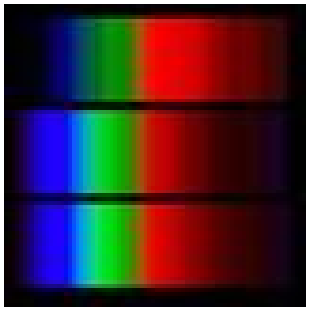


Final As-Built Report

2023-05-03

Project Sponsor: Joe Llama, Lowell Observatory

Faculty Mentor: Rudhira Talla, Northern Arizona University

Team: Empyrean- Henry Fye, Nhat Linh Nguyen, Jakob Pirkl, Jacob Penney, Kadan Seward

Overview

This document overviews the Empyrean Capstone project in its entirety. Beginning with the project introduction, our team's process in design and development, and the final project outcome which includes documentation and testing.

Table of Contents

1.0 Introduction

Astronomical research has proven to captivate the minds of modern people: the Space Race of the 20th century wrested the attention of millions across the developed and developing worlds; minds such as Albert Einstein, Carl Sagan, and Stephen Hawking have become icons of science, popularizing not only interest in the heavens but science more broadly; and research institutions and associated discoveries, like NASA and its illuminating missions, Flagstaff's own Lowell Observatory with the discovery of Pluto, the James Webb Telescope with its crystal-clear images, and events like the acquisition of the first image of a black hole [1], all pique and tantalize the public. In fact, NASA themselves and institutions such as the National Science Foundation and the Marley Foundation provide millions of dollars a year in grants to other astronomical research organizations, like Lowell Observatory [2]. According to one unofficial count, there are a total of 349 other professional observatories operating telescopes in the United States right now [3].

At these institutions, researchers like our client, Dr. Joe Llama at Lowell, spend their nights and days collecting and analyzing invaluable astronomical data. In particular, Dr. Llama's research “is broadly focused on stars and exoplanets, and the interaction between the two" [4]. Currently, he uses a variety of tools "to measure the masses of Earth-sized exoplanets" and, using the data that he collects from these tools, he attempts to "understand how activity from a star from spots, faculae, and plage limits our ability to detect the smallest exoplanets" and "determine methods to remove these signals from the data we collect on other star systems" [5]. This work is undoubtedly very interesting both to the public, with the concept of habitable exoplanets gaining interest as a popular scientific topic in the past twenty years [6], and to the United States Government, who is hunting for these so-called "Goldilocks planets" [7]. Among the aforementioned tools that Dr. Llama uses to conduct this in-demand research are spectrographs, which are "instrument[s] for dispersing radiation (such as electromagnetic radiation or sound waves) into a spectrum and recording or mapping the spectrum" [8]. In Dr. Llama's case, the spectrograph is attached to the end of a telescope, which accepts light from a solar-type star and splits it into its component wavelengths. This allows the astronomer to gather valuable information, such as about the composition of objects orbiting the targeted star. One could see how this tool plays a critical role in the workflow of both our client and many other professional astronomers; with it, the hunt for habitable exoplanets becomes much easier.

Problematically for our client operating some of Lowell’s newest spectrographs is currently inconvenient and costs a great deal of time. With the workflow he has now, our client must be present at the observatory to conduct observations, with no ability to perform them remotely. Furthermore, these operations are complicated because of how “low-level” the interface exposed to the user is, contributing to the aforementioned sensation that the process is inconvenient and costly, both in terms of time and money. These issues constitute unnecessary, and removable, hindrances to the work that he and other astronomers perform.

To solve the problems our client and possibly other observatories are currently facing, we have developed a web application which manages the operation of a spectrograph and a connected camera in order to conduct observations and receive the resulting astronomical data

back in an accessible, intuitive manner. This solution involves a front-end website, back-end API, and a database. The website allows the astronomer to choose what to look at and for how long. This request is then sent to the back-end API, which communicates this job request to the spectrograph. Once the observation is finished, the data is sent back to the back-end server to be processed and stored for future use in the database. The result is a much more intuitive workflow that is remote, easily adopted by new users, and automated so that a user cannot send the data to the wrong place. Our project allows astronomers to gather data from anywhere, furthering their important research by lowering barriers to access.

2.0 Process Overview

When considering our process after completing the product, our team agreed that we worked well together, felt lucky to work with each other, and emphasized that this was due in large part to a fluid, relaxed internal dynamic, which we would argue manifested in other areas, such as our development cycle and external communication style.

Over the course of the year our team spent planning and developing our product, we succeeded by allowing our team members to move fluidly through roles and responsibilities by adopting tasks that they wanted to learn about or felt that they would be competent with. This resulted in technical enrollment from all members and each member contributing to the overall guidance of the team's course. The only true exceptions to this fluidity are that external communication was managed by our team lead and notes were recorded by our recorder. Concerning the aforementioned technical enrollment, our team would not say that we abided strictly by Agile development principles but we did borrow much from that development process style. For example, we checked in and provided new content to our client often, iteratively delivering features and fixes to our product and leaving room for changes in requirements, which we had a few of. We emphasized collaboration with our client, who had a technical background and programs, and maintained a healthy relationship through thorough, ongoing communication and responding to changes in expectations and desires by adapting when possible. An example of our workflow during constructing a new feature or making a fix might be that there is an issue up which records ideas about changes that must be made to the codebase. If an individual felt interested, they would elect to pick up the ticket. Another member may have experience with that topic, so the first member may ask them for input and help as they worked on the ticket. If a member had not written anything in a while, they may be asked to assist by picking up the ticket.

The tools and workflows that we used to facilitate this process were ones that we were all comfortable with and decided upon early on, then relied upon for the duration of the project. For example, we strictly used Discord for internal communication, email for external communication, and Git and GitHub for all tasks related to managing development, such as managing issues or using GitHub Projects as a Kanban where we could assign or adopt development tasks. Concerning development itself, we placed no strict requirement on tools and all were allowed to use their tools of choice. For example, we used four different text editors (only two members used the same one) and three different operating systems. The result was an environment where everyone could use what felt natural and comfortable to them, which we would argue allowed us to produce work at a healthy pace and made our members comfortable and content.

3.0 Requirements

A requirement is a major milestone in the development of the Spectrograph Control System. Our requirement specification includes the functional and non-functional requirements. This indicates the specific features and performance for the Spectrograph Control System

3.1 Functional Requirements

Functional requirements are units of functionality that encapsulate the main features that users will interact with when they use the web application. They are considered as high-level functions which are then decomposed into the more detailed, lower level functionality required in the implementation of the application. Correct implementation of following specifications will ensure an outstanding experience among users.

3.1.2 Website UI

Astronomy Tools Login: The goal of this project is to create open source software that can be easily used by any astronomers that want to take advantage of a remotely controlled spectrograph. Our client considers the global scope of the product as the key requirement because the spectrograph and telescope are commercially available there are other institutions using similar setups to us, so we could make this control system publicly available for use by others, widening the reach of this project to beyond Lowell and NAU. However, each user is responsible for his/her own observation dataset and team Empyrean proposes the solution as the login feature. For each astronomy tool (spectrograph, telescope, etc.), the website plans to require users to login into their saved accounts in the database, and then if successfully login, the website leads users to their private pages containing user's information, in-used astronomy tools and observation data. Otherwise, if the user is totally new to the website, they will be asked to finish a signup form including the user's information and reason to use the software. This form will be sent directly to the administration team for consideration and this team has a responsibility for a quick response in approximately 1 week.

Observations Reservation: It is undeniable that astronomy contributed important achievements to space science in particular and humanity in general. To gain these achievements, astronomers and astronomy tools play an important role. In particular, astronomers need to utilize the astronomy tools every night and spend several hours observing the object. Our client realizes this difficulty and team Empyrean demonstrates the reservation feature for observation. More specifically, at the page for observation reservation, there are 3 main information required to be entered from users and they are time, object need to be observed and duration time for observation. After entering this information, users can reserve the observation by pressing the button "Start Observe" shown below the information box. On the observation date, the system will check the availability of astronomy tools (spectrographs and telescopes) and send an email to announce the tools availability to users. Assuming that spectrographs are available to conduct the observation, the system will collect and store the data from the observation in the filled duration time. Otherwise, the email will say clearly the

problem and automatically set the observation to the next day compared to the previous booking date.

3.1.2 Backend API

Remote Setup: It is obvious that spectrographs and telescopes offer great functionality for astronomers to observe the sky. However, the setup for these tools is massively complicated in terms of hardware and software. In addition, our client mentioned the difficulty of setup on-site and his expectations to solve this problem as soon as possible. Thus, team Empyrean considers the remote setup functionality as one of the most vital parts in the MVP. More specifically, the spectrograph and other tools are assembled and installed with all the required hardwares. Then, our product will connect with tools by using an online package which is introduced by our client.

Ease of Adapt Different Tool: It is obvious that our product or the system needs to connect various astronomy tools including spectrographs or telescopes. The website is required to connect to different spectrographs or telescopes, which leads to reducing the operation load for one astronomy tool and increasing the productivity of the product. Currently, our client allows the team to work on the spectrograph and telescope if necessary. The back-end server connects to the tool by a package provided by Lowell Observatory and our client suggests using Python as the main programming language to utilize the package.

Data Configuration Detection: Besides the login feature in the website section, team Empyrean and our client considers the data configuration file as the substitution for the default login feature. The data configuration file is used to access the observation data without login into the account and astronomers can save it at their computer folders or share it to others by sending only one data configuration file. More specifically, the team confirms that the data configuration file contains the object's name, observation time and username. User uploads the data configuration file to the website page and then the transition from front-end to back-end is going to operate. When the back-end successfully receives the configuration file, the necessary information is extracted and the back-end retrieves the certain observation data. This feature is inspired by our client, and during the development progress, team Empyrean will adjust including adding new things and delete current steps if necessary.

Astronomy Tools Availability Detection: The spectrograph is an un-stopped astronomy tool, which means that astronomers cannot control it when the spectrograph is doing another task. Also, the setup for spectrograph is complicated and astronomers have already set up the working spectrograph. Thus, astronomers need to wait for the current observation, then they can start their observation without resistance. To save their research time, the back-end is responsible to check the availability of spectrographs or telescopes and show the status on the website. If the current tool is busy, astronomers are free to change to others without setting up information from scratch. Currently, team Empyrean will work on 1 spectrograph and plans to accomplish the availability detection for that tool. Besides, the transition to other tools has a plan which the back-end creates and sends the observation configuration file containing the observation's information (date time, object and duration time) to an available tool.

3.1.3 Database

Complex Querying: The database must offer the ability to query the stored data in complex and variegated ways. Our client has a few definite, simple default queries that must be accessible via the front end interface, but more complex queries may be requested in the future and so should be accessible via the Relational Database Management System that we choose.

ACID Compliance: A very important quality of the database we choose is that it must offer strong data integrity and protections. **ACID**-compliant databases ensure that [11]

1. An operation must completely finish, or else the file, document, transaction or database reverts to its prior state, meaning that no partial changes may occur as a consequence of an operation.

2. “Any transaction you complete in the database follows the rules you or others gave to the database”, meaning that rules for tables, or structures in general, cannot be broken by a query.

3. Every read and write operation occurs in isolation, meaning that the effects of disparate operations do not “layer”, or happen in an overlapping manner. This slows down the database but protects data.

4. Data is written to long-term storage, meaning that operations on the database are not stored in primary memory. This prevents data corruption or loss in the case of some catastrophic failure in which primary memory fails.

The database must offer this functionality to the user to prevent the loss or damage of important data

3.2 Non-functional Requirements

A program that performs all of the functions required of it does not make a complete product. Simply satisfying all of the functional requirements has the potential to leave a job unfinished. To more fully round out the expectations of a finished product, it is useful to know how fast it is expected to perform key tasks, how simple it should be for new users to pick up, how accurate certain measurements should be, or various other markers that can be tracked. The following describes the expected performance requirements that Empyrean should meet as the project is completed.

Spectrograph Startup: As it stands, it takes several hours to set up the spectrograph. The automated setup that will be developed will take no longer than that amount of time. This is based on the assumption that this startup task, which includes calibration, is not slow because of human error, or slowness, but because proper calibration of the tools requires significant time. Thus, the overall time of setup will not be reduced by any major factor. However, if this is not the case, and calibration does in fact require a lot of human input, then it would be reasonable to make the case that Spectrograph startup will be significantly faster once automated. This will make remote access not only possible, but preferable to how Dr. Llama's current workflow is structured.

Response Time/Observation Time: The time that it takes for this proposed system will take as long to make observations as the old system. Much of the observation time is the exposure, as they can be around ten minutes long for Lowell's purposes, so there is no ability to significantly reduce the time taken for the total observation. What can be measured is the response time of the spectrograph, and Empyrean will ensure that requests made around from anywhere in the world will take less than 3 seconds from the application to the spectrograph that a request is made to. This expands on the previous point to let our astronomers access their equipment at any time, from anywhere.

Object Selection: Selecting an object for observation will, on average, take one second. This will be much faster than the dozen seconds that it takes for the current system. Currently, astronomers must select not only the star, but find its coordinates, choose what kind of file they want their output data in, where it will be stored, and other configurations. Because of the structure of the proposed system, many of these configurations will be standardized with no choice by the user, reducing time. Compared to manually finding the place in the sky where the desired object is located, our system will be able to automatically describe the location of this object, reducing the time needed to form a request, without the need for the user to select so many other options. This helps to prioritize the ease of use for astronomers.

Global Availability: Due to the nature of Astronomy, much of the work needs to be done at night, meaning it is inconvenient to be at the telescope while collecting valuable information. Also, there is often a need for astronomers to be able to use telescopes they cannot physically access, which can be anywhere. Which is why this product must be able to function anywhere in the world, with no physical restrictions except an Internet connection.

Data Availability: Being able to easily measure data has little meaning if it is time-consuming to access said data. Therefore, access to collected data must be quick when compared to the previous method of data collection. Specifically, Empyrean will ensure that the average time to collect each record of an observation will take no longer than one half of a second as long as the user is in the same Internet subnet as the database hosting the data. This guarantee cannot be ensured globally, because as the number of records requested and distance increases, the chance that some of the data gets lost from the server to the user increases, making it inconsistent to promise such a short time frame.

Real-time Messaging: To keep track of the various subsystems status updates will be sent from the backend to the frontend for users to monitor. It is critical that these status messages are up to date in case something has gone wrong. Empyrean will ensure that these updates are “real-time” - meaning that within two seconds of the subsystem notifying the messaging server of its updated status, it will then update on the users frontend interface. This is to ensure that Dr. Llama has confidence the spectrograph is operating normally, even when remote.

Ease of Use: One of the major goals of designing a new system for managing these spectrographs is that the learning curve is quite steep - in other words, it can be difficult for new users to understand how to use their software at first. For this project, users should be able to have a simple understanding about intended usage the first time they access the app.

Upon entering the homepage for the first time, a user should be prompted to login, sending them to a landing page. This landing page should be able to lead a user to any of the major features of the site: Taking Observations, Viewing Previous Data, etc. Upon selecting any of these options, the resulting page should have plenty of prompts and help to help a new user become instantly familiar with this new system.

Time to Unfilter Data: In viewing their previously collected data, astronomers will be able to see the unfiltered data in case they see some data that might be configured improperly. This must be rather quick - users often resend requests after one second of inactivity. Thus, this reprocessing of the data must occur in less than five seconds. We are giving extra allowance due to the difference in user base from the average Internet user. While the average user of any given website might reload the page after one second of inactivity, astronomers know the size of data that they work with and have a more thorough understanding of the data handling and so reasonably will understand it will take time in order to process their request.

Stalling/Reliability: Another of the issues with the current system is that it occasionally stalls with no apparent reason. With this new system, it will be ensured that when the app is not functioning, it will either be a network error, or a physical error with the tool, such as when a motor falls off of the tracks of the observatory.

Password Security: Logins must be kept secure, especially when accessing expensive equipment like those kept at Lowell Observatory. Each person should have their own account, unable to access the accounts of other users, of course. Thus, we will ensure that there is

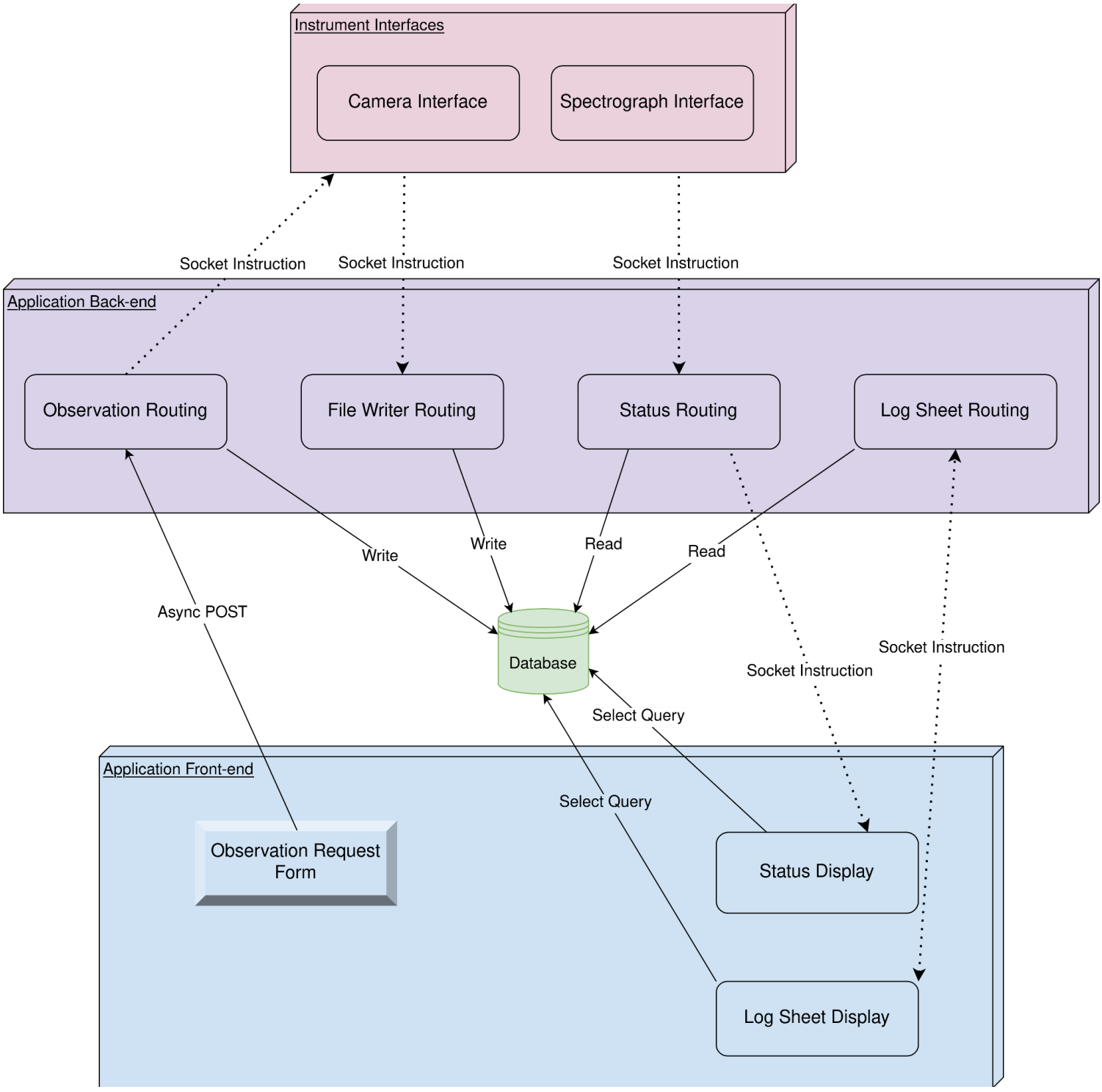
secure storage of every user password. Security in this context means that none of the passwords will be stored in plaintext, only their hashed values, as well as all of the communications between the messaging server and the frontend regarding authentication need to be encrypted. This encryption should not be breakable with current non-quantum hardware in less than 10 years.

Uptime: Empyrean will ensure that our program will be functional no less than 99% of the time, so that from anywhere, Dr. Llama will be able to access data from past observations, or be able to make observations whenever needed. This is because software that is not operational when the astronomers need to use it hampers the collection of data that is needed to make discoveries.

Time Between Observations: Between observations, the data that the spectrograph took during the previous exposure must be written into a FITS file to be stored and used later for analysis. As it stands, this sequence of events can take between one and 30 seconds depending on the amount of data, but considering the data we will be working with, the time between the end of one observation, to the next observation should take no longer than twenty seconds, including the file writing, and storage into the database.

4.0 Architecture and Implementation

Starting with the conceptual background of our web application's architecture, it distributes responsibilities using the client-server model but with the added complexity that all astronomical instruments it manages are instantiated as clients to the server. The consequence is an architecture where information is gathered or created and consumed at one of two ends, either the Front-end or in the instruments, and is moved back and forth in what is essentially a circular feedback loop. The diagram below illustrates these relationships more clearly, and we



will guide the user through the responsibilities, features, modes of communication, and expected outcomes of each segment of the architecture.

4.1 System Overview

4.1.1 Front-end

The Application Front-end, here at bottom and in blue, is the entrance to the functionality of the application to the user but is, again, one of two ends or destinations in the application where information is created or consumed. The responsibility of the front-end is to provide a user interface to the usage and management of the astronomical instruments attached to our system, and the intended qualities of this interface are minimalism and ease of use, such that only the information that is pertinent to the user is displayed and it is displayed in a manner which is sensible and easy to understand. The main features of this part of the application are

1. The Login Portal (not displayed above): allows the user to use their credentials to log in to the system and access the other provided features, discussed below. To send the user's credentials to the system back-end to be validated, an asynchronous POST request is made.

2. The Observation Request Form: allows the user to populate a form containing instructions for the astronomical instruments on the job the user wants them to perform, such as what objects to observe, what type of images to take, how many exposures to take, and how long each exposure should be. After this form is submitted, the instruments will perform their tasks. To submit the form, an asynchronous POST request is made which contains the user input.

3. The Status Display: displays pertinent information about the system and its ongoings to the user. Examples of this information include whether the system is busy; the current status of the camera, such as whether it is busy, idle, or otherwise; and how far through the current set of exposures the camera is, assuming that it is working on a request. This component consumes information that is sent to it by a SocketIO emission from the application back-end.

4. The Log Sheet Display: displays pertinent information about current and past exposures to the user. While the observation request form allows the user to think about and request exposures as part of a series or batch, the log sheet display currently discusses each individual exposure that has been taken, providing information such as what was observed or what type of image was taken, when the image was taken, and what the status is of an observation that has been requested. Like the status display, this component displays information provided to it by SocketIO emissions from the application back-end.

5. The User Management Portal (not displayed above): allows the user to create, delete, or modify users and their settings. This component is capable of producing new data and communicates with the application back-end via asynchronous POST requests.

4.1.2 Back-end

The Application Back-End, at center and in purple, is a messaging server, or middleware, which accepts inputs from all clients and routes the data contained therein to the recipient. An example of this might be that the middleware will receive an observation request from the front-end, deconstruct the request, and pass the resulting information to the instrument that requires it, such as passing to the spectrograph what type of image the user would like to take. Here, I will discuss only the modules displayed in the image above (unlike in section 4.1.1), which are those required by the minimum viable product. From left to right:

1. Observation Routing: processes observation requests sent from the front-end and passes each piece of information in the request to the relevant location, instantiating rows in the application database as needed (whenever an exposure must be instantiated for the Log Sheet display, for example). This component communicates with the database via Flask-SQLAlchemy and with the application instruments via SocketIO emissions.

2. File Writer Routing: when an exposure has been completed, the resulting data is sent from the instruments to the application back-end (a process we will discuss further in section 4.1.3). The File Writer module is the one that receives the resulting data and handles it. The file writer will use the resulting data to update and populate rows in the database which correspond to the observations that were just taken and will write the actual image data into a FITS file, a file format commonly used to hold astronomical data. This module receives SocketIO emissions from the application instruments.

3. Status Routing: during operation, the application instruments constantly output statuses that can be listened for and passed to the front-end for display. This module listens for these status emissions, captures them, processes them, and routes them to the front-end. It receives these emissions and forwards them using SocketIO emissions.

4. Log Sheet Routing: This module listens for signals to update the information shown in Log Sheet display in the user interface. It also listens for and passes information via SocketIO emissions.

4.1.3 Instruments

The Instrument Interfaces, at top in red, provide a means for the rest of the system to interact with the instruments it must manage. They are a generalized means of adapting the communication abilities that the instruments have such that our application can perform the jobs described above. During normal usage, these components accept SocketIO emissions as

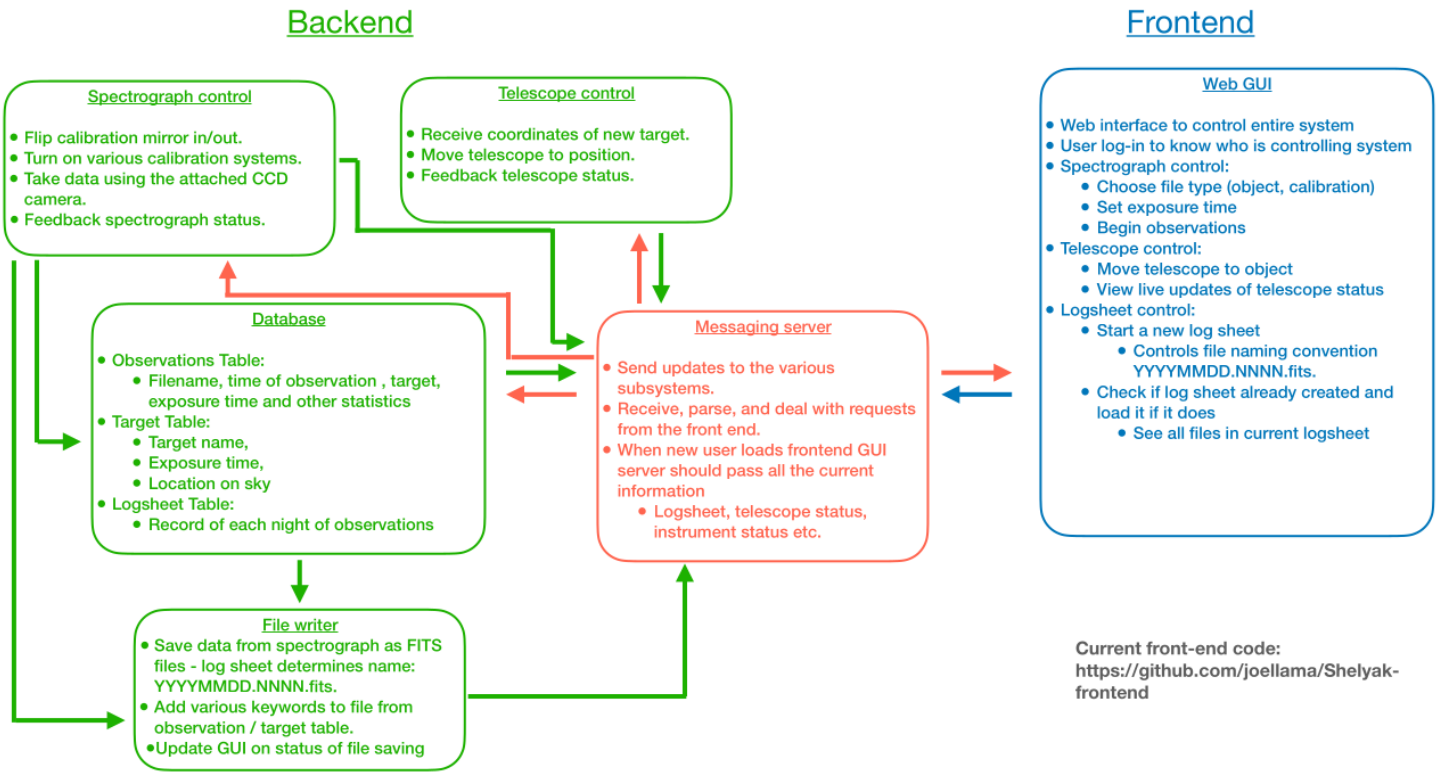
instructions with input, perform their tasks (all the while emitting status updates about their job), and ultimately output data to the application back-end via SocketIO emissions themselves.

1. Camera Interface: the camera interface generalizes functionality that our application requires to start, stop, and get information from any astronomical camera attached to our system. It accepts observation instructions, such as those described in the Observation Request Form functionality of the user interface, and a list of observation IDs, created by the database on demand, to gather data about, via SocketIO emission from the application back-end. After they are received, the camera begins to work on each exposure (each of which correspond with an observation ID, which acts as a request ticket for the exposure), for which it will produce image data that it writes to a temporary file, the path to which is given to the back-end again via SocketIO.

2. Spectrograph Interface: the spectrograph interface generalizes functionality that our application requires to change the image type that the spectrograph will take. This requires being able to turn ports in the spectrograph on and off. It accepts these instructions via SocketIO from the application back-end and emits statuses via SocketIO. It has no data that it returns other than statuses.

4.2 Prescriptive and Descriptive Architectures

There was little difference between the architecture described in our requirements and the one that we produced. The above diagram is the architecture the team derived from our



requirements during requirements acquisition, which we performed in Fall 2022. The diagram shown here is an initial concept provided by our client which we relied upon heavily during the aforementioned requirements acquisition phase. One major difference between our descriptive and prescriptive architectures is that the database is not drawn upon by the file writer; the file writer module, as it is in our implementation, does not need to read information from the database, only to write to it, making the arrow in this diagram from the “Database” module to the “File writer” module unnecessary. There was no functionality in the minimum viable product requirements that we did not produce and, consequently, our architecture remained relatively similar to that prescribed.

5.0 Testing

As with all software, an application needs to have thorough testing. Software testing is the practice or process of evaluating the efficacy of software at completing its intended purpose. Testing can be performed manually, for example by clicking the button of a website to ensure that it functions, or automatically, using a variety of types of tests built as software that execute various functionalities or parts of a software system and check what the output of that execution was. Automatic testing is protean: it can be performed at various scales and granularity, allowing software engineering teams to test fine pieces of functionality, such as guaranteeing the output of a single function, up to the system level, testing how massive tools integrate with each other. Our product, a web application with a relatively typical structure for such a product, requires a few different types of testing in several places.

5.1 Unit Testing

5.1.1 Introduction to Unit Testing

Unit testing is one of the most common ways of testing code, relying on concepts such as modularity to drive assurance. Unit tests are written at the smallest level of code, usually a function or method, testing their validity with known outcomes. These tests are designed to test the smallest pieces of code, without interference with other functions or other parts of the system, such as a database, or other servers.

The goals of unit testing are multifaceted. One reason is to increase documentation. The tests themselves provide documentation, as future writers will have some idea of what inputs to the function will return. Going back to write tests for older functions also forces writers to read their functions and add documentation to sections that were not as obvious. This can also lead to rewrites of code in order to increase clarity. Unit tests also allow programmers to check their work, since once a function is written, the function can be checked by writing a unit test for the function to ensure proper functioning.

The next two sections will detail the process for testing each language used in the makeup of the code, python and javascript. This will begin with a breakdown of options for testing, followed by a list of units that will be tested, along with the boundaries of those units and what erroneous values will be provided to test robustness.

5.1.2 Python Unit Testing

Due to the popularity of python, there were many available options for unit testing. The three most common options for writing unit tests in python are doctest, unittest, and pytest. Doctest is by far an outlier when compared to the other two. This testing framework works via native docstrings. Through this, it can run the code and verify that the outputs are what the docstrings specify. We decided against this framework however, as it bulks up each function's code, increasing the length of files, and is unintuitive to learn. Another option was unittest, python’s native unit tester. This is a more popular option, being based on JUnit for Java. That means it has common syntax. However, we also decided against this option as default operation of this module lacks feedback for the user running these tests. It is very common for the only results being the percentage of tests that pass without any further information. Pytest is even more popular than unittest, despite being a third-party package. Additionally, there is less boilerplate as Pytest is a python package through and through. It also provides more robust feedback from tests, even going so far as trying to find the line of code that offset the results of

the function. Because of these features, and in the spirit of our project, if made more sense to test python with pytest.

The following are the units that are tested on as well as the boundaries and incorrect values that will be provided to each unit. Notable exceptions from the list of units are functions that interact with our database, as that is a kind of integration test, one that is conducted by the creator of whatever library we have chosen to use. Additionally, similar functions, such as the constructors of model classes will all be the same. Included will be the code in instrument files which control the spectrograph and camera if they are entirely internal. Thus:

We will focus code on the backend of our application. Most of this code deals with database access, and thus is not in the scope of unit testing. These are the constructors and methods of each of our models, and one function which deals with file writing.

Instrument Model's Constructor: This method creates an Instrument object which can be submitted to a database.

Equivalence Partitions: The only argument for this method is a name for the instrument, which is a string, so the partitions are along the type of the given argument. The acceptable arguments are non-empty strings, such as "ZWO ASI2600MM Pro". Then blank arguments (strings composed of whitespace) will err. Empty strings will also err e.i. " ". Finally, objects that are not strings, including None will err, as a name is required for our database.

Observation Model's Constructor: This method creates an observation object to be submitted to the database to be used in the future. Unit testing this will also test the class's set_attrs method, as the constructor is a wrapper for the function.

Equivalence Partitions: This function takes in an initialization dictionary, as there are many fields that need to be set in the database. This means that many of the fields will need to be checked in the same way as the previous unit, such as obs_type, and date_obs, will need to be checked. The dictionary will also need to be checked. Entries that are not in the attribute list of the class will need to be checked, such as "NOTREALATTR": "HELLO", which will need to fail. Finally, different types of objects will be the most obvious failure of this function, where any non-dictionary will err.

User Model's Constructor: This method creates a user object that can be used to login, which will be stored in the database.

Equivalence Partitions: This function takes in two strings as arguments. Refer to the Instrument Model's Constructor to learn about the partitions of one string. This is the same, except that there are two strings, both of which are required, so any combinations of the above scenarios can happen.

Status Model's Constructor: This method will create a status keeping track of the instruments during the operation of the app, which is stored in the database.

Equivalence Partitions: Like the above function, two of the values are strings, both required, so they have the same partitions. There is also an InstrumentID value, which refers to the instrument to which the status is connected. This is a positive integer, so correct values would be, for example, 5. Incorrect values would be other numerical types, like floats like 5.3, non-positive numbers like -1 or other types, like "Hello".

File Writer's __init_fits_abspath: This function figures out where to store data collected by the camera and returns the file path where it will go.

Equivalence Partitions: This function takes in two arguments, the file path to the directory where the FITS files are stored, and the name of the temporary file that the camera has provided. These are both strings, so they can first be tested in the same ways as with the Instrument model's constructor, but these strings are more specific. They must be a filepath and file name, respectively, and so all of the restrictions to those will be applied. As an example, a file path with whitespace in it cannot be used, like " / home/user/ Desktop/Data". A correct input may be "/home/user/Desktop/Data". For the file name, forward slashes cannot be contained, such as "file/fits.fits". Whereas a correct input may be "obs1".

5.1.3 Javascript Unit Testing

Similarly to python, there are many available options for unit testing for React and Javascript as a whole. While there are far many more options for JS, the three most widely used options for writing unit tests in Javascript are Jest, Mocha, and Storybook. Jest is the largest and most frequently used unit testing interface for React on the market. It is easy to use and has no setup, meaning unit test development is quick and easy. Its main feature is being able to take snapshots of large amounts of test cases, allowing for a larger variety of test variables for specific functions. Next is Mocha, which was the most commonly used testing framework until recently. The main drawback with Mocha is the overly complex setup process, however the testing suite is incredibly clean and can write complex and flexible tests for both front and back end JS. Finally there is Storybook, which acts more like a development tool with a built in testing suite. It allows you to create independent components that interact with each other and allow for ease of testing and documentation. It is commonly used for large complex projects with many pages, each with many components. Because of all these different choices, there was a discussion on which interface would be most useful for our specific project. We ended up going with Jest for its ease of use and hassle free setup.

The following are all units that are all tested on with correct, incorrect, and invalid inputs for a given unit. As our frontend is a large and complex system, with several interlocking components and pages, some unit tests might include 2 functions or more, so long as no major processing occurs between functions. This will not affect the test conditions as all function inputs and outputs will be logged. There will not be any testing for any React specific components, as they have already been thoroughly tested and documented. The front-end code is entirely based on the website and the UI, and as such will be closely coordinated with other integration tests involving the front-end so that there is not any extraneous overlap, whilst also testing all features and functions.

attemptRegistration - Registration Page: This function sends a post request to the backend and contains the logic to allow a new user to be added to the system.

Equivalence Partitions: The arguments for this function are the fields that exist on the web page in the form of asynchronous values. These fields are username, observer name, and password. The acceptable arguments are determined by the backend and as such any non-NULL ASCII string is a valid input.

attemptLogin - Login Page: This function sends a post request to the backend and contains the logic to login to the system with a valid username and password. The function will navigate back to the login page in the case of an incorrect username and password. Otherwise, the user is directed to the observation page.

Equivalence Partitions: The arguments for this function are the fields that exist on the web page in the form of asynchronous values. These fields are username and password. The acceptable arguments are determined by the backend and as such any non-NULL ASCII string is a valid input.

getChipProps - Log Sheet Component: This function gathers the data of a row in the logsheet table for later display.

Equivalence Partitions: The arguments for this function is a row from the log sheet table in the backend. This includes name, current progress, and icon. The acceptable arguments are pulled from the database and therefore will always be compliant in form and content.

showProgress - Log Sheet Component: This function takes a valid row's completion status and displays it as a loading bar.

Equivalence Partitions: The arguments for this function is a property of current progress in an observation. As this information is pulled from the database using getChipProps, the data will always be properly formatted and easily accessible.

initResolution - Observe Component: This function sends a post request to the backend to resolve an astronomical object into valid observable coordinates.

Equivalence Partitions: The argument for this function is the text field of the object to be requested. The backend deals with whether this is an acceptable astronomical object, and as such the valid arguments are any non-NULL ASCII string.

validateObservationRequest - Observe Component: This function checks that there are no current issues with any values being requested for observation. This checks the coordinates set by initResolution as well as user input.

Equivalence Partitions: The input for this function are all the fields dealing with an observation. Each field has regex checking that will return an error in case of incorrect formats. Thus, this function will check if there are any errors, and as such there are no incorrect input values.

initObservation - Observe Component: This function sends a post request to the backend to begin an observation with the current observe field values.

Equivalence Partitions: The argument for this function is the text fields of the object coordinates to be observed as well as observation duration and number of exposures. The validateObservationRequest deals with whether the values are acceptable astronomical inputs, and as such the valid arguments are any non-errored string.

endObservation - Observe Component: This function sends a post request to the backend to end an observation and log the results.

Equivalence Partitions: There are no input values for this function, as it is simply sending a post request to the backend and logging the response, or sending an error if one occurs.

updateData - Status Component: This function updates the statuses for the instrument. This involves setting color, text, and label for all of the current statuses.

Equivalence Partitions: The inputs to this function are all of the current statuses in object form. Each object is checked from a request to the database and its label, current status, and color are updated. There are no incorrect inputs as they are all status objects.

5.2 Integration Testing

5.2.1 Introduction to Integration Testing

It is undeniable that a typical software project consists of multiple software modules, coded by different programmers. Because of this, developers need to test the integration or interaction between modules and make sure that they work properly. Thus, integration testing becomes an ideal software testing that verifies the proper functioning of individual software components when they are combined and integrated into a larger system. The main objective of integration testing is to identify and eliminate any issues or defects that may arise when different software modules are integrated together.

5.2.2 Front-end to Back-end Integration Testing

The software related to the spectrograph requires the connection or interaction between front-end, back-end and devices' systems. Thus, team Empyrean has discussed and illustrates that the integration testing is important because it helps to ensure that the two parts of the application work together correctly and provide a seamless user experience.

The front-end is responsible for presenting the user interface and allowing the user to interact with the application, while the back-end is responsible for processing requests, retrieving and storing data, and performing any necessary calculations or logic. With the integration testing, some issues can arise when the front-end and back-end are combined, such as data transfer, compatibility, communication, and security vulnerabilities.

More specifically, the integration testing for Front-end to Back-end used the bottom up approach.That allows the team to start at the lowest-level components of the system, such as backend APIs or database queries. These test cases concentrate on the accuracy of response from front-end to back-end and the interaction between the back-end and database.

Valid User Login/Signup: This function indicates the test cases for the login and signup service of the product. More specifically, when the client sends their information including email, name and password through the login form, the function in the backend will receive the same information package and create a database object 'Account' for that client. Thus, it is clear that this test case will test for correctness of the request sent through the backend and whether the database server creates an "Account" object for the client.

Resolve Functionality: This function indicates the test cases for the resolve functionality in the Observe form. The resolve function is responsible for generating the coordinates of the object. The test case creates a driver function that sends a request to the back-end API and verifies the response data. Moreover, the test case checks the correctness of response back from the library.

Observation implementation: The test cases verify that the observation implementation in the product is functioning correctly. The test case sends a POST request including observation data to the path "/observation", then it checks whether a database object is created for the observation. The expected results are that the observation is displayed

correctly in the component log sheet. The observation has a timer to countdown the observed time and displays “Completed” when the observation runs out of time.

Log Sheet Implementation: the test cases verify that the log sheet implementation in the product is functioning correctly. The test case sends a POST request including observation data to the path “/observation”, then the test verifies the request package is received in the backend. There are 2 cases when checking the log sheet, and they are the current log sheet or new log sheet message. In the case of the current log sheet, the test checks the existence of that log sheet database object; otherwise, the test checks the log sheet object initialization. The expected result is that the chosen log sheet even the created one, is expected to display in the component log sheet

5.2.3 Back-end to Instrument Interface Integration Testing

One of the stretch goals that our team accomplished with this project was the implementation of an interface to external astronomical instruments. This interface defines a series of methods that must be provided by all instruments that desire to connect to the backend of our tool and function properly. Our current implementation allows us to narrow and focus our integration testing between these two categories of subsystems on these provided points of access. To test the integration between these tools, we will rely upon a test harness called PyTest. Pytest is one of the most popular Python testing frameworks that currently exists and is claimed to be simple and approachable. The fact that it is both popular and purportedly simple fits the project ethos discussed in our introduction. The following is a discussion of the interface and its endpoints, and how we can exercise them to ensure their proper functioning, and what outcomes should be guaranteed.

The “instrument” abstract class defines the common interface used by all external instruments. It contains several “magic methods” (special methods capable of being provided by Python classes) as well as public methods, cumulatively allowing for one to instantiate, delete, and manipulate an object of this class type.

Initialization: We provide a test which ensures that instances of the instrument class are initialized properly and completely. During initialization, the instrument must

1. connect to the host (the backend of our application)
2. assign its name and, using that name, retrieve its unique instrument ID from the application
3. Initialize its hardware statuses in the application’s database
4. Initialize its unique callbacks

To test this functionality, we must guarantee that steps 1, 2, and 3 transpire properly. This a contract assumption that we can verify by

1. Checking the status code returned by the socket emission which connects the two devices
2. Checking that the instrument receives the proper ID
3. Checking that the database contains initialized statuses after connection

Deletion: We provide a test which ensures that instances of the instrument class can be deleted without side effects or or the system crashing. To guarantee this, a simple test which deletes an instance of an instrument and checks the resulting status will suffice.

Retrieving the instrument name: This method is defined in the interface and guaranteed to be accessible to the back-end, but its implementation is hardware-specific. To guarantee its proper functioning, we must ensure that we receive the correct instrument name and a success status. The alternative is to provide an error status if the name cannot be reached for some reason. Because instrument names are defined in their firmware, we can check returned names of specific instruments against their known and confirmed names in the test source.

Updating the instrument's status: This method is defined and implemented in the interface and has no relationship to the firmware other than relying upon the statuses that the firmware emits as the hardware operates. To test this method, we must compare statuses received on the back-end for submission to the database with the statuses that exist in the firmware at the same moment. The test does not need to check if the statuses themselves are sensical, given that the firmware decides which status contents are emitted during a given internal event, but just that the back-end is receiving the emitted data correctly. For example, we can guarantee this functionality by provoking discrete events, such as the camera initializing, and comparing the status received from the firmware in the backend and the status actually in the firmware with the status that we know to be given during this event.

5.3 Usability Testing

5.3.1 Introduction to Usability Testing

In conjunction with unit and integration testing, a very important piece of the puzzle includes usability testing. Usability testing is where a team gathers actual feedback from real users to understand if their product is working not only as intended but in a way that is natural for their users. Often, the users will "play" with the system in front of observers while certain questions may be asked or user stories played out. This helps to ensure that the user is happy with the system and that everything is working as intended while also aiding the team in finding new problems that they may have overlooked or new recommendations that could be made. By conducting these usability tests, issues within the system were exposed and fixed.

5.3.2 User Testing Pool

Now that usability testing has been defined it is time to explain how this form of testing will fit into this specific project. As this project has a very niche clientele, the range of users that will be testing the product is very limited. Our main contributor towards usability testing was our client, Dr. Llama. Testing with our client occurred throughout implementation. The environment for the project was set up at Lowell at about the halfway point for development and our client worked to provide feedback as he used the product. The only other contributors to usability testing will be any astronomer that uses this open source software. This will be outside of the scope of the project for Empyrean but will allow users to continuously provide feedback for the product over Github.

5.3.3 User Stories

Outlined below are the user stories that were observed when conducting usability testing with our client:

Logging In/Out: The user will initially be prompted to enter a username and password. This will be created in preparation for usability testing and provided to the user. The user will also log out at the end of the testing session to ensure functionality.

Registering New User: Once logged in, the user will be on an admin account that allows the creation of new users. The user will navigate to the web page that allows registration and create a new user. The user will be asked to log in with these new credentials to test the system and then log back into the admin account.

Changing User Permissions: While the user is on the admin account, they will navigate to another web page where they will change the user permissions of the previously registered account. The user will again log in to this account to ensure changes were correctly made.

Requesting Observations Using Object: The user will be asked to make an observation using the Object tab. This will require the user to input an object they would like to view, press the resolve button to populate fields, and input the number of exposures/exposure duration. The user will click the start button and the log sheet will be populated with new entries

Requesting Observations Using Dark: The user will be asked to make an observation using the Dark tab. This section only requires a number of exposures and exposure duration. User will click start and the log sheet will display the observations.

Requesting Observations Using Flat: The user will be asked to make an observation using the Flat tab. This section only requires a number of exposures and exposure duration. User will click start and the log sheet will display the observations.

Requesting Observations Using ThAr: The user will be asked to make an observation using the ThAr tab. This section only requires a number of exposures and exposure duration. User will click start and the log sheet will display the observations.

Sorting Observations in Log Sheet: Now that the logsheet is full of observations, the user will be asked to filter and sort the logsheet to their liking. This includes hiding/showing columns, filtering specific observations, and changing the density of the table. This will also include downloading the current state of the table to a CSV.

Searching for Previous Observations: The user will finally be prompted to search for previous observations on the page labeled “Explore Logs”. The user is given two calendar components that represent a range in time and when two times are selected the user will be able to search for all observations in that time frame. The user can then navigate the entries using the functions mentioned in the previous section.

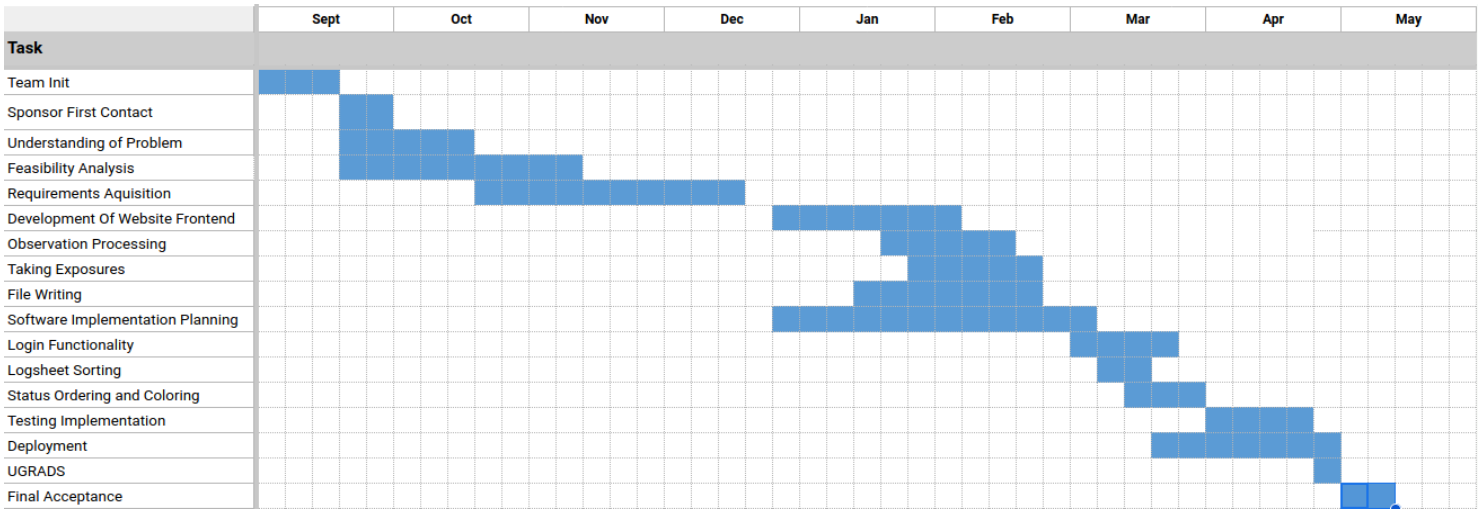
5.3.4 Continued Usability Testing

This project is going to continue expanding once our team leaves it at the end of the school semester and our client will take full control. With this, our client will be able to make specific changes to the project that are deemed necessary and will have to conduct his own usability testing in the future. Due to the open-source nature of this project, it is likely that other users will come across problems or recommendations and will have the ability to make

recommendations/contributions to the software through Github. This will ensure that the project is continuously evolving to fit the needs of astronomers all across the globe

6.0 Project Timeline

Empyrean knows that the only way to stay on track when working on a large scale project is to have a schedule to work from. Without a schedule, it becomes much easier to stay on track and focus in order to achieve our goals as well as the goals of Lowell. Below is our current schedule:



As can be seen, several of these modules have already been completed, which will be described below.

1. Feasibility Analysis: In this phase, the team decides the technology stack applied into the product. More specifically, the team collects opinions from members and the suggestions from our client, Dr. Joe Llama. Then, the team prioritizes the technology that provides the appropriate feature and strong connection to other components as well
2. Requirements Acquisition: The team discusses the specific features and performance expected for the Spectrograph Control System. All the members in the team provide great resources to analyze and research the necessary features for the System.
3. Log Sheet Presentation: On the main page, there is a mini module that shows recent observations that have been made by the user. Jacob and Henry are taking on this task, and have finished a first pass on this module.
4. Status Presentation: There is a section on the main page where the status of different aspects of the spectrograph, like the elemental lamps, or the status of the spectrograph. Jacob has done this module, and will be continuing to work on it as we set up the messaging with the spectrograph to make this module dynamic.

5. Exposure Picker: Also on the main page, the user will want to pick an object to observe with the spectrograph, and the Exposure Picker allows them to do this. Jacob is basing this module off of our prototypes from last semester, bringing it in line with the rest of our design.
6. Status Messaging: In order for the status presentation module to work as expected, a messaging system will need to be developed between the spectrograph and the frontend system. Kadan will be working on this module. Steps will be taken to make sure that this module will cooperate with the status module worked on by Jacob.
7. Exposure Requests/Messaging: After a desired request is input, this module will take that request, and give it to the spectrograph to be handled. Naht is going to be starting work on this, which will also have a component of connecting the exposure picker to this system.
8. Logsheet Generation: Similar to the previous module, in order to access their previous observations, the database will need to be accessed. Henry will be working on this, connecting it to their log sheet work from before as well as accessing the database in an efficient way.
9. Create Login/Admin: This will be the profile system for our service. This important task will be handled by Jakob, and will include the login page and logic, as well as user management on the server. The login applies the hash password algorithm to avoid being hacked from outside, which enhances the data security for Lowell Observatory
10. File Writing: This will require that we take the data from the spectrograph, and add all the parts needed to store in our database. Kadan will be handling this task, making sure that all of the data is stored correctly. This will also have a demonstration component that is not part of the final project, that will emulate the spectrograph that will be in use at Lowell.
11. Unique Observers: This will be a task that ensures that only one user can access a given spectrograph at a time. Jakob will be handling this task, as it is connected to the user login feature.
12. Store Observations: Takes the data written by the file writer, and stores it in the database. Naht will take this task. It will also take data from Lowell to use in order to test correctly.
13. Display Observations: Separate from the log sheet, this will display more raw data. Henry will take this task. This is also an extension task, but one that there is time to
14. Testing Implementation: All the members in the team are assigned for this phase. Kadan

and Jakob are responsible for Unit Testing in the backend and frontend respectively. Otherwise, Jacob, Henry and Nhat are responsible for Integration Testing in the backend - devices, frontend, and frontend - backend respectively.

15. Deployment: The team is responsible to deploy the product at the office of our client at Lowell Observatory. The result expected is that the client successfully uses the product in the Lowell network and applies it to do his research. Kadan and Jacob are mostly responsible for installing and debugging if necessary.
16. UGRADS and Final Acceptance: This phase requires the team to report back to the faculty about the progress through a professional presentation and poster review in the conference. Moreover, the team finishes the report documentation for the client as well as the future developers.

7.0 Future Work

This project is going to continue expanding once our team leaves it at the end of the school semester and our client will take full control. Future work that can be appended by either Dr. Llama or future open-source contributors will be listed below:

7.1 FITS File Visualization

One possible future addition to the web application would be a built-in FITS file viewer. As of now, whenever an observation is made, the data that is collected is transformed into a FITS file and stored in a specified location on the computer. If an astronomer wants to be able to look back at previous files they have to manually open their file explorer and then use a separate technology to then view the picture. It would be very beneficial to include a built in page on the site that allows for this functionality. After the astronomer completes their observations they would be able to navigate to a separate page that would allow them to select on the website the file they would like to view. This would essentially replace the need for a separate piece of software and simplify the process.

7.2 Multiple Log Sheets

One stretch goal that the team couldn't quite get to was the possibility of multiple log sheets. As of right now, the log sheet on the website is just one large table that holds all of the observational information. It would be beneficial to the observers if they could have different log sheets that would hold on to a specified set amount of observations. This would take place as a drop down that allows the astronomers to pick which previous log sheet they want to view/append or allow them to create a new one. This would allow astronomers to better organize their data into different time frames as opposed to all of the data being dumped into one table.

7.3 Further Integration with Telescope

When the team first visited Lowell Observatory, we were shown quite an array of different softwares that needed to be running at the same time before they were able to successfully make observations with the telescope. It is possible that in the future the client may want to further centralize everything into this one website. By doing so, it could become much easier to understand and set up all of the different processes occurring at once.

7.4 Open-Source Capabilities

The ability to provide open-source capabilities for the project was in mind since the beginning. It was our team's priority to get the software working for Lowell Observatory first and foremost and to make it open source after the fact. The project is mainly focused on our client's needs but creating a proper open-source repository for the project could help expand and test the project. As our client uses the product he may feel the need to further its open-source ability.

8.0 Conclusion

With the completion of our project, many of the problems that our client was facing will be solved. As such, Dr. Llama is able to operate his new spectrograph remotely, making it much easier and much faster to collect the important data he needs to do his research. He no longer needs to spend hours not only setting up the spectrograph, but also saves time by being able to operate from the convenience of his house, or wherever he may be for the night. This outcome includes several key features:

- Streamlined web interface control system:
    - Removes all the unnecessary controls of the interface.
    - Displays real-time status updates from the spectrograph and camera, allowing for quick fixes to any problems that arise.
    - Facilitates the review of past observations through an easy to use log sheet interface.
- Backend API:
    - Allows quick and effective communication between the control system, spectrograph, and database.
    - Grants a reduction in hassle when trying to access data.
- Automated spectrograph:
    - Helps remove the necessity of constant maintenance for normal usage.
- Secure login system:
    - Allows only specific user accounts to access data.

Because this product is open-source and adaptable to any spectrograph and camera combination that implements our interface, it is not specific to the use case of Lowell and may be adopted by other observatories. All of Empyrean wishes for this software to be developed and iterated on so that it can help continue the incredible scientific discoveries of observatories around the country. Although there were challenges along the way, our team endured and managed to produce a comprehensive solution to the problems that we were presented with that we are all incredibly proud of. The Computer Science Capstone Program at NAU helped us take the education of our numerous classes and apply them to a real world experience.

References

[1] Solar System Nasa. 2019. First Image of a Black Hole. Retrieved from https://solarsystem.nasa.gov/resources/2319/first-image-of-a-black-hole/.

[2] Lowell Observatory. 2022. Lowell Observatory: About Us. Retrieved from https://lowell.edu/discover/about-us/.

[3] Go Astronomy. 2022. Go Astronomy: Observatories. Retrieved from https://www.go-astronomy.com/observatories.htm.

[4] Lowell Observatory. 2017. Joe Llama: Research. Retrieved from http://www2.lowell.edu/users/joellama//#research.

[5] Lowell Observatory. 2022. Lowell Observatory: Dr. Joe Llama. Retrieved from https://lowell.edu/people/joe-llama/.

[6] Google Trend. 2022. Habitable Planets. Retrieved from https://trends.google.com/trends/explore?date=all&geo=US&q=habitable%20planets.

[7] Jet Propulsion Laboratory. 2013. In the Zone: How Scientists Search for Habitable Planets. Retrieved from https://www.jpl.nasa.gov/news/in-the-zone-how-scientists-search-for-habitable-planets

[8] Merriam-Webster. Spectrograph Definition & Meaning. Retrieved from https://www.merriam-webster.com/dictionary/spectrograph

Appendix A: Development Environment and Toolchain

The purpose of this section is to get new developers up to speed and start developing as quickly as possible after getting an understanding for the project and the motivations behind its development. It will start with a description of used hardware, then, an argument will be made for each of the essential pieces of software in the toolchain. Next, will be an instruction set for developers to quickly install this project for development. Finally, the same set of instructions will be given in order to deploy this application for production.

A.1: Hardware

This application is not the most resource intensive. The React frontend is the most intensive portion of the application, but it is still not greatly impactful. Because of this, we do not believe that there are any minimum hardware requirements for development. Likewise, as long as the target production machine is not decades old, there should not be problems in deployment. Saving exposures can be memory intensive, but it is assumed that whatever machine that will be processing these exposures will have had the ability to process exposures in the system used before this application. Just to be safe, make sure the target machine has at least (size of average exposure in Mb * 10) Mb of memory.

For both development, this application has been built on Windows(through Linux subsystem), Mac, and Linux systems. Thus any will be fine for developers. Originally, this application was made to deploy on Mac systems, so all commands will be geared towards mac users. The commands for users using Linux systems should not be that different, as for the most part ‘brew’ can be switched with ‘sudo apt-get’ (or ‘sudo <package-manager>’) for the same results.

A.2: Toolchain

This section will describe each of the tools used in the development of this application, as well as some of the tools used in sending this app into production. We will also provide rationale as to why these tools are necessary.

General Text Editor/IDE

Important For: Development

This is the most obvious, but a generalized text editor will be necessary for development. Because of the multi-language project, with major portions of code being written in both Python and Javascript, we cannot recommend a specific editor, such as PyCharm. Many of us used VSCode for our development, but we also had success using NeoVim and Emacs for general development. We also recommend a light text editor at least on production machines, for smaller edits that may need to be made, such as Nano, Vim, Emacs, or any other common lightweight text editor.

Python and Its Dependencies

Important For: Development and Production

This is necessary for development and production. The necessary dependencies are python,

where 3.9 is the python version used for this project, and pip, which is a python package manager.

Homebrew

Important For: Development and Production
This is the main package manager for Mac devices. We need this in order to manage many of the packages we use below.

Git

Important For: Development and Production
This is a standard tool for software development, and is used in version management for the app. Every workflow in using this app will involve git.

Conda or Another Python Version Manager

Important For: Development and Production
This is also necessary, mostly for development, but also for production. This will allow a system to run several different python projects without version mismanagement. We use Conda in the following sections, but there are other options as well, including venv.

Node and NPM

Important For: Development and Production
Similar to Python, this is necessary for being able to run the project. Node allows for the development and compilation of Javascript, and npm is the package manager like pip is for python.

Nginx

Important For: Production
This is necessary to serve this application. When set up, Nginx will be able to serve the react application as well as route unknown requests to the flask api.

LaunchD

Important For: Production
This is also necessary to serve the app. This tool, which comes pre-installed on mac devices, allows for services to be set up. These allow a user to make a device, on start-up, run a program. This will allow us to run the flask server, as well as the camera and spectrograph clients.

PostgresQL

Important For: Development and Production
This is our database system. It is widely used and open source, making it an easy choice for this project.

A.3: Setup

By the end of this section, a new developer will be able to develop new code for this shelyak-control app. The following steps are designed for a developer with novice to moderate experience. These steps are geared towards Mac development, but similar commands can be

found for Linux, and more information can be found inside the documentation folder of the main repository. The prerequisites are that python, pip, and conda are installed.

1. Download Code:
   We recommend creating a new repository for this code. Then run the following inside that directory:

   ```
   $ git clone https://github.com/Empyrean-Capstone/Empyrean.git
   $ cd Empyrean
   ```

2. Install npm, postgres:
   Ensure that the following are installed:

   ```
   $ brew install npm
   $ brew install postgresql
   ```

3. Make a python virtual environment
   We need a virtual environment so that we can install packages without version management. This is so the installation does not ruin other packages that have been installed. The environment can be created and activated with the following:

   ```
   $ conda create --name {name of environment, i.e. empyrean} python=3.9
   $ conda activate {name chosen earlier}
   ```

4. Install python requirements:
   We need to move into the directory in which the file requirements.txt is, then we can install the requirements with the following:

   ```
   $ pip install -r requirements.txt
   ```

5. Configure python variables:
   In the directory main, there is a .env file. Move into this directory, and open this file, changing the variable for DATA_FILEPATH to the location where FITS file should be saved. The default is the directory created in step one. Remember this file, as we will be coming back to make one more change to this file.
6. Install npm requirements:
   Move back to the root directory of the project and install the node packages needed for react by running:

   ```
   $ cd ../../
   $ npm install
   ```

7. Configure database:
   To get data, we will need to set up postgresql so that our app can access a database. Open postgresql, create a new user with the ability to create databases, and create a database with:

```
$ psql -U postgres
        > CREATE USER empyrean WITH PASSWORD '{chosen password}' CREATEDB;

        > CREATE DATABASE {name of database};

        > \q
```

Next we need to edit the .env file from before, changing the line SQLALCHEMY_DATABASE_URI="postgresql://{chosen_username}:{chosen_password}@localhost/{created_database}

8. Migrate database:
   We have an empty database, but we still need to generate a schema for this database. Move to the api directory, then upgrade the database with the following:

```
$ cd /path/to/project/root/Empyrean/api
$ flask db upgrade head
```

   Then, a default database can be migrated from the root directory with the following command. This will load the database with a default user with username 'root', and password 'password'. Expect to change these as soon as possible:

```
$ psql {created database} < ../documentation/base_database.sql
```

9. Run the project:
   The project can now be run. Two terminals will be needed as one server will serve React, while one will serve the Flask application. These can be run with:

```
# The python project can be run with:
$ python api/wsgi.py

// The React project can be run with:
$ npm start
```

10. Installation of Instruments:
    The instruments are run using the same environment as the main application. The prerequisite is that the camera is able to be used already with the device being developed on, and that the camera works with ZWOasi.
    a. Install the code for the instruments in the root directory of the project:

```
$ git clone https://github.com/Empyrean-Capstone/shelyak_control.git
```

    b. Each of the instruments can be run in separate terminals with:

```
$ python spectrograph.py
```

```
$ python camera.py
```

A.4: Production Cycle

This documentation will ensure that a developer will be able to install this app on a production machine. The requirements are the same as for A.3.

1. Follow the steps from A.3 to install the application in a development environment.
2. Install nginx:
   Install with:

   ```
   $ brew install nginx
   ```

3. Configure nginx:
   This will make sure we can serve our files from React to a user. We first need to build our project with:

   ```
   $ npm run build
   ```

   Make note of the absolute path of the build path (path/to/project/Empyrean/build). Make sure that this location is accessible to any user. Find the location of the configuration files for nginx with:

   ```
   $ nginx -t
   ```

   Open this file in a text editor, and add the configuration found in Appendix B, making sure to change the location of the ‘root’.
   After this, we will start nginx as a service with:

   ```
   $ sudo brew services start nginx
   ```

4. Start Flask server as a service:
   We already were able to run the flask server, but the server will stop whenever the computer loses power. Starting it as a service means that the server will start back up after power is lost.
   We need to add this process to the service list. First, make not of where the python we are running is located, then get to the directory where these services are stored, where we can then create our service with:

   ```
   $ which python
   $ cd /Library/LaunchDaemons/
   $ touch com.empyrean.plist
   ```

The services for each of the instruments can also be made with:

```
$ touch com.empyrean.camera.plist
$ touch com.empyrean.spectrograph.plist
```

Copy into each of these files the contents of each file found in Appendix C, making sure to change the two strings inside of the ProgramArguments key, as well as the filepath to the error and standard output lines to somewhere convenient.

5. Finally, restart the machine, and make sure everything is working. It may be though, that the camera and spectrograph will need to be reloaded, which can be done with:

```
$ sudo launchctl unload  /System/Library/LaunchDaemons/com.empyrean.plist
$ sudo launchctl load  /System/Library/LaunchDaemons/com.empyrean.plist
```

Appendix B: nginx.conf Configuration

Below is a default configuration for nginx.conf. It provides nginx with all of the information it needs in order to serve the react server as well as proxy requests that the react server does not recognize to the flask server.

```
worker_processes  1;

events {
        worker_connections  1024;
}


http {
        include           mime.types;
        default_type  application/octet-stream;
        sendfile          on;
        keepalive_timeout  65;

        server {
                # Port listened on, this allows for outside access to the site.
                listen    80;
                server_name  localhost;


        location / {
                  # Location to the build directory
                  root   /Users/naucapstone/empyrean/Empyrean/build;
                  # Name of the generated file in this build directory
                  index  index.html index.htm;
                  try_files $uri /index.html;
        }
        location /api/ {
                proxy_set_header Host $http_host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                proxy_set_header X-Forwarded-Proto $scheme;
                proxy_pass http://127.0.0.1:5000;
        }
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
        root   html;
        }

        include servers/*;
}
```

Appendix C: Example Launchctl Service Configuration

Below are example configurations to make the flask server, camera and spectrograph work as services on mac devices, using the launchctl that comes with macs. Some things to note. The first string in the ProgramArguments array is the path to the python being used. This can be found by switching to the desired environment, and running the command “$ which python” (without the quotes). The second string is the absolute path to the program we are trying to run. For more reading, this launchtl introduction is a good place to start.
https://www.launchd.info/
For creating services on linux, here is a similar resource:
https://www.freedesktop.org/software/systemd/man/systemd.service.html

Here is an example of the flask configuration, with the filename com.empyrean.plist:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
        <dict>
        <key>Label</key>
        <!--Name of plist-->
        <string>com.empyrean</string>
        <key>ProgramArguments</key>
        <array>
        <!--Location of python file-->
        <string>/Users/naucapstone/anaconda3/envs/empyrean/bin/python</string>
        <!--Location of the python file to be run.-->
        <string>/Users/naucapstone/empyrean/Empyrean/api/wsgi.py</string>
        </array>
        <key>KeepAlive</key>
        <true/>
        <key>StandardErrorPath</key>
        <string>/Users/naucapstone/empyrean/errorlog.txt</string>
        <key>StandardOutPath</key>
        <string>/Users/naucapstone/empyrean/standardlog.txt</string>
        </dict>
</plist>
```

Here is an example of the similar camera configuration, named com.empyrean.camera.plist:

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
        <dict>
        <key>Label</key>
        <string>com.empyrean.camera</string>
```

```
        <key>ProgramArguments</key>
        <array>
        <string>/Users/naucapstone/anaconda3/envs/empyrean/bin/python</string>
        <string>/Users/naucapstone/empyrean/shelyak_control/camera.py</string>
        </array>
        <key>KeepAlive</key>
        <true/>
        <key>StandardErrorPath</key>
        <string>/Users/naucapstone/empyrean/camera.errorlog.txt</string>
        <key>StandardOutPath</key>
        <string>/Users/naucapstone/empyrean/camera.standardlog.txt</string>
        </dict>
</plist>
```

Finally, here is an example spectrograph configuration, named comempyrean.spectrograph.plist:

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
        <dict>
        <key>Label</key>
        <string>com.empyrean.spectrograph</string>
        <key>ProgramArguments</key>
        <array>
        <string>/Users/naucapstone/anaconda3/envs/empyrean/bin/python</string>
        <string>/Users/naucapstone/empyrean/shelyak_control/spectrograph.py</string>
        </array>
        <key>KeepAlive</key>
        <true/>
        <key>StandardErrorPath</key>
        <string>/Users/naucapstone/empyrean/spectrograph.errorlog.txt</string>
        <key>StandardOutPath</key>
        <string>/Users/naucapstone/empyrean/spectrograph.standardlog.txt</string>
        </dict>
</plist>
```

Appendix D: Modifying Database Schema

The database schema as is is believed to be nearly complete, but there may be instances where the schema must be modified in order to accommodate unseen processes that must be performed. The following is a list of steps to perform in order to update the schema of the database for future changes.

D.1: Using Alembic to Create Model Changes:

1. Creating a new migration file: In the api directory of the project (api/), run the following:

```
$ alembic revision -m “Name of Migration”
```

This will create a new file in the migrations directory inside, which can be used to make the database schema modification.

2. We can now edit this file. Here is a basic layout that only needs modified, which would add a column to a table.:

```
# revision identifiers, used by Alembic.
revision = 'ae1027a6acf'
down_revision = '1975ea83b712'

from alembic import op
import sqlalchemy as sa

def upgrade():
    op.add_column(‘<table_name>', sa.Column('<column_name>',
sa.<database_type>))

def downgrade():
            op.drop_column(‘<table_name>, '<column_name>')
```

3. Finally, we can run this migration with:

```
$ alembic upgrade head
```

This will add this new column to the database.

4. For more reading, alembic has very good documentation. The reading this is based on was found here: https://alembic.sqlalchemy.org/en/latest/tutorial.html#running-our-second-migration

Appendix E: Creating New Flask Blueprints

In development, it will likely be necessary to add new groups of api endpoints for the Flask database. The standard for this process is a new Flask Blueprint, which can be thought of as classes that will accept different requests made by a frontend or instrument. The following instructions can be used in order to add new Blueprints in a way that the application will recognize. There is also liberal documentation on this topic, however no style convention. Thus, this guide will represent Empyrean's style guide which should be followed for future development.

Creating a new Blueprint:

1. Create a new directory within the main directory of the api. This name should be snake case, and reflect the name of the desired blueprint. As an example, from the root directory of the project:

```
$ mkdir api/main/new_blueprint
```

2. While in this newly created directory, add two files, "__init__.py" and "views.py". The __init__ file allows Flask to be able to see these new routes, while the views file are where these new routes will be placed.
3. First for the __init__ file, open this file to be edited, and copy the following. The example should be replaced with the name of the blueprint, angina in snake case:

```
from flask import Blueprint
example = Blueprint( 'example', __name__ )
from . import views # imports all of the routes from views
```

4. In views.py, here might be some example code. The only required line is the first one, which imports the blueprint created in the __init__ file:

```
from . import example

@example.route('/')
def index():
    # Code here
    return ""

@example.route('/get')
def get():
    # Code here
    return ""
```

5. Finally, we must add this new blueprint to the main flask app initialization so that on reboot the blueprint is registered and its routes can be accessed:

```
from .example import example
app.register_blueprint( example, url_prefix="api/example" )
```

Now, any new routes in this blueprint will be registered, and can be found at /api/example/new_route.