive Unit, Integration, and End to End testing to ensure the api is not only confidently correct, but also able to handle high loads and edge cases.

Unit Testing

At the unit level, every single endpoint will be tested individually with the help of Gin's routing signatures to provide for coverage with the least amount of variability on the router end. External services like DynamoDB and S3 will be mocked using Amazon's interfaces.

Every function will be tested once for a "success" case logic flow, and then subsequently for any points in the function when failures are possible. This includes checks for authorized tokens, and external services like DynamoDB and S3.

Integration Testing

Integration testing will be used to verify code correctness for multiple endpoints used in sequence. External services will not be mocked, so as to emulate an environment closest to the actual staging/release environments. Calls are made through a go script that would start the server, make a sequence of calls to emulate a user registering, logging in, creating mushroom entries and removing such, before finally deleting their account.

End to End (E2E) Testing

End to End testing will be used to verify the server's capability to handle high loads of calls. The application will be run as a container and calls to every endpoint will be generated at a high volume for 15 minutes. This will only be run during the staging phase of the application, when the app is already deployed to Elastic Beanstalk. Any crashes during this time will prevent further deployment to the release environment of the application.

Machine Learning Specification

Machine Learning Model

One of the core components of any system that uses any amount of Artificial Intelligence (AI), or machine learning is the model. There are many types and variations of machine learning models, and each is suited to different kinds of tasks. Particularly, in our project, machine learning for the classification of fungi is one of the core features and it is a major draw for using our application. So, without proper attention and training, a poorly performing model might drastically decrease the appeal and usefulness of our app.

As with any machine learning model, our model must take in some kind of encoded data in the form of vectors that have been derived from some type of previously classified or measured data. There are a lot of ways to tweak a model to perform better or to work with different types of data, however those are the basic principles.

Our specific problem is that of classification. We need to create and train a mode that can take encoded images and the structure and biases derived from training, then return a specific classification for that particular thing. This is called image classification or recognition, and it's quite widely used in many fields. It even sees growing usage in botany and agriculture already.

This is not a new kind of problem, and there exist many examples of models built and trained specifically for the question of image classification. In general, we can look towards the relatively recent introduction of neural networks into the public and professional space as the basis of essentially all modern image classification models.

Neural networks are simple in concept. Essentially, they are entirely composed of small nodes that each perform some calculation and based on that, determine whether or not to pass their 'message' to subsequent nodes in the network. However, despite their simple components, these types of networks prove to be very difficult to analyze and comprehend at even middling scales. The inner workings of these networks can often seem like a black box where you can only understand what goes in and what comes out, but not the processes that determine the content of the outputs. That said, we still have many tools for attempting to understand what works and why. Machine learning is a constantly growing and changing field of research for a very good reason.

For image classification in particular, one of the most common models, and the one that we intend to use for this project, is a Convolutional Neural Network (CNN). This type of

model is very well suited to image recognition problems because, when properly trained and implemented, a CNN can classify many objects with a very high accuracy. However, to reach that level of accuracy, it will require an extensive amount of training and data.

Convolutional Neural Network

Given that this is an image classification problem, the most obvious and common choice is using a Convolutional Neural Network (CNN). This type of model can boast a high level of accuracy, even in the high nineties. Much like a human, a CNN is able to 'look at' an image and determine what key features exist in the image. With machine learning models, we call this process 'feature extraction'.

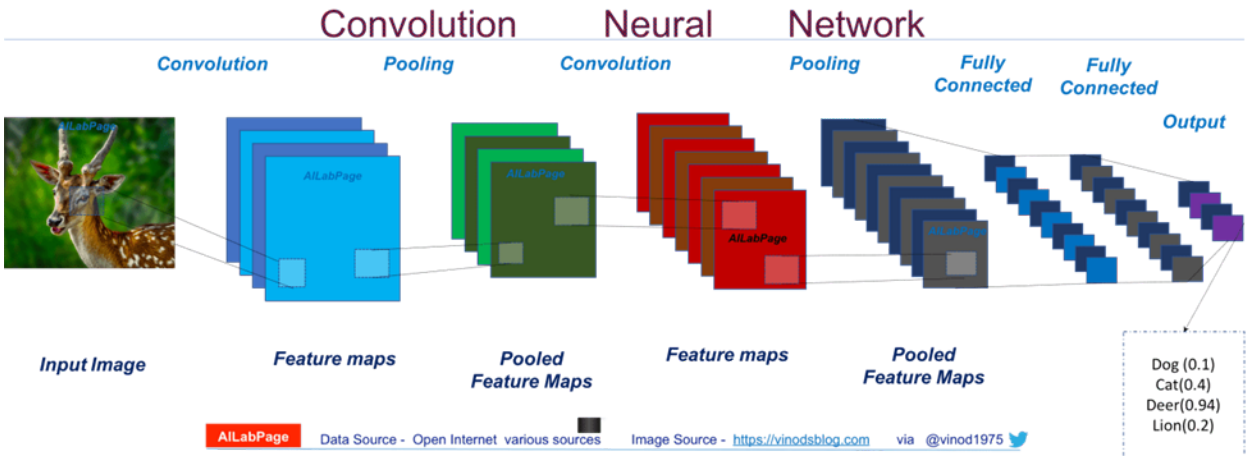


Figure 42: Diagram of a CNN

Feature extraction is done through a series of convolutions and pooling layers. A convolution is a mathematical term which, to describe simply, is the name of an operation that composes two functions together. What this really means, within the context of a neural network, is that we can create a kind of filter and apply that to our data to produce a new vector that only contains the filtered results. Once filtered, the new vectors can be recombined in the pooling layers. These pooling layers take the filtered fragments of the image and determine a 'summary' of each fragment. This summarized set of features is essentially a less detailed version of the original image, but with more pronounced features thanks to the filtering. Through multiple iterations of

this process with each iteration decreasing in size via downsampling, the model produces a ‘feature map’ of the image.

This feature map is essentially the ‘essence’ of the original image in terms of the features that the model uses. This feature map is then passed into a segment of the model called the ‘fully connected layer’ which has one input for every vector value in the final layer’s output after all of the convolutional and pooling operations. It also has a single output for every possible classification for the model. For example, if we were to build a classifier for if something is or isn’t an apple, we would have the final fully connected layer output be a vector of length two: ‘apple’ and ‘not an apple’.

The feature extraction process is interesting because it also removes a major component used in many other machine learning models that would be typically done by a human, or at least overseen by a human. The process of human feature engineering is not necessary within a CNN because the major selectors for features are the filters themselves and are accounted for by training the model and adjusting the biases via backpropagation.

The last step that gets performed within the CNN is decoding the output vector. Normally, this is a simple step and is done using a ‘softmax’ function, which takes a vector, then returns the highest value in that vector. This works because of how the fully connected layer’s outputs can be seen as a set of probabilities or confidence for each classification. So, of course, we should take the highest confidence choice from the model.

Something to note is that, depending on the input and a variety of aspects of that input, there may be multiple values within the vector that may be very similar, however using a simple softmax function would only return one of them. It may be advantageous to return multiple ‘likely’ classification for certain scenarios.

Building the Dataset

For any model, the data we use to train and test performance is possibly the most important aspect of the system. With CNNs especially, the quantity and quality of images in the training set is hugely important.

We searched for a pre existing dataset that would suit our needs early on and we came across a few such as one from the University of Central Irving’s Machine Learning Repository which only contained descriptions of a hypothetical set of mushrooms, as

well as many others which all shared the same issue for us: they were not composed of images of mushrooms. They almost all consisted of purely text-based descriptions of the mushrooms, which, for our purposes, is unusable.

To handle this problem, we have decided the best course of action is to generate our own database of images and ensure that they are labeled correctly. Proper labeling is very important as CNNs are designed to handle labeled data and any performance evaluation metrics we use will be based on correct labels, so unlabeled data simply will not do.

Of course, taking thousands of images of mushrooms in a period of a few months with only 6 people who also must work on developing this project is completely infeasible and entirely out of the question at this time. However, it would be ideal if, in the future, we were able to gather our own collection of curated and properly labeled images to use and possibly share online for other projects, models, and even research to utilize.

Rather than collect them by hand, we originally intended to use a web scraper built using Selenium to collect images that we will have to curate for our training set. The details on that web scraper can be found in the Selenium section**.**

Despite those intentions, what we ended up doing using GBIF and Kaggle was quite a bit more straightforward.

Size and Classes of the Dataset

With deep learning models, we tend to treat their training data as ‘never enough’. This is because for these models, possibly the most important key to success is the data we train it on. Without enough data, we can’t expect the model to be able to generalize to the real world. Similarly, if that data isn’t good, then we can’t expect the model to be able to ‘learn’ any useful things from it. Thus, the number of data points and the quality of every point is crucial for our model.

Tied to this fact, one of the major hurdles we expect to face with training the machine learning model is that we will not be able to collect enough high quality data within the timespan allotted for this project to train the model to a serviceable level of accuracy. There is a direct correlation between the number of images in a training set’s batch size and the level of accuracy a CNN can achieve within a number of iterations [4]. As an attempt to create a large and varied number of batches, we must collect a large number of varied images to train on.

We were originally planning on using a list of common species of mushrooms local to Florida and are from the University of South Florida's Species Project (Florida Fungi) [5]. This list contains one hundred and one different species of mushrooms from different families that are common to Florida. There are over two thousand different species in Florida alone, so this is far from an exhaustive list of mushrooms, but it would be a good starting point for this project given the resources and projected timeframe.

One concern regarding this selection of mushrooms is that it is composed of a small number of species and that it is specific to our state. While this is a valid concern for a general 'Mushroom Identifier' model, this implementation is effectively a proof of concept, and we intend to make it scalable for further increases to the size of the dataset it is trained on. Another thing to note is that, since this model is being programmed for use with this project's app, it is better that we can get results from local users and that users can potentially identify many mushrooms they find in this region before adding more. It would be unfortunate to have a large portion of the mushrooms on the list inaccessible to users because we included mushrooms from many other regions.

After some further research, we did come across a much more scalable and exhaustive dataset from the Global Biodiversity Information Facility. The section regarding the specific details of this dataset can be found in the GBIF sections. We also utilized images from an open source website that hosts machine learning competitions called Kaggle.

The next obvious question is that of size. How many data points do we need per class to produce a sufficiently accurate classification model? This topic is still up for debate in research circles and is a very important one for deep learning in general. Given our limited time and resources, however, we intend to gather at least one hundred images per mushroom as a simple starting point. This would mean that the initial dataset would be over ten thousand images in total.

We planned on using a web scraper that was built by one of our members, however, the discovery of the GBIF dataset made it somewhat moot. Instead, we ended up writing a few python scripts that utilized Pandas and some building functions for collecting, manipulating, and downloading the data from their website.

Training, Validation, and Testing

Splitting the data into testing and training data is done very commonly throughout the world of machine learning, and the ratio that we use to perform the split matters. The reason we separate training data from testing data is to avoid overfitting. If we didn't perform this split, there is a chance that our model will become exceptionally good at predicting the subset of data within the set we had gathered, but it would likely perform extremely poorly on real world data.

With Deep Learning Networks especially, the issue of memorization is a real problem. If trained on too little data, our model could potentially memorize all the training data, which would cause the same issue as overfitting. So, in an attempt to mitigate overfitting and memorization, we will split the data as such.

This ratio of testing to training data is a small, but impactful decision for our model's performance in testing and for predicting its effectiveness on real world data. We need enough data to train on so that the model is able to sufficiently generalize the traits of different classes, but enough testing data to also measure the model's performance accurately in a contained setting. With this implementation, it will be very hard to directly measure accuracy in the field, so the evaluation of our model is more important than with models that can receive 'real world' feedback.

There have been studies that have shown a correlation between a model's maximum accuracy and the ratio of training to testing data. As a good rule of thumb, one study gives the recommendation of 2:1, or 66.7% training data and 33.3% testing data. By default TensorFlow uses an 80% training split, so it will be easier if we just use the default split which gives us 20% for validation testing on our model.

We also plan to use random sampling within our dataset to help counteract the possibility of memorization. So, on each iteration of training, specifically each epoch (a full cycle through all of the data), we will be redividing the data using a random shuffle to match our training and testing data split.

One of the common methods to help prevent overfitting for CNNs is to augment the training and validation data in various ways during the training process. Our training script uses random pixel swapping and minor rotations. The purpose of this is to prevent the model from memorizing the training set and leading to overfitting. If we augment all the images it sees during training, then it's actually training using slightly different sets of images on every epoch.

Outside of training and validation, the last time we run the model across any images before porting it to TF Lite is via a testing process. When we are training the model, we use data augmentation to try and prevent the model from overfitting. However, this also affects the validation data. This means that both training and validation are not quite as ‘real’ as the original data. To actually test the model on ‘real’ data, we then run it on the entire original dataset that it was trained on with no augmentation and no training, just to gauge actual performance.

Data Collection and Methodology

When creating a dataset of images for any project, a major aspect of preparing for training and testing is how you plan to collect the images of the dataset. Thanks to Fair Use laws, we are able to use pictures gathered online to create our dataset, and thanks to many available online datasets, they allow us to use their sets for all kinds of training. While originally planning to use our web scraper, we plan to use GBIFs extensive dataset of species to collect our images.

An important issue we must deal with is how we can determine which of the returned images have valid content for our purposes. For example, in some of our preliminary testing of the web scraper, we had several images which were not mushrooms at all as well as several that looked vastly different from images we had seen from trusted sources. Even when using GBIF’s dataset, there are many images that have plain backgrounds, which do not match up with real world data, or they contain extra things like measuring sticks and other similar tools. It is likely not feasible for us to write any kind of algorithm or train a fresh model to help with this process given the timeframe we have and the complexity the task would likely add. Instead, we must manually select for images that seem to be within the range of expectations when compared to the greater set.

The issue of varying conditions in the dataset is also important for several reasons. Most importantly, if we were able to incorporate many varying conditions in the set, the resulting model would likely be much more robust and less sensitive to various conditions like dim lighting or rain. Another reason is because of how this model is intended to be used. We want the model to be used out in the real world and on many possible devices. Things like image quality and lighting will vary wildly based on device, so if we can avoid device dependent issues, then the model would be more useful to our target audience. A specifically important condition is the angle from which the photo is taken. For certain species of mushroom, sometimes, the most important factor for

identification is the underside of the cap called the “gills”. There isn’t a way for us to account for pictures of those mushrooms taken where the gills aren’t visible, but they would play a large role in identifying types where those are important. So, it is important that we try to include pictures from many angles in an attempt to account for those types of features inherent to different species.

That said, it is likely that the majority of images that our model would be used on in the real world will likely be from a higher angle, thus the gills would not be visible. This definitely isn’t something that we can do anything about since it’s just a product of users taking pictures that they want, rather than trying to take optimal pictures for the model. So, maybe, the fact that we cannot account for many instances like that may not have a significant impact on performance of the model. This is difficult to accurately predict or test, but in the case of poor performance, we may be able to get away with replacing pictures that explicitly feature the underside of the mushrooms given that we know what the standard image will likely be.

We could also potentially consider user surveys to gain insight on the user base for what angles, conditions, and other related features that they tend to use our model in conjunction with.

The creation of the training and testing set is possibly the single most important factor in how accurate the resulting model would be. So, we must try our best to set up for success now so that the resulting training and tweaking does not include reevaluating our input data.

Outcome of the Model

Due to the limitations of our dataset, there was a natural cap to how many mushrooms we could have in the model. Especially since we wanted really high performing models. Over the course of the semester, the models we trained had its ups and downs. Most of the downs weren’t actually related to poor model architecture, but the data itself. I think we amassed over 1 million images over the semester from all the tinkering; with each set of mushrooms being cleaned. At the very start, the accuracies were below 10% with offshoots at around 10% due to overfitting. However, by the end the data we used on our final model was not only clean, but showed some flaws in our project as a whole. In the demo we explained this in detail, but will re-explain here. Our final dataset had two Pleurotus mushrooms: Pulmonarius and Ostreatus. Both of the mushrooms not only looked similar, but to the laymans eye they are identical.The differences between the

two are based on the color of the cap, which tends to fold on top of itself as the mushroom matures. By the time we realized this, there was nothing we could really do; so, we left it in as a learning experience showcase.

Figure 43: Pleurotus Pulmonarius (left) and Pleurotus Ostreatus (right)

Regardless, by the very end, prior to changing the dataset one last time to account for another live demo mushroom, we achieved really solid results on mushroom scores. The end average was around 88%, and for the limitations of the dataset this really wasn’t bad. We even tested it on 2 different mushrooms in person and achieved pretty consistent true positive results.

	class	total_correct	total_incorrect	total	accuracy
0	Chlorophyllum_m	612	128	740	82.7027027
1	Clathrus_columnn	620	45	665	93.23308271
2	Coprinellus_diss	539	93	632	85.28481013
3	Coprinopsis_lagc	596	202	798	74.68671679
4	Cylindrobasidium	339	25	364	93.13186813
5	Ganoderma_pfeif	322	20	342	94.15204678
6	Hortiboletus_rube	578	64	642	90.03115265
7	Leucocoprinus_b	499	171	670	74.47761194
8	Leucocoprinus_fr	571	43	614	92.99674267
9	Lycoperdon_mar	519	88	607	85.50247117
10	Mycena_galericu	356	20	376	94.68085106
11	Omphalotus_sub	624	57	681	91.62995595
12	Phallus_rugulosu	682	48	730	93.42465753
13	Pholiota_adiposa	678	69	747	90.76305221
14	Pleurotus_pulmo	636	160	796	79.89949749
15	Pycnoporus_cinr	689	94	783	87.99489144
16	Scleroderma_citr	507	87	594	85.35353535
17	Trametes_lactine	759	37	796	95.35175879
18	Tricholoma_scalp	342	72	414	82.60869565
19	Xerocomellus_ch	390	16	406	96.0591133

Figure 44: Result of model analysis.

Future Expansion of The Dataset

In the potential future case, where we would be expanding this dataset using a greater range or the wider internet, there are several things that require attention and consideration. Our options for future expansion are somewhat nebulous currently, however there definitely are certain options that are more likely with our current process.

Primarily, we can use the GBIF dataset to expand our region to the greater United States, then possibly continue growing continent by continent. Or, maybe, it would be better to grow from multiple ‘hotspot’ areas across the globe. This is just speculation; however, I believe it would be a better idea to grow from one point rather than many.

The other much more difficult and ‘dirty’ option is to use the web scraper as we had intended to use it in the first place. It’s ‘dirty’ because the data itself would be extremely difficult to clean. From a data engineering perspective, it has many, many more risks.

As we do not have access to, nor are any of us experts on mycology, we will likely make some mistakes and we may not notice the subtle differences between pictures of similar mushroom species. While we would try our best to be as accurate as possible, one of the benefits of using a CNN is that it will be able to generalize the features of those particular mushrooms, assuming we provide enough useful data.

So, for example, in the case where we have a mushroom that is distinctly orange and bulbous, we may accidentally add pictures of a mushroom that is also orange, but skinnier. They are undoubtedly similar, but if we are able to gather a majority of pictures of the correct one, then the algorithm will pick up on the important features and hopefully mitigate the effect of incorrectly classified training images. It is very pertinent that we do actually gather significantly more correct images than incorrect ones because the fewer correct ones we have, the more likely that the model will try to classify both as the same.

One major possible flaw in our attempt to correctly classify the examples is that we are trusting the web scraper to return correctly labeled information most of the time. We cannot guarantee this to be true because we would need to individually check the context of every single image, which would be exceedingly time intensive.

Another major foreseeable issue is the possibility of introducing our own biases into the dataset, primarily as a result of us using a set of references for each class. What we consider “similar enough” to the references may vary from person to person and it may also vary based on many other factors acting on each person. Another reason this arbitrary measure of closeness matters is because of how certain mushrooms look vastly different at different stages of their growth cycles and even in different times of the year.

This is something that a model can be trained to account for, but it would likely require a much larger dataset as well as many more iterations during training. It would be a very good idea to expand this dataset to include those kinds of variances, although this would introduce a lot of extra work in training and gathering the data. An obvious necessity resulting from that choice would be that we would require more people to help with curating the data or developing a model to assist with gathering and curating.

Kaggle

Kaggle is an open source and free to use website that hosts machine learning competitions and provides many resources for those interested in machine learning in general. One of the key features of these competitions is that they typically come with prebuilt, clean, and labeled datasets. This is a very useful resource for kickstarting training without the need for going through the whole process of collecting, cleaning, and preparing the data from other sources.

Global Biodiversity Information Facility (GBIF)

The Global Biodiversity Information Facility is an online collection of documented species from across the globe. Their major goal is to catalog and provide easy to access information and documentation for anyone to view and use for research and educational purposes. They have (at the time of writing) well over two billion recorded instances of species from all over the world. They are an incredible free resource funded by multiple governments and they also provide a decently documented API for accessing their records. One of the major resources that they use is actually other online datasets hosted by third parties.

Really, the GBIF dataset is a collection of not just their own documents, but so many more from other parts of the internet. This collaboration makes it fantastic for projects just like ours where we need data collected by people who are familiar with subject matter. The fact that it also connects other databases together allows for each of those databases to have their own way of being interacted with by their community, but then shared with GBIF, is great as it only serves to benefit everyone else.

Collecting Data From GBIF

For this project, we focus on fruiting bodied mushrooms, so we had to figure out which broader classifications of fungi we could use to get mushrooms that we could actually use with our model. After doing some research, the classification we found is within the following classification:

- Kingdom: Fungi
- Division: Basidiomycota

- Subdivision: Agaricomycotina
- Class: Agaricomycetes

Generally speaking, any species under this classification has a fruiting body and thus we can see it with our eyes and use it in our model.

We have two options for getting the occurrences data from GBIF: PyGBIF and using their manual download options.

PyGBIF

PyGBIF is a python wrapper library developed and provided by GBIF as a method of interacting with their APIs autonomously via scripting. We did try to use this to determine which mushrooms actually have over one hundred images, but as far as we could find in the documentation, doing an occurrence query of that kind just wasn't possible via the PyGBIF module. We could try to query for all metadata for the occurrences within a certain range or polygon that we could define, however that also doesn't work because it limits us in the number of results to a couple thousand. For context, the number of all mushrooms in the region we were looking at contained approximately 56,000 results for only the occurrences that were paired with pictures.

So, after scraping through the documentation for it, we realized that we would have to figure out a different way of getting what we needed.

GBIF Manual Downloading

The GBIF website has the option of making an account and then using that account to set up downloads for a range of query options. The major restrictions we wanted to add were the taxonomic one, to ensure we only get fruiting bodied mushrooms, and a geographic range.

Our original plan was to use a list of mushrooms specifically in Florida to help limit our scope, however, this turned out to be much too limiting. Essentially, that list of 'common mushrooms' were just too hard to scrape for without heavy supervision and curation, so when we found GBIF, we realized that we could instead perform some kind of query to determine which mushrooms in our specified region actually had enough images in their database to use for training. So, that's what we did.

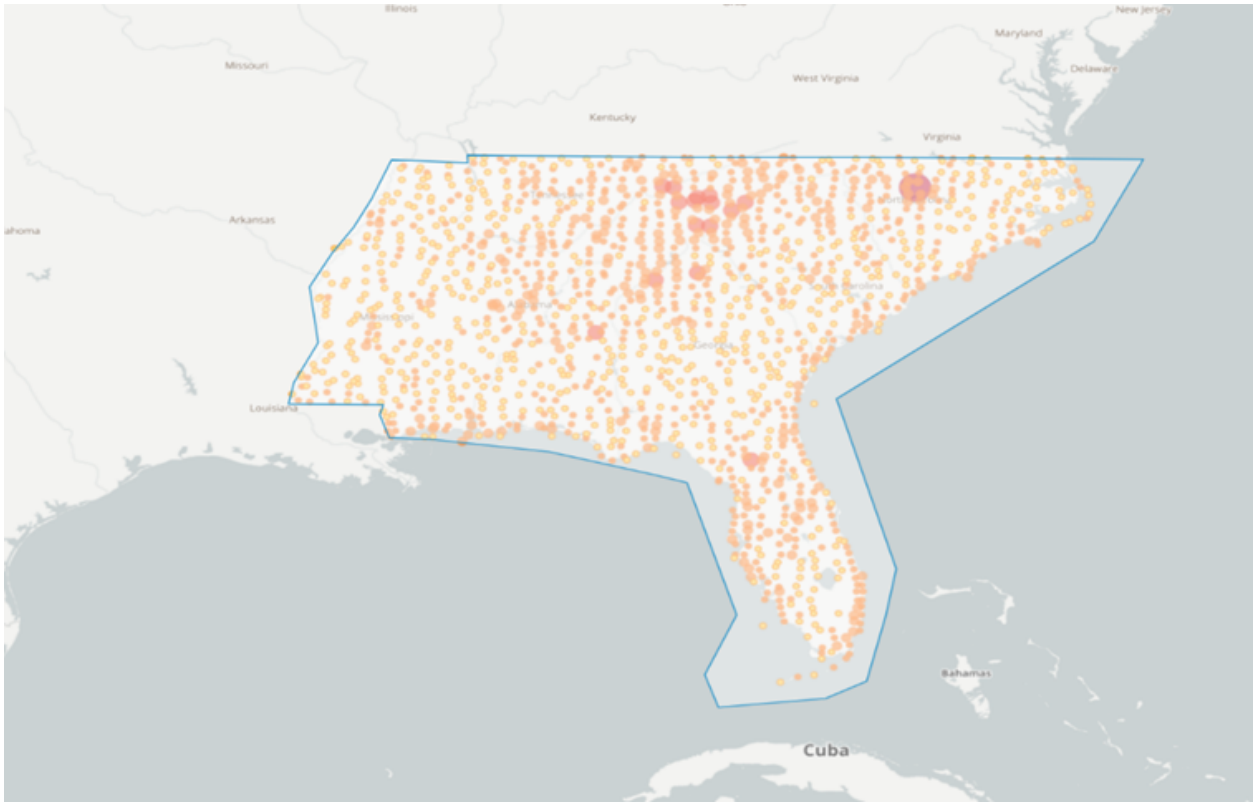


Figure 45: Region that we queried for on the GBIF website.

All of the small dots represent some number of occurrences that meet our criteria. This query contains over 120 thousand occurrences from which we can continue to add requirements to fit our needs.

Only specifying region and taxon doesn't ensure that each occurrence contains a related image, however. So, we must also add the constraint for 'Still Image' as the media type we're looking for. Once we perform this extra query, we get about 56,200 distinct occurrences that have images tied to them. This is the data we need to sort through.

To download this data, all we need to do is make an account, then click the 'Download' button and we have a few options for what kind of download we want from this query.

Something to specifically note is that downloading from GBIF does not actually include the images in the download. I assume this is because collecting and packaging something with over 56,000 images is extremely computationally expensive, and since the images already exist on the internet, we can just figure out how to download them ourselves. So, instead, they only provide links in the data we downloaded.

In the end, after all of our training iterations, the scope was narrowed down even further to only include 20 species from this dataset. The main reasons were due to poor performance across larger subsets as well as not having enough images per class to properly train on any larger subset. The final dataset contained about 16,000 images primarily composed of images sourced from GBIF and a small portion from Kaggle.

Probing and Processing the GBIF Results

After collecting the data from the GBIF website, we're still left with 56 thousand possible images to choose classifications from. What we needed to do was to write a script to run some statistics and determine which species of mushroom are worth using in our final dataset.

The way the results from the download are structured is a little strange, however. The information that we need is split across 3 separate files, so that requires me to collate those files in such a way that we can get species occurrences, select which species we want pictures from, then subset the occurrences data to aggregate the links, which can then be downloaded by another script.

While this solution is a bit messy, it does work. The script I wrote uses the CSV file of species metadata to count up how many occurrences of each species are contained within this set. The results of that tell me that there are 1307 species that are distinctly marked according to GBIF. Performing a simple sort and slice returns 109 unique 'taxonKey' values (the values that distinguish each species) that have over one hundred occurrences with images.

This is the part of the script that performed this search:

```
# Sort using the values as numbers, not objects/strings

sort_order = query['numberOfOccurrences'].astype('int32').argsort()



# Sort the queries by occurence number and then reverse it because argsort
is always ascending order

taxons = query.iloc[sort_order][query['numberOfOccurrences']

     .astype('int32') > 99][::-1]
```

```
      .dropna(subset=['species'])
```

After collecting the taxonKey for each of those species, we need to then get the desired one hundred image links per species and organize them via their Key and species names.

This was a little difficult to figure out because, while one of the other two files uses taxonKey to identify the species, the other one only uses a tag called ‘identifier’, which is something different in both of the files. So, we had to figure out which one corresponds to the file with IDs for the links.

Following that, we have to then collect the first 100 images of each species and then generate a CSV to store the useful data for the downloading script to read in. We also included a filter so that any nonstandard image files were excluded, and since the majority of the files are JPG or JPEG, we decided on simply only allowing those two extensions. Also, JPG and JPEG are literally the same file type, but for some reason they have two allowed extensions, which is really weird.

The part of the script that performs the link aggregation with comment explanations:

```
# Initialize the empty DataFrame

links = pd.DataFrame(columns=['key',  'species', 'link'])



# Loop through and grab 100 rows per taxon

for i in range(len(taxons)):



    # Subset occurrences to get just the first 100 matching rows

    current_taxon = taxons['taxonKey'].iloc[i]

    subset = occur_media.loc[occur_media['taxonKey'] == current_taxon]

                        .dropna(subset=['identifier_y'])
```

```
    # Initialize a (100, 3) array to fill
    arr = np.ndarray((100, 3), dtype='object')
    count = 0


    for j in range(len(subset)):
        # Get the file extension
        url = subset.iloc[j]['identifier_y']
        root, ext = os.path.splitext(url)


        # Filter out any files that dont have .jpg or .jpeg extensions
          for uniformity
        if ext == '.jpg' or ext == '.jpeg':


            # Take the useful values from the current row and insert
              them into the ndarray
            row = subset.iloc[j][['taxonKey', 'species', 'identifier_y']]
            arr[count] = [
                row['taxonKey'],
                row['species'],
                row['identifier_y']]
            count += 1


        # Filled array
        if count >= 100:
```

```
            count = 0

            break



    # Convert the filled ndarray into a DataFrame

    sub_100 = pd.DataFrame(arr, columns=['key',  'species', 'link'])



     # Concatenate the DataFrame

    links = pd.concat([links, sub_100])



# Reset and drop the indices

links = links.reset_index()[['key',  'species', 'link']]



links.to_csv('links.csv', index=False)
```

After all of that script runs, we can just export the DataFrame as a CSV and send it to the downloading script.

	A	B	C
1	key	species	link
2	2548311	Trametes versicolor	https://inaturalist-open-data.s3.amazonaws.com/photos/243561424/original.jpg
3	2548311	Trametes versicolor	https://inaturalist-open-data.s3.amazonaws.com/photos/243442659/original.jpg
4	2548311	Trametes versicolor	https://inaturalist-open-data.s3.amazonaws.com/photos/243504836/original.jpg
5	2548311	Trametes versicolor	https://inaturalist-open-data.s3.amazonaws.com/photos/243504832/original.jpg
6	2548311	Trametes versicolor	https://inaturalist-open-data.s3.amazonaws.com/photos/243504840/original.jpg
7	2548311	Trametes versicolor	https://inaturalist-open-data.s3.amazonaws.com/photos/243460559/original.jpeg
8	2548311	Trametes versicolor	https://inaturalist-open-data.s3.amazonaws.com/photos/242664208/original.jpg
9	2548311	Trametes versicolor	https://inaturalist-open-data.s3.amazonaws.com/photos/242664222/original.jpg
10	2548311	Trametes versicolor	https://inaturalist-open-data.s3.amazonaws.com/photos/242664198/original.jpg
11	2548311	Trametes versicolor	https://inaturalist-open-data.s3.amazonaws.com/photos/242664236/original.jpg
12	2548311	Trametes versicolor	https://inaturalist-open-data.s3.amazonaws.com/photos/243346306/original.jpg
13	2548311	Trametes versicolor	https://inaturalist-open-data.s3.amazonaws.com/photos/243346285/original.jpg
14	2548311	Trametes versicolor	https://inaturalist-open-data.s3.amazonaws.com/photos/243346295/original.jpg
15	2548311	Trametes versicolor	https://inaturalist-open-data.s3.amazonaws.com/photos/242424481/original.jpeg

Figure 46: The resulting CSV file.

Downloading The Images

After generating the links along with their species names, the next major step was to actually download all of them and use a standardized naming convention for ease of use (also to rename all of them to have the same file extension).

Once we loaded in the CSV, we noticed that there was a small peculiarity regarding the ‘taxonKey’ and the ‘species’ fields that were in the generated file. After a little probing, it turns out that, internally, there are two separate taxonKey values for a species sharing the same exact string name. I’m not sure whether or not that is intentional or not, however, it means that there is a single redundant classification that we don’t care about, so the script is hardcoded to handle that duplicate.

Following that, the script goes through the almost eleven thousand links in batches of 100 to download and name them all using ‘species_name_XX’ where the ‘XX’ is a number from zero to ninety nine. This will be useful to the machine learning model since each image will be named using its classification and an identifying value. After each batch, it moves to the next and restarts the count and uses the current batch’s classification.

The script is intended to download all of these images to the Google Drive where it it’s being run, so the images themselves will have to be exported for offline training.

NumPy

API Reference

[NumPy Reference — NumPy v1.23 Manual]

What is NumPy?

NumPy is a python library that is built with efficiency and vector math in mind. It provides a special type of array structure that is extremely useful and fast for operations

across it. It also provides several functions for doing statistics, linear algebra, and many other useful operations using their array. Primarily, it will be used for performing operations within and in tandem with our neural network as a CNN needs a lot of simultaneous matrix operations to be performed at basically every step.

Why NumPy?

NumPy is a very fast library for the large computations it supports. The reason it's so fast is because on the back end, it uses precompiled C code. This lets it be much more efficient than Python tends to be as Python deals mostly in high level objects as opposed to low level primitive data types as C does. The C code is also vectorized meaning that it runs without loops, which also helps the speed and efficiency.

The NumPy library also is much more math friendly in terms of writing out complex operations like the ones we may need to implement for this project. It makes it much easier to correctly write out these operations and makes it more readable in turn.

It also helps that it is one of the most widely used Python packages and there are a vast number of resources to reference and use.

While NumPy is incredibly useful strictly due to the ability to use large and space efficient matrices using the ndarray class, the other component of it that makes the package an absolute 'must have' for any project involving linear algebra is their linear algebra module. It allows for high level complex operations and even has options for performing multiple operations simultaneously for certain functions. NumPy really is the core of this and many other machine learning projects.

N-Dimensional Array

NumPy's ndarray is a container class that functions similarly to your classic single-typed array in other languages and in python. It has an associated shape attribute that represents the maximum size of the array. Internally, this is stored as a tuple in the form (rows, columns). Since it is meant as a replacement to your typical array of one type, it of course supports indexing (selecting specific components) and slicing (selecting a specific subset of components). This makes it extremely easy to select, analyze, and manipulate the values within the ndarray.

One extra thing to note is that, while NumPy does have a Matrix class that can be used, the NumPy docs make it clear that it is better to use a 2-dimensional ndarray rather than a Matrix object. So, for our purposes, a ‘matrix’ will be referring to a 2-dimensional ndarray.

Useful Functions and Methods for handling ndarrays

* ‘ndarray’ is the name of an arbitrary ndarray object used for method purposes

- **numpy.all(<ndarray>, …)**
    - Returns a boolean if all values along the specified axis of the given ndarray evaluate to true.
    - If **axis** is not specified, it will check all axes in the ndarray.
    - If **out** is specified, it will instead return a reference to the specified array.
- **numpy.any(<ndarray>, …)**
    - Returns a boolean if any of the values along the specified axis of the given ndarray evaluate to true.
    - If **axis** is not specified, it will check all axes in the ndarray.
    - If **out** is specified, it will instead return a reference to the specified array.
- **ndarray.copy(order='C')**
    - Returns a copy of ndarray using the given order specifier.
    - **order** only affects the internal storage structure of the values in memory
    - ‘C’ represents C-Order. The default is ‘C’.
- **numpy.cumsum(<ndarray>, …)**
    - Returns the cumulative sum (running total) of the values along the given axis.
    - If **out** is specified, it will instead return a reference to the specified array.
    - If **axis** is not specified, it will use all axes in the ndarray in its calculation.
- **ndarray.dump(<file>)**
    - Stores a pickle, or serialized object, of the array in the specified file to be deserialized later.
    - The **file** parameter can be a string or a Path object.
- **ndarray.dumps()**
    - Returns a pickle, or serialized object, of the array as a string.
- **ndarray.fill(value)**
    - Fills the ndarray with the given **value**.
- **ndarray.flatten(order='C')**
    - Returns a flattened one-dimensional array using the given order style.

    - **order** only affects the internal storage structure of the values in memory
    - ‘C’ represents C-Order. The default is ‘C’.
- **numpy.amax(<ndarray>, …)**
    - Returns the maximum value along the given axis.
    - If **axis** is not specified, it will check all axes in the ndarray.
    - If **out** is specified, it will instead return a reference to the specified array.
- **numpy.amin(<ndarray>, …)**
    - Returns the minimum value along the given axis.
    - If **axis** is not specified, it will check all axes in the ndarray.
    - If **out** is specified, it will instead return a reference to the specified array.
- **numpy.mean(<ndarray>, …)**
    - Returns the arithmetic mean of all the values along the given axis.
    - If **axis** is not specified, it will use all axes in the ndarray in its calculation.
    - If **out** is specified, it will instead return a reference to the specified out array.
- **numpy.ptp(<ndarray>, …)**
    - ‘ptp’ stands for ‘Point to Point’
    - Returns the range (maximum - minimum) of the values along the given axis.
    - If **axis** is not specified, it will check all axes in the ndarray.
    - If **out** is specified, it will instead return a reference to the specified array.
- **ndarray.resize(<new_shape>, refcheck=True)**
    - Changes the shape (and size) of the array object.
    - It flattens the array, then adds, or removes, values to match the new size, and finally rebuilds the array in the new shape.
    - **new_shape** is a tuple or list of integers that represent the new shape of the array.
    - **refcheck** is an optional parameter that relates to counting references. If set to False, it may potentially raise errors.
- **numpy.std(<ndarray>, …)**
    - Returns the standard deviation of the values along the given axis.
    - If **axis** is not specified, it will use all axes in the ndarray in its calculation.
    - If **out** is specified, it will instead return a reference to the specified array.
- **numpy.sum(<ndarray>, …)**
    - Returns the sum of the values along the given axis.
    - If **axis** is not specified, it will use all axes in the ndarray in its calculation.
    - If **out** is specified, it will instead return a reference to the specified array.

numpy.linalg

NumPy's linear algebra module is filled with powerful and fast implementations of many linear algebra algorithms. Given that we're working with a neural network, using these functions will substantially help with reducing runtimes and keeping training and testing times low.

An important detail mentioned in the documentation regarding this module is that many of the functions accept array_like data as parameters. When passed this kind of data, it treats it like a stacked set of matrices. This means that it is possible to use a single function call to perform the same or chained operations across multiple matrices within a single function call. This also means that, for some, the returned values will be a list of the stacked solutions.

For more notes on this, see the specific API pages for a particular function as different functions treat stacks in different ways depending on the operations they implement.

Useful Functions

- **numpy.dot(a, b, out=None)**
    - Returns an ndarray representing the dot product of the two passed arrays.
    - If **out** is not specified, it will allocate a new ndarray to return.
- **numpy.linalg.multi_dot(arrays, *[, out])**
    - Works similarly to numpy.dot, however it takes a variable number of arguments for matrices and performs an algorithm to select the fastest multiplication order via parenthesizing the multiplications to reduce the number of multiplications.
    - If **out** is not specified, it will allocate a new stacked ndarray to return.
- **numpy.inner(a, b, …)**
    - Returns an ndarray representing the inner product of the two passed arrays.
    - If **out** is not specified, it will allocate a new ndarray to return.
- **numpy.matmul(x1, x2 [, out, ...])**
    - Returns an ndarray representing the matrix cross product of the two passed arrays.
    - If **out** is not specified, it will allocate a new ndarray to return.

    - **Note**: We should avoid directly calling this function because internally, the symbol '**@**' is used and NumPy uses it to represent matrix multiplication. So, instead of doing `np.matmul(A, B)`, do `A @ B`.
- **einsum(subscripts, *operands [, out, ...])**
    - Does some wacky stuff.
    - **subscripts** is a string that is parsed by the function using Einstein summation notation to perform complex matrix-vector operations.
    - **operands** is the variable number of ndarrays that will be used in the operations performed described by the **subscripts** string.
    - If **out** is not specified, it will allocate a new ndarray to return.
    - I'm still trying to figure this one out, but it's very useful. As it stands, however, it might as well be magic thought up by the Stack Overflow Gods.
- **linalg.matrix_power(a, n)**
    - Returns a ndarray representing the result of multiplying a matrix by itself n times.
    - If **out** is not specified, it will allocate a new ndarray to return.

Universal Functions (ufunc)

It is important to note that the many math operations that work with ndarrays are all element-wise operations, unless specified otherwise in the official documentation. It is important to remember this, as there are some functions where it may produce confusing results as it is not the expected type of operation (e.g., the product of two column vectors would produce the inner product, not their cross product when using numpy.multiply()).

There are many universal functions, but they are all mostly self-explanatory due to being mathematical operations, so they will not be enumerated here.

Pandas

API Reference

[API reference — pandas 1.5.0 documentation (pydata.org)]

What is Pandas?

Pandas is a python library built using NumPy that is commonly used for data preprocessing, data manipulation, data analysis, and data visualization. It focuses on usage in any statistics-oriented applications, as it is very well suited for them. Given that we will be using it alongside a convolutional neural network, we won't have any real preprocessing and cleaning to perform on the data that Pandas can assist with. As such, we will primarily use it for analysis during the training and testing iterations as well as structuring the data using Pandas' DataFrame objects.

One other major component of Pandas that will be useful to this project is pickling. Essentially, pickling is a goofy term for Pandas' serialization method, which would make it significantly easier to train in separate sessions or to store objects for reuse or comparison later on.

Why Pandas?

Pandas is a very popular library for statistics and machine learning applications. It is very well documented and has many useful functions for our project. The Primary component that will serve the most value for us is their DataFrame object and all its associated functions that allow us to run calculations on large sections of data.

Another reason Pandas is great is because it was built using the NumPy library, which is very well known for making things like vector-matrix operations and other linear algebra operations exceptionally fast. As the data we will be performing analysis on will consist of many large vectors, it is ideal for us to use a package that utilizes NumPy.

Pandas also has some simpler analysis-oriented functions that use Matplotlib for data visualization, which can make it even easier to see progress and analyze trends of our model.

DataFrames

(ndarrays, but easier to manipulate)

Essentially, DataFrames are NumPy's ndarrays (In fact, it is built using them), but with more useful features for general usage relating to data science.

One major difference between an ndarray is that DataFrames have labels that can be used for indexing them as you would with a row or column number (e.g., 0, 1, 2, …, n). Another one is that they aren't *quite* composed of ndarrays, but rather, they are made of Series objects, which are one-dimensional ndarrays (Close enough). Because of this, we can use multiple different data types (dtype) within a single DataFrame. So, we could store boolean, integer, and floating-point values all within a single DataFrame, which is normally impossible when using ndarrays.

Attributes

* 'DataFrame' is the name of an arbitrary ndarray object used for method purposes

- **DataFrame.index**
  - The row labels of the DataFrame as strings.
- **DataFrame.columns**
  - The column labels of the DataFrame as strings.
- **DataFrame.dtypes**
  - A Series object with the labels of the columns and their dtypes.
- **DataFrame.shape**
  - A tuple that describes the dimensionality of the DataFrame.
- **DataFrame.empty**
  - A boolean that is True if and only if there are no values stored in the DataFrame.

Indexable Attributes

- **DataFrame.at[ ]**
  - A specific value in the DataFrame indexed using the row and column labels.
- **DataFrame.iat[ ]**
  - Returns a specific value in the DataFrame indexed using the row and column indexes (like a normal array).
- **DataFrame.loc[ ]**
  - Returns a Series object of the specified row or column if passed in a single label.
  - If passed a list of labels, returns a DataFrame containing the specified rows and columns.

    - Supports slicing, however, unlike base python array slicing, the ‘upper bound’ is inclusive.
- **DataFrame.iloc[ ]**
    - Returns a Series object of the specified row or column if passed in a single row or column index.
    - If passed a list of row or column indices, returns a DataFrame containing the specified rows and columns.
    - Supports slicing, however, unlike base python array slicing, the ‘upper bound’ is inclusive.

Useful Functions and Methods

- **DataFrame.to_numpy()**
    - Returns a ndarray object that contains only the values stored within the
    - If the DataFrame is not of a uniform dtype, then it will try to convert the data to have a single matching dtype since ndarrays can’t contain multiple dtypes.
    - In the worst-case scenario where there are dtypes that are inconvertible, all dtypes will be coerced to be of type ‘object’.
- **DataFrame.astype(dtype, …)**
    - Works similarly to ndarray.astype(), but instead casts all values in the specified DataFrame to the specified dtype.
    - This method may have weird interaction or may raise errors if used on improper or inconvertible dtypes.
- **DataFrame.copy(…)**
    - Returns a complete copy of the labels and values of the given DataFrame.
- **DataFrame.head(n=5)**
    - Returns the first **n** rows of the DataFrame.
    - The default value for n is 5, but it can be any integer value.
- **DataFrame.tail(n=5)**
    - Returns the last **n** rows of the DataFrame.
    - The default value for n is 5, but it can be any integer value.
- **DataFrame.where(cond, …)**
    - Replaces values within the given DataFrame where the passed condition is not met. Very useful for data cleaning.
    - **cond** should be a Series object with bool as its dtype or a DataFrame object.

    - Essentially, the way **cond** decides what values are returned is that, for anywhere that **cond** is true, that value or row is returned.
- **DataFrame.mask(cond,)**
    - ○ Replaces values within the given DataFrame where the passed condition is met.
    - ○ Very useful for data cleaning.
    - ○ **cond** should be a Series object with bool as its dtype or a DataFrame object.
    - ○ Essentially, the way **cond** decides what values are returned is that, for anywhere that **cond** is true, that value or row is returned.

Functions and Methods For I/O

- **pandas.read_pickle(<path>, …)**
    - Returns a DataFrame constructed from the given .pkl file.
    - **path** should be a string of the file location.
- **DataFrame.to_pickle(<path>, …)**
    - Doesn't return any value or object.
    - Instead, it pickles (serializes) the DataFrame and stores it in the specified file using the 'pickle' format.
    - Pickle files are useful for internal use as they are fast and minimal using python's serialization format.
    - **path** should be a string of the file location.
- **pandas.read_csv(<path>, …)**
    - Returns a DataFrame constructed from the given .csv file.
    - **path** should be a string of the file location.
    - There are very many other optional function parameters. Check the API page for more details.
- **DataFrame.to_csv(path_or_buf=None, …)**
    - Serializes the specified DataFrame in a csv format and returns a string with that serialized object.
    - If **path_or_buf** is specified, it will instead return None and store the string in the specified file.
    - Using the comma separated value formatting could be useful for viewing large amounts of data within an excel file or for general external use as it is widely supported by other tools.

- There are very many other optional function parameters. Check the API page for more details.

Matplotlib and Pyplot

API Reference

[matplotlib.pyplot — Matplotlib 3.5.3 documentation]

What are Matplotlib and Pyplot?

Pyplot is a module from the matplotlib package. Matplotlib is an open source python package that is centered around data visualization. This package is very extensive, and its uses are extremely varied. Matplotlib is used extensively in the fields of data science, statistics, and machine learning. It has a massive number of options for customization across all of its modules, as well. This not only allows for very nice looking graphs, but it also allows the creation of much more complex visualizations. It's an all around amazing package for all of your data visualization needs in any field.

Why Pyplot?

Matplotlib has many modules however, we will only be using the pyplot module for our purposes as they are limited in scope and complexity. Pyplot is also what we are most familiar with. As our visualizations are not complex, we only need some basic graphs and pyplot provides us with a simple API to use in that endeavor. The pyplot module only has two dimensional graphs, which is all we will likely need as our primary use for this API is to chart progress in training, accuracies, and other useful statistics or trends. Another useful component of using pyplot is that it will produce images of the graphs that we can store, and it will also open a window when we use it to display those graphs. It's a very convenient and simple module, so it's perfect for this project.

Another reason why we're using pyplot is because it works using the Pandas DataFrame class to great effect. Since we intend to store all of our aggregate data in DataFrames, plotting that data will be extremely straightforward.

Useful Functions and methods

* For each of these functions and methods, there are a very large number of parameters that could be used for this function, so check the API page for more details.

- **pyplot.figure(…)**
    - Returns a Figure object or activates a preexisting one.
- **plt.plot(…)**
    - Plots a line or series of markers of x versus y on the currently active figure.
- **plt.scatter(…)**
    - Creates a scatter plot of points on the currently active figure.
- **plt.show(…)**
    - Displays the current figure.
- **pyplot.subplots(…)**
    - Allows the creation of multiple plots within a single figure object.
- **plt.ticklabel_format(…)**
    - Changes the look of axis tick marks.
- **pyplot.xlabel(…)**
    - Set the x-axis labels to the given values.
- **plt.ylabel(…)**
    - Set the y-axis labels to the given values.

TensorFlow

API Reference

TensorFlow Python API Documentation
TensorFlow Keras Losses Python API Documentation
TensorFlow Keras Optimizers Python API Documentation

Overview

TensorFlow is a free and open-source software library for machine learning and artificial intelligence. Everything required to use TensorFlow is readily available on their GitHub web page. TensorFlow can be used for a variety of things, but specializes in deep structured neural networks. TensorFlow was initially released in November, 2015 and developed by the Google Brain team. Since then, it has seen continued development and appreciation. The library includes a range of pre-built models and datasets, as well as tools for data visualization and model evaluation, making it easier to get started with machine learning projects, as opposed to PyTorch. This is achieved through a number of features, such as AutoDifferentiation, which allows users to compute gradients automatically, and distributed computing, which enables training across multiple devices or machines. TensorFlow also includes a range of APIs and tools for building and deploying machine learning models, including TensorFlow Lite for serving models into phone applications. The library supports eager execution, which allows for more dynamic and intuitive model development, as well as a range of lost/cost functions, accuracy metrics, primitive neural network operations, and optimizers for training neural networks. Overall, TensorFlow is jam packed with useful libraries and features we attempted to utilize fully.

Use Case

TensorFlow is a brilliant framework and library for machine learning. TensorFlow uses an open-source software library called Keras, which is a Python interface for artificial neural networking. Keras is specifically used to implement commonly used features for neural networking. The TensorFlow framework supports both Python and C++ without any strings attached. However, it works decently with other languages such as JavaScript and Java. Additionally, the community surrounding TensorFlow has taken some initiative to integrate it into languages such as Rust, Haskell, and MATLAB. Aside from Languages, TensorFlow also supports a variety of platforms: Linux, macOS, Windows, Android, and even Raspberry Pi. The API allows the user to create advanced

CNN’s without knowing precisely how it works. Almost everything is abstracted to the point of being purely pythonic.

Why TensorFlow

For a generic image classification application, designed specifically for desktop or laptop use, there is an incredible amount of machine learning APIs that can accomplish the job. That being said, as someone who knows nothing about Python and is intrigued to learn it, PyTorch and TensorFlow are the only worthwhile APIs to consider. PyTorch and TensorFlow seem to have been going at a head to head battle, but the end result of which to pick is entirely subjective. PyTorch and TensorFlow are exceptional in almost everything they set out to do. However, this really only applies in a scenario where we are looking at the possibility of developing a desktop application. Such is not the case for our project, we are instead developing a phone application. In this case, TensorFlow is the clear and cut winner. A simple search on any web based search engine will spill out an overwhelming amount of tutorials and articles about using TensorFlow for image classification on mobile devices. The main reason for this is its availability and the recent adaptation of TensorFlow Lite and the easy to use TensorFlow Serving (more on these in a later section).

Hardware

To run an efficient large scale image classification model via deep learning using TensorFlow, the hardware requirements are monumentous. It is recommended to have 4 CPU cores for each individual GPU accelerator. For this reason alone, AMD CPUs are generally the obvious pick for this. AMDs Thread Ripper processors, for example. can contain either 64 or 128 cores per unit, quite impressive. Unfortunately, deep learning is very dependent on the GPU, especially if you want to train a model quickly. This is particularly an issue because to use a GPU for machine learning, it has to be an NVIDIA branded card due to their CUDA toolkit. Which also means, you need an NVIDIA card that supports CUDA. Right now, this is the only safe and effective way to utilize a GPU for machine learning (i.e. via TensorFlow). CUDA is a parallel computing platform and application that allows for quick processing of data, and it is vital to making machine learning practical. Most systems designed for deep learning have multiple NVIDIA GPUs, and of course multiple AMD CPUs, all of which are extremely pricey (pugetsystems.com). Additionally, most of the GPUs present in machine learning

computers are specifically designed for this sort of computation.For example, a popular choice is the A100. The A100 costs over ten thousand dollars!

However, Google, the creators of TensorFlow, also offer a user-managed jupyter notebook instance on their Google Cloud servers known as Google Collab. This would be advantageous to use as it offers NVIDIA cards for the user to utilize for such an event. Additionally, the GPUs are part NVIDIAs tensor core line. This option is what we ended up going with because the advantages are too decent to look past. Not only do you get access to incredible hardware, but you get access to it for a cheap price. In total, only twenty dollars was needed to train and test 50 or so models.

- Hardware Requirements
    - AVX support on the work machines CPU (wikipedia.org)
    - NVIDIA Tensor Core GPU (i.e. A100)

- System Requirements
    - Linux based distribution (64-bit)
    - MacOS 10.12.6 (Sierra) or higher (64-bit)
    - Windows 7 or higher (64-bit)
    - Windows WSL2, Windows 10 19044 or higher (64-bit)

- Software Requirements
    - Python 3.7-3.10
    - Pip, requires PEP 571 and PEP 513 (peps.python.org):
        - Linux: 19.0 or higher
        - MacOS: 20.3 or higher
        - Windows: 19.0 or higher
    - Windows requires Microsoft Visual C++ Redistributable for Visual Studio 2015, 2017, 2019

- NVIDIA Software Requirements (for GPU support)
    - NVIDIA GPU 450.80.02 or higher drivers
    - CUDA Toolkit 11.2
    - cuDNN SDK 8.1.0
    - Optional:
        - Tensor Ray Tracing (TensorRT) abilities

See subheading “Pip Installation” to see how Python and Pip are installed on various operating systems.

Function Calls and Imports

- Importing TensorFlow:
    - The TensorFlow library houses everything we will need for TensorFlow purposes. Other libraries will be used (i.e. NumPy) to improve usability and performance.
    - *import tensorflow as tf*

Processing Images:

Assume the call *import tensorflow as tf* was made.

- **tf_image = tf.io.read_file(<image_path>):**
    - Returns a Tensor that can be used with other TensorFlow image operations. Let’s assume we store the return value as tf_image.

- **tf_image = tf.image.decode_jpeg(tf_image, channels=<int>):**
    - Requires image to be in jpeg format. If it is not, this will not work properly. Required to convert all images to jpg/jpeg format prior to use.
        - Other decode options are available (i.e. png).
    - Increases or reduces color channels of the passed in image.
        - For example, channels = 3 ensures that the image has RGB color channels only.
    - One main way to identify a mushroom is through color, so we need to make sure it has appropriate color channels.
    - While other options are available, they are not particularly useful or applicable in this case.
    - Returns an 8 bit unsigned integer Tensor.

- **tf_image = tf.image.convert_image_dtype(tf_image, tf.float32):**
    - Changing the data type to a floating point value gives a value range between **[0,1)**; otherwise, **[0,MAX]** is used for integers.
    - Other data type options are available, they may or may not be used depending on performance impact. It’s reasonable to believe having the data type as **float32** will stop needless conversions later on.

    - Returns a Tensor that represents the data type specified.

- **tf_image = tf.image.resize(tf_image, size=[<image_height>,<image_width>]):**
    - TensorFlow works best with small square images. Thus, we will want image_height to equal image_width.
    - The larger the image, the heavier the performance cost will be. Most likely we will start off using **64x64** images and then progress to larger images if the losses are great enough to warrant adjustments.
    - Returns a Tensor with the previously specified image height and image width.

Batches

What are batches? Batches are a way to create a collection of training or test samples to iterate on. The amount of samples per batch is based on a hyperparameter, batch size. Batch size will determine the amount of examples to use on one iteration (machinelearningmastery.com). A smaller batch size means that the model will update its parameters more frequently, as each batch of data is processed. This can lead to faster convergence and more frequent updates to the model, which can be beneficial for certain types of problems. However, smaller batches can also increase the amount of noise in the training process, as the model is updating its parameters based on a smaller subset of the data. However, larger batch sizes can be more efficient, as the model can process more data at once; resulting in faster training times. This comes at a steep hardware cost since larger batches also require a lot more memory and computational resources, making them more challenging to work with. In this case we will be creating a batch gradient descent, a collection of training or test samples put into a single batch (radiopaedia.org). The model then processes the entire batch and computes the loss function for that batch. The parameters of the model are updated based on the loss for the entire batch, rather than individual samples. This process is repeated for multiple batches, with the intention of minimizing the overall loss across the entire dataset.

Assume the call *import tensorflow as tf* was made.

- **dataset = tf.data.Dataset.from_tensor_slices(<...>):**
    - Creates a dataset with a separate element for each row of the input Tensor(s).
    - One can specify a single Tensor or multiple Tensors. In certain scenarios it may be wise to do multiple, for example if the dataset has labels. If you specify multiple Tensors they must be placed in between parenthesis, as to signify that they are a tuple of Tensors.

- **dataset = tf.data.Dataset.from_tensors(<>):**
    - Combines values passed in and returns a dataset with a single element.
    - Generally used to make large datasets from several smaller datasets. Thus, the length of the dataset becomes larger.
    - Returns the new compacted dataset.

- **dataset = data_set.map(<image>):**
    - Returns a map object based on the results after applying the function to each item of the parameter. The parameter(s) should be list, tuple, epoch, etc.

- **data_batch = dataset.batch(<max_batch_size>):**
    - max_batch_size should be an integer value.
    - Returns a batch based on the dataset and the batch size hyperparameter.

Keras Model

- **tf.keras.Sequential(<layers>, <...>):**
    - An efficient way to group a linear stack of layers into a Keras Model.
    - Keras Models have many additional functions that can be used later on in the application. Assume that this Keras Model will be defined simply as **model** in the future statements.
    - An example would be to take a model as one layer and your labels as another to make a single return **model**.

- **tf.keras.layers.Dense(<units>, <...>):**
    - Creates a fully dependent layer where each output depends on each input.

    - As such, you can create a model with the shape of the layer that gets passed into Dense. For example, labels can be used to create a single layer so it's compatible with a previously made model.

- **tf.keras.losses.CategoricalCrossentropy():**
    - Computes logarithmic loss between the labels and predictions (wikipedia.org).
    - Helps serve as a basis for how off our models are.
    - Many other loss functions are available and will require significant testing.
        - I.e. RMSE

- **tf.keras.optimizers.Adam():**
    - When compiling a model, you must select an appropriate optimizer so that the "guesses" made are more accurate.
    - Adam uses the Adam algorithm which is an extension to stochastic gradient descent and is generally used for broad scale deep learning applications (tensorflow.org).
    - Adam algorithm works by updating the NN weights based on iterations through the training data.

- **model.compile(<loss>,<optimizer>,<metrics>, <...>):**
    - A way to take a compiled model and configure it with losses, optimizers, and other metrics.
    - The loss parameter allows the user to specify the algorithm to use when calculating loss. A popular loss function is RMSE, but cross-entropy might be more practical for this project.
    - The optimizer parameter allows the user to specify the algorithm to use when attempting optimize guesses.
    - The metrics parameter allows the user to specify how the model should be graded. For example, accuracy via metrics=["accuracy"].

TensorFlow Callbacks

This library allows the user to monitor the state of a model during training and take specific actions based on that state.There are many different types of callbacks available in TensorFlow, each designed to perform a specific function.

- **tf.keras.callbacks.TensorBoard(<>):**
    - Enables the visualization and logging for TensorBoard.

- **tf.keras.callbacks.EarlyStopping(<>):**
    - Create an early stop for when our model stops improving or training.
    - Prevents overfitting.
    - The metrics for when an early stop is called is complicated and dependent on the model you're designing. For our cases: validation loss.
    - **monitor**: declare what should be monitored
    - **mode**:
        - min: when the monitor stops decreasing (we want lowest validation loss)
        - max: when the monitor stops increasing
        - auto: inferred from monitor
    - **patience**: how many epochs to wait to see if performance changes

- **tf.keras.callbacks.ModelCheckpoint(<>):**
    - Allows us to create checkpoints during or after a certain amount of epochs have been completed.
    - The idea is, we can load from this checkpoint if the model runs out of VRAM, RAM, disk space, etc..
    - **filepath:** where the checkpoints should be stored
    - **save_weights_only**: if set to true, only save the weights of the model, and not any other information such as the optimizer state or the epoch number.
    - **verbose:** display when a checkpoint is made and where it's stored.
    - **save_freq:** how often should the model create checkpoints

Other TensorFlow Calls

- **tf.constant(<>):**
    - Creates a tensor with constant properties from a preexisting Tensor, or Tensor like object.

Exporting the Model

Assuming a model has been trained, and a single variable houses the data of that model, you can export the model to a physical disk via a few different ways.

Assume the TensorFlow API has previously been defined inside a Python environment as **tf**.

Definitions:

- TensorFlow module: “Base neural network module class.”
- SavedModel: “A SavedModel is a directory containing serialized signatures and the state needed to run them, including variable values and vocabularies.”

If the trained model is a TensorFlow module object (Keras), and the user wishes to export the data into SavedModel format :

- **tf.saved_model.save(<model>, export_dir=<directory>, options=<signature>)**:
  - model: pass in the variable that houses the data for the trained model.
  - export_dir: string specifying where the model should be saved on local storage.
  - options: pass in a SaveOptions variable.
  - **tf.saved_model.SaveOptions**(
    namespace_whitelist=<whitelist>,
    save_debug_info=<boolean>,
    function_aliases=<aliases>,
    experimental_io_device=<io_device>,
    experimental_variable_policy=<policy>,
    experimental_custom_gradients=<boolean>
    ):

    - **namespace_whitelist:** contains a list of string namespaces to whitelist when saving a model. If no whitelist is provided, all namespaced ops by default will be allowed.

    - **save_debug_info:** boolean value to determine whether any debug information should be saved. If true, a

debug/saved_model_debug_info.pb file will be generated whilst the model is being saved.

- **function_aliases:** store aliases that map from alias name to all concrete function names.

- **experimental_io_device:** string value used to denote where in the hosts file system to access the data. If no string value is provided the filesystem is “accessed from the CPU:0 device of the host where that variable is assigned.”

- **experimental_variable_policy:** either an enumeration or strings. Easiest way to get a variable policy is with tf.saved_model.experimental.VariablePolicy.

- **experimental_custom_gradients:** boolean value that defaults to True. Determines whether or not to use tf.custom_gradient.

Usage of Save

This should be called after the training. In our instance, it is saved to a directory with a manually chosen name. From here, the model can be found with the *.h5 extension. This is an uncompressed version of the model. We ended up using this a lot more than the Lite version because it gave us a really good understanding about how the model performed. Scripts were made to test the performance on the entire dataset, as well as on random images found on the internet.

Test Plan

As a result of the hardware complications using TensorFlow, we decided to do all of our training and testing over Google Colab. Google offers a service where anyone, with a Google account, can create a Jupyter Notebook and run it over their servers. This will allow us to test our python models on good hardware without any extra cost or limitations. With the implementation of TensorFlow callbacks we can stop a model early if it’s performing lower than our expectations. If the model is up to snuff, it will eventually be saved onto Google Drive. From here, we can use the model to evaluate its’

performance across the entire dataset and check what mushrooms are excelling and which are performing poorly.

Quality Management

The test data for the training model will be composed purely from what we retrieve from GBIF via PyGBIF. GBIF is a trusted and verified data source that openly distributes accurate, detailed, and properly labeled data for various organisms all over the globe. However, we will be specifically gathering data on agaricomycetes in the South-East region of the United States of America. Agaricomycetes are a class of fungi that represent what most people think of as mushrooms (cap and stem). All data must and will be properly labeled. Doing so will ensure that our data losses are as accurate as possible. The intention is to train a large model on batches of individual mushrooms. Using the root-mean-squared-deviation (or some other loss function), we can compare the results gathered through our training data on the testing data. Each model we train will require a tuning of epochs and batch sizes. This was a recurring theme, and continued to be so up until the end. The quality of the model is mostly evaluated after training through testing scripts.

Selenium

Overview

Selenium encompasses a ton of tools and libraries that ease the automation process involved in scraping data from a web browser. Selenium testing can be done in a variety of different languages and even through a web browser extension. Selenium is an open-sourced project under the Apache License, initially released in 2004 but has seen continued development.

Use Case

There are a seemingly infinite amount of possibilities using this product. Companies have notoriously used this product to develop scripts that reduce the tedious nature of gathering data from websites. Our use case has the intention of reducing the time spent collecting images to test on. After a long search trying to find a properly compiled mushroom database, our team concluded that there wasn’t one at this point in time. Thus, we are tasked with creating our own mushroom database. Selenium will be used to download images of mushrooms based on input, the input being the mushrooms

scientific name. Given this input, we can download hundreds of images for that mushroom instantaneously (with different search engines). Why do we need so many mushroom images? This allows us to quickly test our model on random images pulled from the internet. This was used to great extent after the model was trained because the images we actually trained on aren't going to show us how well the model performs on new data. Originally, this was intended to be used to get images to train on, but this ended up being unethical and down right naive. Also, the models we made were limited in scope and had to constantly be adjusted. Doing so meant different mushrooms and the need for more test data. The script made with Selenium was really handy to have around.

Requirements and Installation

Certain software will be required to run this: Firefox, Geckodriver, Python and Selenium. Geckodriver requires various versions of software to run properly with Selenium. As a result, noting the versions that will be used is important.
The versions that will be used:

- Mozilla Firefox: 106.0
- Geckodriver: 0.30.0
- Python: 3.10.8
- Selenium: 4.5.0

* Based on requirements listed on Firefox's source documentation.

See subheading "Pip Installation" to see how Python and Pip are installed on various operating systems.

Function Calls and Imports

- **Importing "webdriver":**
    - webdriver will be used to fetch a web browser's driver. In this project, we will use it to open the Gecko driver, required by Firefox.
    - *from selenium import webdriver*

- **Importing "By":**
    - By allows the use of various attributes to assist with locating elements on a webpage. This is particularly useful for very large webpages.
    - Attributes included:
        - ID, NAME, XPATH, LINK_TEXT, PARTIAL_LINK_TEXT, TAG_NAME, CLASS_NAME, CSS_SELECTOR.
    - *from selenium.webdriver.common.by import By*

- **Importing “Keys”:**
    - Keys allows selenium to simulate any key presses. For this project's use case, we will only need to simulate pressing the enter keyboard key. All other text can be sent as a string through a generic selenium import.
    - *from selenium.webdriver.common.keys import Keys*

- **webdriver.Firefox() :**
    - This function call specifies the web browser you wish your script to run off of. I personally use Firefox so I will be referencing Firefox as opposed to Google Chrome; or, any other chromium derivative. Additionally, to use this function the user must have the appropriate driver, in this case the Gecko driver.
    - Only one person needs to gather data, and since I (Jan) will be running the script on Arch Linux “sudo pacman -S geckodriver” will suffice.
    - Alternatively, and most preferable to someone using the Windows operating system, you may specify the path of a downloaded Gecko driver inside of the functions parenthesis; surrounded by single or double quotes.

- **driver.get(“URL”) :**
    - Here driver specifies the returned value from the previously mentioned function: webdriver.Firefox().
    - From this function call we can specify the direct location we want our browser driver to go.
        - For example: browser.get(‘https://google.com/’) will put open the Gecko webdriver and direct it to that specific link.

- **driver.find_element(by=By.ATTRIBUTE, value=’ATTRIBUTE_IDENTIFIER’) :**
    - Here driver specifies the returned value from the previously mentioned function: webdriver.Firefox().
    - “ATTRIBUTE” signifies the attribute to locate via elements loaded on the webpage. The attribute to use is dependent on the structure of the website you are viewing. This is where “ATTRIBUTE_IDENTIFIER” comes in.
    - The programmer behind the website will leave trace elements (i.e. names or text) that we can use to specify the location we mean to obtain.

        - For example, if we wish to download images from “google.com” we could first use browser.get() and then browser.find_element(by=By.LINK_TEXT, value=’Images’).
    - This could in turn allow the user to go directly to https://www.google.com/imghp?hl=en&authuser=0&ogbl.
        - However, notice how this link has an authentication attached to it, by simply using the link it can get messy (not to mention entirely undynamic).

- **driver.find_elements(by=By.ATTRIBUTE, value=’ATTRIBUTE_IDENTIFIER’) :**
    - This is the same as the above function call, except the user is expecting a return of multiple values, rather than a single value.
        - For example, multiple images on the loaded page.

- **element.click() :**
    - Here, “element” refers to the result of driver.find_element(…).
    - Assuming the data we received is that of the image link found on Google’s homepage, we can now simulate an action: in this case, a click.

- **search.send_keys(query, Keys.ENTER) :**
    - Here, “search” refers to the location of the search box, received via: driver.find_element(…).
    - Additionally, “query” refers to some stored text.
        - For example, query = “mushroom logo”.
    - From these variables we can simulate an entire search, of which will display the result on the open browser.
    - send_keys() will pass the values and keyboard interactions specified to the element it’s called from.
    - All together, we would have a selected text field and the text we would like to search for.
    - Keys.ENTER simulates the enter key.
        - For all intents and purposes, this will be the only value we will use in the Keys import.

- **driver.implicitly_wait(time_to_wait) :**
    - Here driver specifies the returned value from the previously mentioned function: webdriver.Firefox().
    - The time_to_wait is used to denote the amount of time one wishes to wait (in seconds) before proceeding.
        - “time_to_wait” should be an integer value.
        - This can be used if a previous action is expected to take a while to load or process and works as expected.

- **driver.execute_script(script, *args) :**
    - Here driver specifies the returned value from the previously mentioned function: webdriver.Firefox().
    - Script is a snippet of JavaScript code and should be wrapped in quotation marks.
    - The user may wish to have a predefined variable here with the script, especially if it’s complex, to avoid needless confusion.
    - *args is any argument required to have the script work properly.

- **element.get_attribute(“attribute name”) :**
    - This function call allows the user to get an attribute from a previously gathered return of driver.find_element(...). For example, we can point to a specific image and then get the hyperlink that defines it:
        - image_link = browser.find_element(by=By.PARTIAL_LINK_TEXT, value=”Images”)
        - image_link.get_attribute(‘href’)

Test Plan and Design Principles

For our use case, no extensive planning is necessary. The script itself will be extremely simple. We wish to open a browser through the web driver, go to a search engine's image search page (if multiple, we will use the most practical option), and begin collecting and downloading images from it. With only a few lines of code, we will be able to save ourselves an incredible amount of time; truly, this is the perfect use case for Python. To test the script, we will first have to verify the driver works as intended: opens a temporary browser page. Afterwards, ensure we are able to download all loaded images from the loaded image search query. Finally, simulate scrolling to continue loading images.

The program should get search query values from a large list, as opposed to manually entering the names of each search. The downloaded images for the search must download into a folder properly labeled by the search. The program should allow for multiple search engines.

While the next part of the test plan isn't directly related to Selenium, it is tied to it. We use the Selenium collected images to test the model on performance. The basic idea of the script is to load the model using tf.keras.models.load_model(model/path), load the images using keras.preprocessing.image_dataset_from_directory(...), and then predict using model.predict(image). There are different ways to measure the accuracy of the result but we chose to use the softmax function, the frontend also implemented this in their Java plugin. Finally, after calculating the result, we can shove all the wrong predictions into a single file that can later be used to analyze.

Quality Management

Unfortunately, it is unrealistic to assume all images we download will be useful. Having a list of scientific names for the mushrooms will definitely help isolate our search to images that are more likely to be relevant and useful for model testing. However, manually testing a search, I have found images results that don't explicitly show the mushroom. These images are entirely useless. To combat this and the horrifying nature of prolonged scrolling on any search engine, we will be forced to limit the number of images we download. Additionally, after all the images are downloaded, we will have to manually delete images that are obviously wrong.

TensorFlow Hub

Overview

TensorFlow Hub is a way for TensorFlow users to freely access various repositories of trained machine learning models. That is it. By using the TensorFlow Hub library, any user may pick and vet a particular repository and use it freely. By simply providing a URL to the repository they wish to use, Keras can define the model layer for later use and conversions into Tensors. However, all trained data is barebones. The user will still need to hypertune the model.

Use Case

TensorFlow Hub can save anyone a lot of time. The most important and time consuming step to machine learning is data collection. Bad data and insignificant counts of data can lead to poor losses and inaccurate results. It's better to reuse a repository if available, and add to it if need be. However, not everything is available in the TensorFlow hub. Which is where the issue of TensorFlow Hub arises.

Plans for the Future

Originally, we planned on uploading our dataset here so that others could use it in future mushroom classification projects. However, the legality of it is questionable since our images come from GBIF. Additionally, our models only use a limited number of mushrooms, because of this I believe it has no purpose being on this website quite yet. If this project ends up being picked up again in the future, it will definitely find its way here along with the entire dataset being uploaded to Kaggle.

TensorFlow Lite

Overview

TensorFlow Lite is a tool provided by TensorFlow to deploy trained models into a condensed format that's ready to be used on a mobile device. To use TensorFlow Lite, the user must first have a trained model known as a Tensor. Once a model has been trained it can be saved in *.tflite format using various TensorFlow Lite packages.

Why TensorFlow Lite

There are several reasons why we chose TensorFlow Lite. One of the main advantages is that it allows users to use trained models on their mobile devices without requiring any additional resources. This can be particularly useful for applications that require real-time processing of large amounts of data, such as image classification. By using TensorFlow Lite, we can quickly deploy new models simply by replacing the old with the new. TensorFlow Lite also offers a number of other benefits, including support for hardware acceleration, which can help improve the performance of models on mobile devices. Additionally, TensorFlow Lite offers a range of tools and libraries that help to optimize machine learning models for mobile devices, including tools for quantization, compression, and performance profiling.

API Calls

Metadata: TensorFlow lite models require a lot of metadata…

Build Info:

- **model_meta = _metadata_db.ModelMetadataT()**
    - **.name, .description, .version, .author**
    - This allows us to add some basic tag information to the model, with the names being self explanatory.

Input:

- **input_meta = _metadata_fb.TensorMetadataT()**
    - **.name**
    - **.description**
    - **.content**
        - **.contentProperties**
            - **.colorSpace**
            - Color space is super important for the model to know. Any ill changes here could result in a confusing disaster.
        - **.contentPropertiesType**

- **input_normalization = _metadata_fb.ProcessUnitT()**
    - **.optionsType**
    - **.options**
        - **.mean, .std**
    - Options should reflect what's expected of the model. The options here must be relayed to front end.

- **input_stats = _metadata_fb.StatsT()**
    - **.max, .min, .stats**

Output:

- **output_meta = _metadata_fb.TensorMetadataT()**
    - **.name, .description, .content, .associatedFiles**
        - **.content.content_properties, .content.contentPropertiesType**

- **output_stats = _metadata_fb.StatsT()**

    - **.max, .min, .stats**

Label:

- **label_file = _metadata_fb.AssociatedFileT()**
    - **os.path.basename(labels/path)**
    - **.description, .type**

Creating the Lite model:

- **converter = tf.lite.TFLiteConverter.from_keras_model(h5/model)**
- **tflite_model = converter.convert()**

There is an incredibly wide range of different ways to save the model and to use the model. It would be ridiculous to include all the way here. Though, different formats will be touched upon in the *Testing and Design Principles* section.

Once the model has been saved it can be evaluated to see how it compares to the original Tensor. To do this you will need to have access to the previously saved model.

- tensor.evalutate_tflite('tensor.tflite', test_data)

Depending on how you choose to save the model, you could end up with swinging performance values compared to the original. For example, either low accuracy with low latency; or, the complete opposite.

Testing and Design Principles

Previously mentioned in the API Calls heading was the amount of variation with how the user plans on saving their model to TensorFlow Lite format. Figure 47 (below) shows the size/latency/precision of the different model architectures which are derived from a paper called EfficientDet (1911.09070). However, the GitHub page explains the point of EfficientDet in easier to access format with graphs, explanations and a tutorial Jupyter notebook. From the README.md, "EfficientDets are a family of object detection models, which achieve state-of-the-art 55.1mAP on COCO test-dev, yet being 4x - 9x smaller and using 13x - 42x fewer FLOPs than previous detectors. Our models also run 2x - 4x faster on GPU, and 5x - 11x faster on CPU than other detectors." Not only is this an impressive performance boost all around, but the performance boost on the CPU is extremely important to note. Since the image classifier will be used on mobile devices,

without any form of GPU to utilize, it’s important to create models that are efficient on CPUs.

Model architecture	Size(MB)*	Latency(ms)**	Average Precision***
EfficientDet-Lite0	4.4	37	25.69%
EfficientDet-Lite1	5.8	49	30.55%
EfficientDet-Lite2	7.2	69	33.97%
EfficientDet-Lite3	11.4	116	37.70%
EfficientDet-Lite4	19.9	260	41.96%

Figure 47: Choose a model architecture 1

Later on we found it to be more effective to test the model on the actual compiled app or through phone emulation. When working on a model to recognize different types of mushrooms, it is important to test it in different scenarios to ensure that it is accurate and effective. While testing the model on a computer may give some insights into its performance, it is not always the most effective way to evaluate the model's performance. One approach that has proven to be effective is to test the model on the actual compiled app or through phone emulation. This approach allows us to test the model's performance in real-world scenarios, where users are likely to encounter it. It also allows us to see how the model performs in different lighting conditions and in different angles, which can help identify any potential issues or areas for improvement.

Jupyter

Jupyter is an open source project that originated from another project called IPython, which was attempting to make python more interactive and easier to ‘play around’ with. The Jupyter team took this idea and decided that many, in fact all, languages would benefit from this way of writing code. So, they set out to make a new medium that effectively was a halfway point between a paper and source code.

This type of programming ‘notebook’ serves as a simple, yet versatile and useful way of prototyping essentially any kind of procedural program. While it does work with parallel

programming, this form of code does not lend itself very well to taking advantage of it. All of these notebooks have two simple and repeatable options: code blocks, and text blocks. The code blocks allow just that; code written as though it were in a standard program or script. The text blocks actually use Markdown to format the text and thus allows for formatting similar to many other text editors.

One of the best uses for these notebooks is documented prototyping. You have the ability to write full paragraphs throughout the code without dealing with massive comments or strings of text in the code. Not to mention, you can use hyperlinks and more advanced formatting than those comments or strings could provide. One thing to note as well is that the code blocks are split up and separately runnable. This means that you can treat each individual code block as its own function and run it repeatedly for iterative results within the program. This does have some interesting interactions and quirks, but I will not describe them here.

So, for the purposes of prototyping, we decided to use this software because it allows us to have much more interactivity compared to your more standard program. Many modern research projects for machine learning, data science, and plenty of other fields use Jupyter Notebooks to illustrate their research in a way that allows a viewer to read the code as if it were a book, yet still run it like code. It’s perfect for a self-contained exploration of a topic.

Google CoLaboratory (Google CoLab)

Google has recently created a super simple to use hosting service for Jupyter notebooks. Dubbed Google CoLaboratory, this new software is a completely free and pre-configured online integrated development environment (IDE). Anyone with a free google account can have access to it. CoLab is a huge benefit to python programmers everywhere because you can also connect it to your Google Drive and access your files on the cloud. It’s free, easy to use, and it even allows you access to some GPUs. However, we opted to pay for computation units to get access to higher end GPUs with a considerable boost in machine learning performance and VRAM. It is important to note that CoLab only works with Python. It does not currently support any other type of Jupyter notebook, so creating a model using C++ would not be practical.

It is also useful to note that, because it is using Google Drive, it is also completely collaborative (hence the name CoLab), so we are able to share and interact with these notebooks simultaneously. Additionally, only one person needs to house the scripts and

data. So long as everyone has access to the links, anyone can train a model and adjust parameters.

Pip Installation

The installation process will be described briefly. Since BSD operating systems aren't very popular they will not be included. However, Apple's MacOS, Windows, and various Linux package managers will be included.

Apple

Homebrew isn't installed by default on some Macintosh devices. To install it, visit https://brew.sh and follow their installation procedure. After verifying the Homebrew installation the user may now install python:

```
$ brew update && brew upgrade
$ brew install python3
```

Linux

Advanced Package Tool:

```
$ sudo apt update && sudo apt upgrade
$ sudo apt install python3.6 python3-pip
```

Dandified YUM:

```
$ sudo dnf install python3
$ sudo yum install python3-devel
$ sudo yum groupinstall 'development tools'
```

Pacman:

```
$ sudo pacman -Sy python3 python-pip
```

Windows

To install python on a Windows operating systems, visit https://www.python.org/downloads/windows/ and download the latest stable release of python3. Run the installer and make sure to add Python 3.# to your path. This can be done from the installer, or manually.

To verify the installation of both python and pip, a few simple commands can be used.

```
>: python --version
>: pip -v
```

Once both python and pip have been verified you can use python to execute programs and pip to manage (install/remove/list/etc) packages.

```
$ pip install [package_name1] [package_name2] [...]
```

Continuous Integration Specification

GitHub

GitHub is a software development hosting service that allows for many features important for software development. Most importantly, GitHub implements and supports Git's version control functionality, allowing for multiple users to work on distributed instances of the same codebase, and streamline code integration to a single source, hosted online for shared viewing. Codebases are owned by users or organizations, and represented as repositories under their ownership.

As the largest source code host in the world, our team decided it best we host our codebase on GitHub since all members were familiar with GitHub. Becoming more familiar with GitHub would allow the team to familiarize themselves with the tools provided, allowing for growth in personal skills in preparation for software development in the tech industry.

Repository

GitHub uses repositories to store and separate code bases in ways that are reasonably consistent with the manner in which projects are separated by developers. They emulate file structures and help in separating concerns between apps from one user to another.

Polyamanita will host two major repositories to separate concerns between the front and back end sides of the application. This will allow workflows and merges from different sectors of developers to be kept completely separate, avoiding duplicate workflow triggers and easier compilation.

Continuous Deployment Specification

GitHub Actions

GitHub Actions are a CI/CD platform provided by GitHub that allows for automatic pipeline triggering during different events on GitHub. Workflows are configured to define a set number of jobs that get triggered during different events, like pull requests or commits. These jobs can be run in sequence and use previous output from previous jobs to determine functionality for their current job. Each job contains a sequential number of steps to run, which can include running custom actions or public actions defined by GitHub or otherwise.

Technologies

Amazon Elastic Beanstalk

Elastic Beanstalk (EB) is a cloud service provided by Amazon that builds and deploys app images to a running Elastic Compute instance.

Buildpacks

Buildpacks are libraries of various language repositories that support the automated build and creation of applications into containers. Each buildpack performs specific operations on the application to support in the building of it, and multiple buildpacks can be strung together to fully flesh out a build app container.

Polyamanita will use buildpacks through the Pack CLI to build both the front and back end facets of the app, for a streamlined deployment to Amazon S3 and Elastic Beanstalk. Leveraging GitHub Actions, the Pack CLI can be interfaced with and artifact builds can be created and pushed to further workflow steps for deployment.

Pack

Pack is a CLI used to interface with buildpacks and build application source code. It's used secondarily to rebase app images and create additional components used for more complex app building.

Builders

Builders orchestrate the systematic approach to creating the app through its image, including all buildpacks that will be used to build the app, configuration for each buildpack, and additional metadata to help in the execution of the app build.

Paketo

Paketo Buildpacks are a set of highly modularized buildpacks that provide a high control approach to app building, through the use of robust configuration options from every buildpack. Polyamanita will be using Paketo Buildpacks to custom build both the front and back end facets of the app.

Workflows

Polyamanita uses GitHub Actions for the purpose of creating workflows to help automate and verify our code integrity during the development process, for both the front and back-end facets of the software. Each facet of Polyamanita's code structure, limited to the front end, API layer, and back end, will be accompanied with several workflows to help with code validation and deployment. Detailed below are the workflows that will be created for each facet, including the jobs each workflow will take and the steps each job will undergo to perform its intended process.

Workflow: **Front End - Test Pull Request**

Front End - Test Pull Request validates code correctness for any pull requests on the Front End repository. Code correctness will be verified by running all unit test suites on the front end. This workflow will be run upon every pull request made from a feature branch of the Polyamanita Front End repository to the main branch of the repository.

If any job in the workflow fails, then the pull request will be blocked from merging until changes are made to the feature branch to pass the tests.

Job: **Unit Tests**

The **Unit Tests** job sets up a test environment for the Front end repository and runs all the unit test suites associated in the repository.

Step 1: Setup Job

- Run on Operating System: Ubuntu LTS
- Run on image: **node:latest**
- Retrieve GITHUB_TOKEN
- Download action repository: **actions/checkout@v3**

Step 2: Checkout

- Run **actions/checkout@v3**

Step 3: Build npm node modules

- Run command: `npm install`

Step 4: Run unit tests

- Run command: `npm run test`

Workflow: **Front End - Lint**

Front End - Lint ensures that the front end codebase maintains cleanliness in structure and syntax, while also catching potential bugs in code.
The workflow will be run upon any commit made to a feature branch, as well as any pull requests made to any branch.
If any job in the workflow fails during a pull request, then the pull request will be blocked from merging until changes are made to the branch to pass the tests.

Job: Lint

The **Lint** job contains all lint checks for the front end codebase.

Step 1: Setup Job

- Run on Operating System: Ubuntu LTS
- Retrieve GITHUB_TOKEN
- Download action repository: **actions/checkout@v3**
- Download action repository: **actions/setup-node@v3**

Step 2: Checkout

- Run **actions/checkout@v3**

Step 3: Setup Yarn

- Run **actions/setup-node@v3**
- Run command: `yarn install`

Step 4: Run ESLint

- Run command: `eslint . –ext .js,.jsx,.ts,.tsx`

Workflow: **API - Test Pull Request**

API - Test Pull Request ensures code correctness for any pull requests for the API. Code correctness will be verified through running all unit test suites for the API, followed by integration tests that will run the server locally with test requests to every endpoint. This workflow will be run upon every pull request made from a feature branch of the Polyamanita server repository to the main branch of the repository.

If any job in the workflow fails, then the pull request will be blocked from merging until changes are made to the feature branch to pass the tests.

Job: **Unit Tests**

The **Unit Tests** job sets up a test environment for the API codebase and runs all the unit test suites associated in the codebase .

Step 1: Setup Job

- Run on Operating System: Ubuntu LTS
- Retrieve GITHUB_TOKEN
- Download action repository: **actions/setup-go@v3**
- Download action repository: **actions/checkout@v3**

Step 2: Setup Go

- Run **actions/setup-go@v3**
- Run command: `go version`
- Run command: `go env`

Step 3: Checkout

- Run **actions/checkout@v3**

Step 4: Run unit tests

- Run command: `go test`

Job: **Integration Tests**

The **Integration Tests** job sets up a test environment for the API codebase, runs the API server locally, and then runs the integration test script to make cumulative calls to the local server and verify systemic correctness.

Step 1: Setup Job

- Run on Operating System: Ubuntu LTS
- Retrieve GITHUB_TOKEN
- Download action repository: **actions/setup-go@v3**
- Download action repository: **actions/checkout@v3**

Step 2: Setup Go

- Run **actions/setup-go@v3**
- Run command: go version
- Run command: go env

Step 3: Checkout

- Run **actions/checkout@v3**

Step 4: Run integration tests

- Run command: `go run tests/integration_test.go`

Workflow: **API - Lint**

API - Lint ensures that the codebase maintains cleanliness in structure and syntax, while also catching potential bugs in code.

The workflow will be run upon any commit made to a feature branch, as well as any pull requests made to any branch.
If any job in the workflow fails during a pull request, then the pull request will be blocked from merging until changes are made to the branch to pass the tests.

Job: Lint

The **Lint** job contains all lint checks for the API codebase.

Step 1: Setup Job

- Run on Operating System: Ubuntu LTS
- Retrieve GITHUB_TOKEN
- Download action repository: actions/setup-go@v3
- Download action repository: actions/checkout@v3
- Download action repository: golangci/golangci-lint-action@v3

Step 2: Setup Go

- Run actions/setup-go@v3
- Run command: go version
- Run command: go env

Step 3: Checkout

- Run **actions/checkout@v3**

Step 4: Run Golangci-lint

- Run **golangci/golangci-lint-action@v3**

Workflow: **API - Deploy to Dev**

API - Deploy to Dev checks out the latest commit on the main branch, sets up authentication with Amazon Web Services and docker, then builds the server image and pushes it up to Amazon S3 so that it can be run in a dev environment through Elastic Beanstalk.

The workflow uses the environment SERVICE_NAME: `polyamanita-server-dev`

If any job in the workflow fails during execution, then the associated commit for the server image will be blocked from being built or deployed into staging.

Job: **Build**

The **Build** job sets up authentication with AWS and docker, builds the image, and then uploads it to Amazon S3.

Step 1: Setup Job

- Run on Operating System: Ubuntu LTS
- Retrieve GITHUB_TOKEN, GITHUB_SHA, AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, DOCKER_TOKEN,
- Declare env BUILDER, SERVICE_NAME, PROJECT ID, REGISTRY
- Download action repository: **actions/checkout@v3**
- Download action repository: **docker/login-action@v1**
- Download action repository: **amazon-github-actions/auth@v0.4.0**
- Download action repository: **amazon-github-actions/setup-aws@v0.2.1**
- Download action repository: **dfreilich/pack-action@v2.1.1**

Step 2: Checkout

- Run **actions/checkout@v3**

Step 3: Setup Authentication

- Run **docker/login-action@v1**
    - With: DOCKER_TOKEN
- Run **amazon-github-actions/auth@v0.4.0**
    - With: AWS_SECRET_ACCESS_KEY
- Run **amazon-github-actions/setup-aws@v0.2.1**
- Run **amazonauth configure-docker**

Step 4: Build Image

- Run **dfreilich/pack-action@v2.1.1**
    - With: SERVICE_NAME, BUILDER

Step 5: Tag Image

- Run `docker tag $SERVICE_NAME $REGISTRY/$PROJECT_ID/$SERVICE_NAME:$GITHUB_SHA`

Step 6: Push to S3

- Run `docker push $REGISTRY/$PROJECT_ID/$SERVICE_NAME:$GITHUB_SHA`

- Run `aws s3 cp ./$GITHUB_SHA.zip s3://${{env.AWS_S3_ARTIFACTS_BUCKET}}/$GITHUB_SHA`

```
        aws elasticbeanstalk create-application-version \
          --application-name ${{env.AWS_EB_APPLICATION_NAME}} \
          --version-label $GITHUB_SHA \
          --source-bundle
S3Bucket=${{env.AWS_S3_ARTIFACTS_BUCKET}},S3Key=$GITHUB_SHA \
          --auto-create-application \
          --tags Key=COMMIT_SHA,Value=$GITHUB_SHA \
```

Job: **Deploy**

The **Deploy** job sets up authentication with Amazon Web Services and deploys it with Elastic Beanstalk on the development environment.

Step 1: Setup Job

- Run on Operating System: Ubuntu LTS
- Retrieve GITHUB_TOKEN, GITHUB_SHA, AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY,
- Declare env BUILDER, SERVICE_NAME, PROJECT ID, REGISTRY
- Download action repository: **amazon-github-actions/auth@v0.4.0**
- Download action repository: **amazon-github-actions/setup-aws@v0.2.1**

Step 2: Setup Authentication

- Run **amazon-github-actions/auth@v0.4.0**
    - With: AWS_SECRET_ACCESS_KEY
- Run **amazon-github-actions/setup-aws@v0.2.1**

Step 3: Deploy Image

- Run `aws elasticbeanstalk update-environment \`

```
        --application-name ${AWS_EB_APPLICATION_NAME} \
        --environment-name ${AWS_EB_APPLICATION_NAME}-dev \
        --version-label $GITHUB_SHA \
```

Workflow: **API - Deploy to Staging**

API - Deploy to Staging Retrieves the application image that was built during **API - Deploy to Dev** and deploys it onto the staging environment. After successful deployment, a Go script is run to generate and send fake calls to all endpoints for 10 minutes, to simulate higher loads and verify app correctness.

The workflow uses the environment SERVICE_NAME: polyamanita-server-staging

If any job in the workflow fails during execution, then the associated commit for the server image will be blocked from being used for deployment into production

Job: **Deploy**

The **Deploy** job sets up authentication with AWS and docker, builds the image, and then uploads and deploys it with Amazon S3 and Elastic Beanstalk.

Step 1: Setup Job

- Run on Operating System: Ubuntu LTS
- Retrieve GITHUB_TOKEN, GITHUB_SHA, AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY,
- Declare env BUILDER, SERVICE_NAME, PROJECT ID, REGISTRY
- Download action repository: **amazon-github-actions/auth@v0.4.0**
- Download action repository: **amazon-github-actions/setup-aws@v0.2.1**

Step 2: Setup Authentication

- Run **amazon-github-actions/auth@v0.4.0**
    - With: AWS_SECRET_ACCESS_KEY
- Run **amazon-github-actions/setup-aws@v0.2.1**

Step 3: Deploy Image

- Run
```
aws elasticbeanstalk update-environment \
        --application-name ${AWS_EB_APPLICATION_NAME} \
        --environment-name ${AWS_EB_APPLICATION_NAME}-staging \
        --version-label $GITHUB_SHA \
```

Job: **Run Mock Calls**

The **Run Mock Calls** job runs a mock call script to verify server correctness and high read/write load. The Job will run for approximately 15 minutes.

Step 1: Setup Job

- Run on Operating System: Ubuntu LTS
- Retrieve GITHUB_TOKEN
- Download action repository: **actions/checkout@v3**
- Download action repository: **actions/setup-go@v3**

Step 2: Checkout

- Run **actions/checkout@v3**

Step 3: Setup Go

- Run **actions/setup-go@v3**
- Run command: `go version`
- Run command: `go env`

Step 4: Run test script

- Run command: `go run tests/mock_calls.go`
    - With: `SERVICE_NAME`

Workflow: API - Deploy to Production

API - Deploy to Production checks out the latest commit on the main branch, sets up authentication with Amazon Web Services and Docker, then builds the server image and pushes it up to Amazon S3 so that it can be run in a production environment through Elastic Beanstalk

The workflow uses the environment SERVICE_NAME: `polyamanita-server`

Job: **Deploy**

The **Deploy** job sets up authentication with AWS and docker, builds the image, and then uploads and deploys it with Amazon S3 and Elastic Beanstalk.

Step 1: Deploy Image

- Run `aws elasticbeanstalk update-environment \`

```
--application-name ${AWS_EB_APPLICATION_NAME} \
--environment-name ${AWS_EB_APPLICATION_NAME} \
--version-label $GITHUB_SHA \
```

Lessons Learned, Insight Gained

Throughout this project, we spent a lot of time learning about and evaluating what it means to make not just a functional product, but a good one. Together as a group, we had many interesting struggles and, as individuals, we had our own unique difficulties that had to be overcome in the end. The difficulties of designing, planning, building the infrastructure for, implementing, testing, and polishing any application are most definitely not unique to ours, but the experience gained from overcoming them are ours alone.

From the beginning, we knew that we wanted to make this application work the way it does, but starting with the end goal in mind meant that we had to determine a path of software and connections that could lead us there, which is much easier said than done for less experienced programmers, but with many hours worth of discussions in both planning and *re*planning components, systems, and features, we did manage to reach our end goal.

What goes into a good mobile application

The heading of this section poses a question that any mobile developer can tell you is very difficult to pin down and relies on too many contextual details to answer generally. For us, however, what goes into a good application is (hopefully) something we have been able to answer for ourselves in *this* app. We tried to create an application that users would be able to use to interact with the world around them in a way that allowed them to both explore and learn something along the way. Other applications have their own purposes and goals, but any feature we added in furtherance of our goal is one that answers the question of “what goes into a good mobile application”.

Our UX is clean and easy to read as our front end members wanted users to enjoy and appreciate the look and feel of our app. Our Database and APIs are well designed and are able to work in large batches because we wanted users to be able to use the app without worrying much about internet connections and just go out into the world. And our machine learning model is scalable, reusable, and robust because we wanted to

help the user learn something when they use our app and maybe learn a lot more in the future.

We spent almost 8 months working on this and the struggles and niche issues we encountered are innumerable, but important as the time we spent solving them is what led us to answering that question.

For any other mobile developers reading this (firstly, why?), know that your answer will likely be very different, but setting a goal for your team and application and making decisions that will help your users appreciate that goal is what will lead you to the answers you seek.

What goes into a good machine learning application

The question of this section is very distinct from the last as our application is not just a mobile app that only has mobile app problems. Our application integrates machine learning, and with that comes many other issues that are complex and nuanced in their own right. It also tends to be a lot more technical than making a good mobile app.

One of the most important features of any good mobile application is whether or not the model can actually do what you say it does. Thankfully, we can say that our model *does* (mostly) correctly identify different species of mushrooms. However, the process that led us to a functional model was very long, tedious, and took many hours of research, experimenting with different choices and settings, and *so* much data collection.

A factor that tends to be hidden to the users that plays into a good ML application is the actual data backing the model. Data is a massive component of machine learning, and it often is overlooked by those who want to utilize modern ML techniques. We learned the hard way how important having reliable, clean data is and it should be something that is treated as the primary basis for any ML app. Without enough good data, you *don't* have a model. You just have a heap of layers that could *become* a model. The rule across all computer science is 'Garbage in, garbage out', and this might be even more true of machine learning.

If you have good data, and you have a good model structure, then you also need access to good hardware, or a *lot* of time to spend waiting on training to complete. The more you want your model to do, the more complex it has to be, the more data you need, the more time you spend training. These are all large time sinks and all need to

be considered when attempting to make any ML model. And as a word of advice, be sure to make it clear what your model actually *can* do. If you tell someone that your model can identify 100 different things, make sure that people are aware of the things it *can't* identify. This is more of an ethical concern than anything, but if your users try to use your model to do things it can't do, then they could potentially cause harm to themselves or others. Using this technology brings with it a lot of important concerns that must be considered. In our app, for example, we are very well aware of the chance that a user comes across a poisonous mushroom, and that's why it is very important to us that users are aware that our application is meant for education, not for use in guaranteeing safety.

The data, model structure, ability, and handling ethical concerns are what lead to good and effective applications that use ML integration. They should all be carefully considered and care should be taken when creating any app that uses machine learning, as it is a very powerful tool, but it is often taken as gospel or not treated like what it is: a fallible tool made by fallible humans.

What we should have done

Despite all the pretty words from the last two sections, this app isn't perfect, and there are definitely things we could have and should have done in hindsight.

We should have been more proactive in utilizing UCF's other department's resources such as the members of the Biology department, for example. If we had reached out to some of them, we might have saved many hours looking for images online as opposed to asking for resources and being directed to the websites we ended up using. Doing so also could have given us more direction early on to targeting our audience better, or even to help us consider other applications such as for purposes within academia as opposed to for the 'everyman'.

We also should have taken the time (weather permitting) to test our application more thoroughly in the field. The little testing we were able to do on live species gave a lot of insight that would have been very useful, had we given ourselves more time to act on it and update our application and retrain the model accordingly. It is one thing to test that all of the individual components of your application work individually and work together to do what you *expect* them to, but it is another thing to make sure that they work how you *want* them to.

The largest difficulty with this application is simply that to do what we truly set out to do in terms of allowing users to explore *anywhere* is that we can't tell them about every species of mushroom out there. We can't even get close, honestly. There are just too many species that exist. The only way we could have gotten closer to our goal (by our methods) is by having a larger dataset with more species, more images, better images, etc.. This wasn't feasible for us, but it was something that, if we had more time, we could have improved on at least a little. We are proud that our model can identify as many species as it can, but we are sad that it can *only* identify so many. This problem is a common one and a huge limiting factor to many ML-integrated applications, not just our own, and it is one that will likely undercut many other developers' work.

Milestones

Due Date	Milestones
2022-08-31	Set up Jira project, GitHub organization, and Discord server
2022-10-01	Assignment #3 Requirement (Milestones, Executive Summary)
2022-10-05	TA Check-In #1
2022-10-14	[ML] Webscraper created (Selenium)
2022-10-15	Half Of Design Doc written (15 pages per member)
2022-10-21	Set up AWS project
2022-10-26	[ML] Webscraper created (GBIF)
2022-10-28	[ML] Preliminary image search run and results evaluated
2022-10-30	[ML] Final dataset images collected
2022-11-07	[ML] Prototyped a simple CNN in a Python notebook
2022-11-22	[ML] Benchmark on Python notebooks run and analyzed
2022-12-05	[ML/DB] Database filled and organized using image dataset
2022-12-05	Submit document before 23:59.
2022-12-15	[FE] Mobile app navigation ready.
2023-01-09	Create running skeleton on Front-end / Back end

2023-01-12	[CI/CD] Set up CI/CD Workflow Skeletons for front and back-end
2023-01-20	[CI/CD] Automatic deployment of api server
2023-01-20	[ML] CNN model implemented
2023-01-22	[API] Implement user CRUD operation endpoints
2023-01-22	[FE] Ability to take pictures and save.
2023-01-25	Model trained and tested on a small subset and results analyzed
2023-01-28	[API] Implement user authentication endpoints
2023-01-31	[FE] Integrate UI to identify mushrooms.
2023-01-31	[FE] Snap Feature Complete
2023-01-31	[ML] Adjustments made to program and model
2023-02-04	[API] Implement user journal endpoints
2023-02-14	[ML] Initial full-scale training and testing run completed (Single Epoch)
2023-02-14	[API] Implement image/data sync endpoints
2023-02-21	[ML] Results analyzed and adjustments made
2023-02-23	[FE] Map Feature integrated
2023-02-23	[FE] Account Integration, User Registration
2023-03-01	[ML] Continued training and evaluation of results
2023-03-01	[FE] - Photo Sync
2023-03-01	[FE] - Account Sync
2023-03-05	[ML] Manual testing (Sanity Check)
2023-03-07	[ML] Export model to project application (mobile)
2023-03-10	[ML] Integration and real world testing (on app)
2023-04-01	Complete project implementation
2023-04-01	Feedback across users
2023-04-19	Final Presentation

2023-04-28	dfghdfgfghjPadffghfriuP-pp-ppp-party!!!!!!

Organization

Communication (Discord / Meetings)

Communication across the time of writing this document and implementation will be used with the help of Discord. Discord allows not only text and voice communications, but allows organization of different topics to be discussed in unique channels for different topics. The team has used this extensively by separating different concerns into different channels that way any relevance can be quickly searched, indexed, and read while looking at the channel. For example, the group's Discord has a unique channel for references; this is where all important links have been going regarding group project tools such as Figma and Jira.

Division of Labor

Labor will be divided into three different teams: Front End, API / Backend, and the ML team. With the six members on the team, two members have been assigned each role in that team. Each role will discuss with one another about design and implementation decisions while prototyping and integrating the application. This pair programming dynamic becomes powerful when solving a problem. This dynamic also works well within team meetings, as each team discusses what has been completed for the week.

The division of labor:

Front End : Tyler Clarke, Jan Darge (post ML setup), Layne Hoelscher
API / BE : Amelia Castilla, Nawras Rawas Qalaji
ML / DC : Jan Darge, Ethan Wollet

Appendices

Copyright permissions & License Agreements

GBIF:

> "CC0, under which data are made available for any use without restriction or particular requirements on the part of users"
>
> "CC BY, under which data are made available for any use provided that attribution is appropriately given for the sources of data used, in the manner specified by the owner"
>
> "CC BY-NC, under which data are made available for any use provided that attribution is appropriately given and provided the use is not for commercial purposes"

Jupyter:

> Copyright © 2022 Project Jupyter Contributors. All rights reserved.

Matplotlib:

> Copyright © 2002–2012 John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team; 2012–2022 The Matplotlib development team.

Numpy:

> Copyright © 2005, NumPy Developers. All rights reserved.

Pandas:

> Copyright © 2008-2011, AQR Capital Management, LLC, Lambda Foundry, Inc. and PyData Development Team. All rights reserved.

Pip:

> Copyright © 2008-present, The pip developers. MIT License.

PyGBIF:

> Copyright © 2019, Scott Chamberlain. MIT License.

Selenium:

> Apache 2.0 License.

TensorFlow:

> Apache 2.0 License.

TensorFlow Hub:

> Apache 2.0 License.

TensorFlow Lite:

> Apache 2.0 License.

Software

Global Biodiversity Information Facility (GBIF)

Google CoLaboratory (Jupyter Notebook)

SwaggerHub. Accessed December 1, 2022. https://app.swaggerhub.com/home.

Sources

GBIF

The download citation for the dataset used:

GBIF.org (21 November 2022) GBIF Occurrence Download
https://doi.org/10.15468/dl.pjkxtn

In Text

Material used to help write documentation.

Almeida, Pedro. “The UX of Pokémon Go : A Case Study.” Medium. Medium, November 17, 2016. https://medium.com/@pedro_ux/pok%C3%A9mon-go-a-case-for-ux-and-psychology-8b6377db573a.

Atlassian Design System. Accessed December 5, 2022. https://atlassian.design/.

Brownlee, Jason. “Difference between a Batch and an Epoch in a Neural Network.” MachineLearningMastery.com, August 15, 2022. https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/.

Dua, Dheeru, and Casey Graff. “Mushroom Data Set.” Irvine, CA, 2019. https://archive.ics.uci.edu/ml/datasets/mushroom.

Eyal, Nir. “User Investment: Make Your Users Do the Work.” Nir and Far, August 25, 2022. https://www.nirandfar.com/makeyourusersdothework/.

Eyal, Nir. “The Hooked Model: How to Manufacture Desire in 4 Steps.” Nir and Far, November 23, 2022. https://www.nirandfar.com/how-to-manufacture-desire/.

Franck, Alan, and Mark Hafen. “USF Species Project: Florida Fungi.” Florida Fungi. Tampa, FL, 2013. http://arborist.forest.usf.edu/floridafungi.

Google. “Automl/Efficientdet at Master · Google/Automl.” GitHub. Accessed December 3, 2022. https://github.com/google/automl/tree/master/efficientdet.

Mortensen, Ditte Hvas. “User Research: What It Is and Why You Should Do It.” The Interaction Design Foundation. Interaction Design Foundation, December 3, 2022. https://www.interaction-design.org/literature/article/user-research-what-it-is-and-why-you-should-do-it.

Murphy, Andrew. “Batch Size (Machine Learning): Radiology Reference Article.” Radiopaedia Blog RSS. Radiopaedia.org, May 2, 2019. https://radiopaedia.org/articles/batch-size-machine-learning?lang=us.

Sharma, Naiya. “A Look at React Native and React.js.” DZone. Dzone.com. January 17, 2020. https://dzone.com/articles/whats-the-difference-between-react-native-and-rea c.

Tan, Mingxing, Ruoming Pang, and Quoc V. Le. “EfficientDet: Scalable and Efficient Object Detection.” arXiv.org, July 27, 2020. https://arxiv.org/abs/1911.09070.

Yalcin, Hulya, and Salar Razavi. “Plant Classification Using Convolutional Neural Networks | IEEE ...” IEEE Xplore. IEEE. Accessed December 3, 2022. https://ieeexplore.ieee.org/document/7577698.

“Advanced Vector Extensions.” Wikipedia. Wikimedia Foundation, November 24, 2022. https://en.wikipedia.org/wiki/Advanced_Vector_Extensions.

“Build and Deploy a Custom Object Detection Model with Tensorflow Lite (Android) | Google Developers.” Google. Google. Accessed December 3, 2022. https://developers.google.com/codelabs/tflite-object-detection-android.

“Cross Entropy.” Wikipedia. Wikimedia Foundation, November 10, 2022. https://en.wikipedia.org/wiki/Cross_entropy.

“Creating a Style Guide: One-Stop Place for Your UI Design Team.” AltexSoft. AltexSoft, September 18, 2019.

https://www.altexsoft.com/blog/uxdesign/creating-a-style-guide-one-stop-place-for-your-ui-design-team/.

“Google Colaboratory.” Google Colab. TensorFlow. Accessed December 3, 2022. https://colab.research.google.com/github/googlecodelabs/odml-pathways/blob/main/object-detection/codelab2/python/Train_a_salad_detector_with_TFLite_Model_Maker.ipynb?hl=zh-cn.

“Hardware Recommendations for Machine Learning / AI.” Puget Systems, December 2, 2022. https://www.pugetsystems.com/solutions/scientific-computing-workstations/machine-learning-ai/hardware-recommendations/.

“Install Tensorflow with Pip.” TensorFlow. Accessed December 3, 2022. https://www.tensorflow.org/install/pip.

“Maslow's Hierarchy of Needs.” Wikipedia. Wikimedia Foundation, November 25, 2022. https://en.wikipedia.org/wiki/Maslow%27s_hierarchy_of_needs#Transcendence_needs.

“Module: TF : Tensorflow V2.11.0.” TensorFlow. Accessed December 3, 2022. https://www.tensorflow.org/api_docs/python/tf.

“Module: Tf.keras.losses : Tensorflow V2.11.0.” TensorFlow. Accessed December 3, 2022. https://www.tensorflow.org/api_docs/python/tf/keras/losses.

“Module: Tf.keras.optimizers : Tensorflow V2.11.0.” TensorFlow. Accessed December 3, 2022. https://www.tensorflow.org/api_docs/python/tf/keras/optimizers.

“Module: tflite_support.task : Tensorflow V2.11.0.” TensorFlow. Accessed December 5, 2022. https://www.tensorflow.org/lite/api_docs/python/tflite_support/task

“Python Enhancement Proposals.” PEP 571 – The manylinux2010 Platform Tag. Accessed December 3, 2022. https://peps.python.org/pep-0571/.

“Quickstart for Android : Tensorflow Lite.” TensorFlow. Accessed December 3, 2022. https://www.tensorflow.org/lite/android/quickstart.

"Supported Platforms." Supported platforms - Firefox Source Docs documentation. Accessed December 3, 2022. https://firefox-source-docs.mozilla.org/testing/geckodriver/Support.html.

"Technical Documentation in Software Development: Types, Best Practices, and Tools." AltexSoft. AltexSoft, December 9, 2019. https://www.altexsoft.com/blog/business/technical-documentation-in-software-development-types-best-practices-and-tools/.

Figures

Outsourced images, examples or diagrams will be included in this section.

02. WCAG Colour Contrast Checker

"Contrast Checker." WebAIM. Accessed December 2, 2022. https://webaim.org/resources/contrastchecker/.

03. Red-Green Color Blind Example

"Contrast Checker." WebAIM. Accessed December 2, 2022. https://webaim.org/resources/contrastchecker/.

05. Fly Agaric (the Polyaminita)

Fly Agaric. *WildFoodUK*. Accessed December 2, 2022. https://www.wildfooduk.com/wp-content/uploads/2018/01/Fly-Agaric-3.jpg.

06. Google Font Inter Example

"Inter." Google fonts: Inter. Google. Accessed December 2, 2022. https://fonts.google.com/specimen/Inter.

07. Google Material Icon Example

"Material Symbols and Icons." Google Fonts. Google. Accessed December 3, 2022. https://fonts.google.com/icons.

42. Diagram of a CNN

Sharma, Vinod. *A Simple Diagram Outlining the Process of a CNN*. October 15, 2018. *Vinodsblog.com*. https://vinodsblog.com/2018/10/15/everything-you-need-to-know-about-convolutional-neural-networks/.

47. TensorFlow Lite: Choose a Model Architecture

TensorFlow. “On-device object detection: Train and deploy a custom TensorFlow Lite model”. Accessed December 3, 2022 https://www.youtube.com/watch?v=sarZ_FZfDxs. Credit goes to TensorFlow.