TEAM SKYWARD

Frost Monitor Project

Final Report

5/4/2017

Team Name:

Skyward

Team Members:

Gage Cottrell
Justin Kincaid
Chris French
Alexander Sears

Sponsors:

Dr. Michael Mommert and Dr. David Trilling

Mentor:

Dr. Otte

Table of Contents

Introduction

Document Purpose

The purpose of this document is to outline the technical aspects of the FRoST Monitor System and the technologies used to develop and implement the application. The goal of this document is to give the reader a better understanding of how the application is being developed and implemented through examples of requirements, constraints, and system architecture.

FRoST Monitor Project

We are Team Skyward and we are building an online dashboard for the FRoST Monitor. The FRoST monitor is a telescope dome that is a few miles south of the Northern Arizona campus. The FRoST monitor dashboard will display vital telescope and weather information quickly and easily. Weather information gathered from the FRoST monitor and displayed on the dashboard will include data such as current temperature, dew point temperature, wind, and the latest image from the all-sky-cam. All of this data is supplementary to help our clients use the FRoST monitor telescope to photograph and study Near Earth Objects.

Near Earth Objects (NEOs)

Everyday about one hundred tons of particles originating from outside of Earth's atmosphere makes contact with Earth's surface. While the vast majority of the particles contacting Earth are very small, instances of larger objects impacting the surface have been recorded. Roughly every 10,000 years, a rocky or iron asteroid larger than 100 meters impacts the earth and causes localized destruction. On an average of every few 100,000 years, asteroids larger than 1 kilometer impact the earth and cause global disasters that can severely impact life on Earth. Due to the risk of having Earth impacted by objects that have the potential to cause massive destruction, it is vital that near earth objects be discovered, tracked, and analyzed.

Researching Near Earth Objects (NEOs) is an ongoing joint effort between many countries and governments. Using telescopes, it is possible to discover and track NEOs as they move throughout the solar system. Making publically available the information gained from NEO discoveries, allows for a collaborative environment in which NEOs can be discovered, tracked, and analyzed very quickly. To date, there have been more than 15,000 different NEOs discovered. As telescope technology advances, so do the amount of NEOs discovered. Figure 1 below depicts the discovery rate of near-earth-asteroids from 1980 to present date.

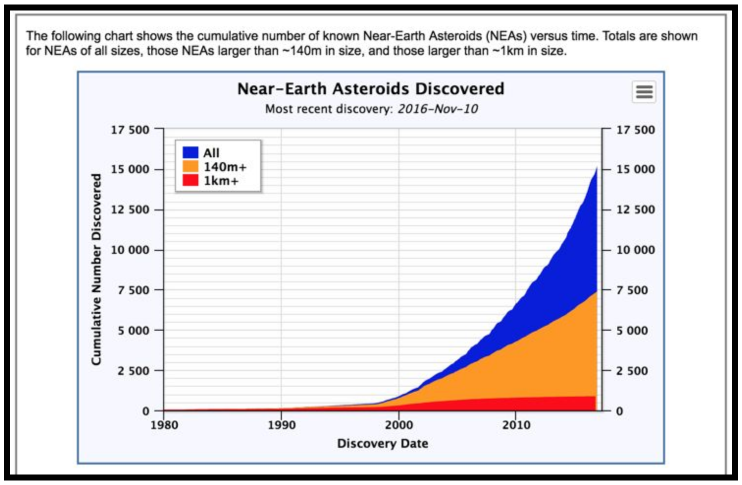

The following chart shows the cumulative number of known Near-Earth Asteroids (NEAs) versus time. Totals are shown for NEAs of all sizes, those NEAs larger than ~140m in size, and those larger than ~1km in size.

Figure 1: Near-Earth Asteroids Discovered

Nuthatch Server

"Nuthatch" is a server built and maintained by our clients. Our team will not be responsible in any way for the development or maintenance of the Nuthatch server. This server will gather and host weather and telescope information. Our team will then take this information and use it to populate all of the required data to display on our web application.

Team-Process Overview

For our project, we used only a few tools. Our team is not big and our project is not very complicated, so it was sufficient to use some simple tools. For version control, we used git and git accessories like Bitbucket, which hosted our repository. As for CASE tools, we did not use any. The only software design help we used were a few pieces of paper and a whiteboard. We used Discord to record our tasks and for communication between team members. It was on Discord that we posted our agendas, our to-do lists and our notes from various meetings.

We shuffled our roles around a lot, we did not have static roles at all, except for the team lead. Originally everyone decided what jobs they wanted to do, but by the end we all ended up working on anything we could when it was available. We all programmed both frontend and backend, as well as writing the documents. The team lead was the main mode of communication with the sponsor, although sometimes other group members sent communication as well.

We had weekly meetings with each other in the spring semester, and we used that time to figure out problems we were having with the programming, as well as any difficulties with reports. In between the meetings we were usually very productive, and everyone managed to stay on schedule and complete all assigned work.

Requirements

Project Requirements

The first requirement is a web application that will host display all of the weather and status information that is gathered from the telescope. The second is including a password protected user account for functionality and information that will be hosted behind a layer of security. All other requirements involve communication between the FRoST monitor web application and our clients server Nuthatch in order to get all of the necessary data to display on the web application.

Brief project requirements list includes:

- Web Application
    - Access to our clients Nuthatch server

    - Database to store mirror of data hosted on Nuthatch server
    - Ability to update data to web application in real time
    - Maintain a list of issued commands to the telescope
- Single Password Protected User
    - Dome shutdown button

Implementation Overview

The FRoST Monitor system will be built to give researchers and users of FRoST the ease of being able to quickly and efficiently check on the live status of the telescope and surrounding weather conditions. The FRoST Monitor System will be built using the Django web development framework - and will be using the Django REST Framework as a means to receive current data. The entire FRoST Monitor System will be created within a virtual environment using Anaconda, to ensure that our application can easily be modified without worrying about other outside factors.

Using Anaconda, we will install a virtual environment on our clients server so that we may make any changes we need in order for Django to run the way we want it, just so these changes aren't made for the whole machine. We aren't the only ones using it since it belongs to the whole NAU physics department, so it's best for us to use a virtual environment in order to keep our footprint on the system to a minimum. The machine will have to be configured to direct certain traffic to this environment in order to provide access to the FRoST Monitor system.

The Django REST Framework will be used to communicate with the client's own data processing machine. That machine will have a script that sends data to the web server which, since the data will be available locally, will make it quick and easy to display data on the web page.

The Django web development framework utilizes a Model View Controller (MVC) architecture that is essentially a 3-tier architecture. This document will later discuss in detail this architecture and its strengths and weaknesses.

The webpage that is produced with the Django framework will update the information being displayed automatically every 60 seconds using asynchronous JavaScript (AJAX) calls. Implementing AJAX into our Django application allows the application to automatically update the data being displayed, which is extremely convenient for users who need to view the latest data available.

The following section of this document will discuss in detail the architecture behind the FRoST Monitor System.

Architectural Overview

Application overview

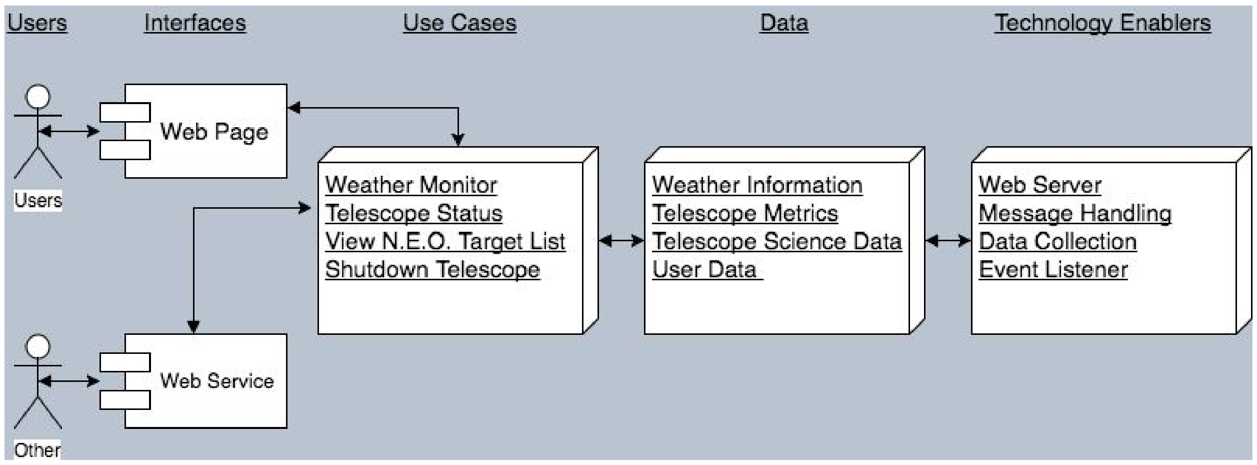


Figure 1: Application Overview and Diagram

User and delivery

The view in Figure 1 allows for different audiences to access the frost monitor through different channels. These two channels are:

- Researches and the public users will access application over the internet using a web browser. These are the main users of the FRoST Monitor System.
- Other users such as the web service requester will access the application through a web service implemented in a ReSTful service. Essentially these "other users" are applications/systems that access our application via the REST service to get/post data such as Nuthatch.

Core application uses

Figure 1 highlights the use-cases for the Frost Monitor application. These uses are the expectations of information to be presented to the user.

- Monitor Weather
    - The user will be able to monitor current weather conditions around and surrounding the FRoST telescope.

- Retrieve Telescope Status
  - The user will be able to quickly observe the current telescope status. An example is where the telescope is currently pointing in the sky.
- View N.E.O list and information
  - The user will be able to retrieve information on the current and upcoming Near Earth Objects that are to be observed by the telescope.
- Emergency Shutdown
  - An admin user will be able to login and send a signal to the telescope. This signal will trigger a mechanical response to the telescope to shut the dome.

Data

The core information and data constructs required to realize the core application uses are highlighted in Figure 1. The follow data is fundamental to the core use cases.

- Weather
  - The current weather data will be the current weather conditions of surrounding the FRoST telescope. This system will manage the propagation of weather data from web services and local hardware.
- Telescope Metrics
  - Telescope Metrics represent the physical status of the FRoST telescope. Examples are the current telescope position, and dome status.
- Telescope Observation Data
  - The telescope observational data represents the past and current observed data from the FRoST telescope. Examples of the data will include a list of N.E.O that are queued to be observed and pictures and telemetry data of past observations.
- User
  - The user data will allow for permission based action between the web application as well as the web service. There will be a user to allow for web service data push's. There will also be a user for access to the emergency shutdown.

Technology Enablers

Figure 1 highlights the key set of components to support implementation of the Frost Monitor application.

- Web Server
  - The application will reside on a web server. The web server will be required for providing access to the FRoST monitor application interface and the REST api.

- Message Handling
    - Will manage request over http from the event listeners. The handler will provide the implementation of client request for information and web service request.
- Data Collection
    - The data collection will be implemented on a gateway machine that will propagate weather and telescope data from the telescope and other external sources. This module will also be responsible for sending data to the web database through the REST api.
- Event Listeners
    - This module is the main function of the REST api. It will handle incoming events and request made over http from clients and web service users. It will then hand off the messages to the message handler.

The FRoST Monitor application is built predominantly upon the Django framework. The Django framework uses a 3-tier architecture very similar to the popular Model, View, Controller (MVC) architecture. Figure 2 can be seen below and is a diagram of the implemented framework in respect to the web application. In Figure 2 below you can see that a Django project can have multiple applications within itself, and that the FRoST Monitor System is using two applications in tandem.

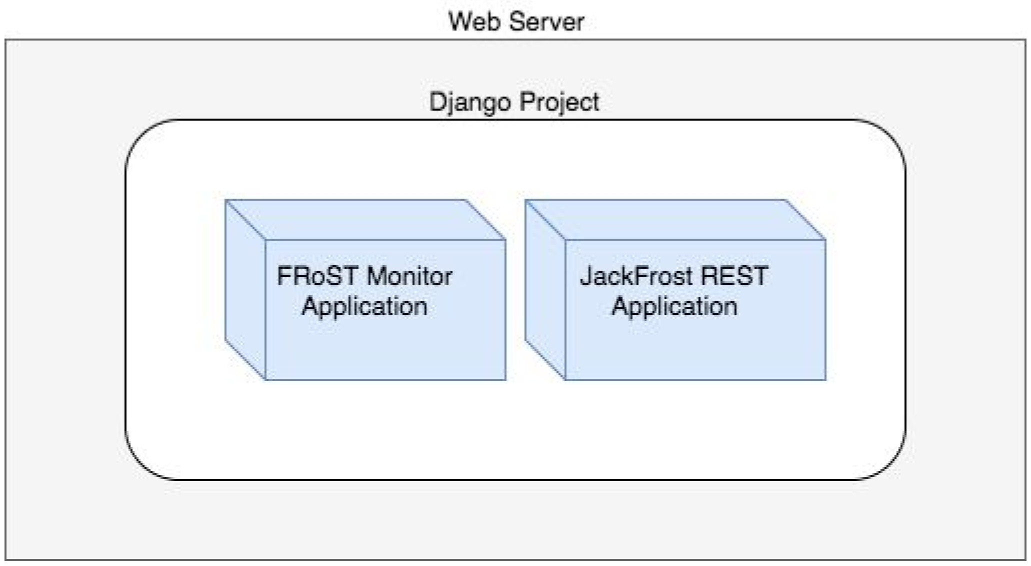


Figure 2: Web Application Layers Diagram

In Figure 2 above you see that there are two applications within the single Django Project. These two applications work together to accomplish the goals of the FRoST Monitor System. The FRoST Monitor Application is the application that is built upon the standard Django framework and is modeled after an MVC architecture. The JackFrost REST Application is the application that is built upon the Django REST Framework, and follows a REST architecture. Keeping the standard Django application separate from the REST application allows us to keep a separation of concerns within our Django project, and will allow for easy modification in the future. Both of our applications live within the same Django project, and that Django project is hosted on a web server so that it is available online.

Figure 3 below is a diagram of the FRoST Monitor and Rest application layers. In the diagram below you can see that there are three main layers: Models, Views, and Controllers. Inside the Models you can see that there are the actual data objects that our application uses to store and display, as well as the REST Serializers that are able to serialize our data into JSON and make it easily consumable by the REST api. The Controller layer handles the GET/POST requests made by users and our application to send and receive data. The View layer is the piece of the application that allows our webpage to be created and displayed with data from the models.

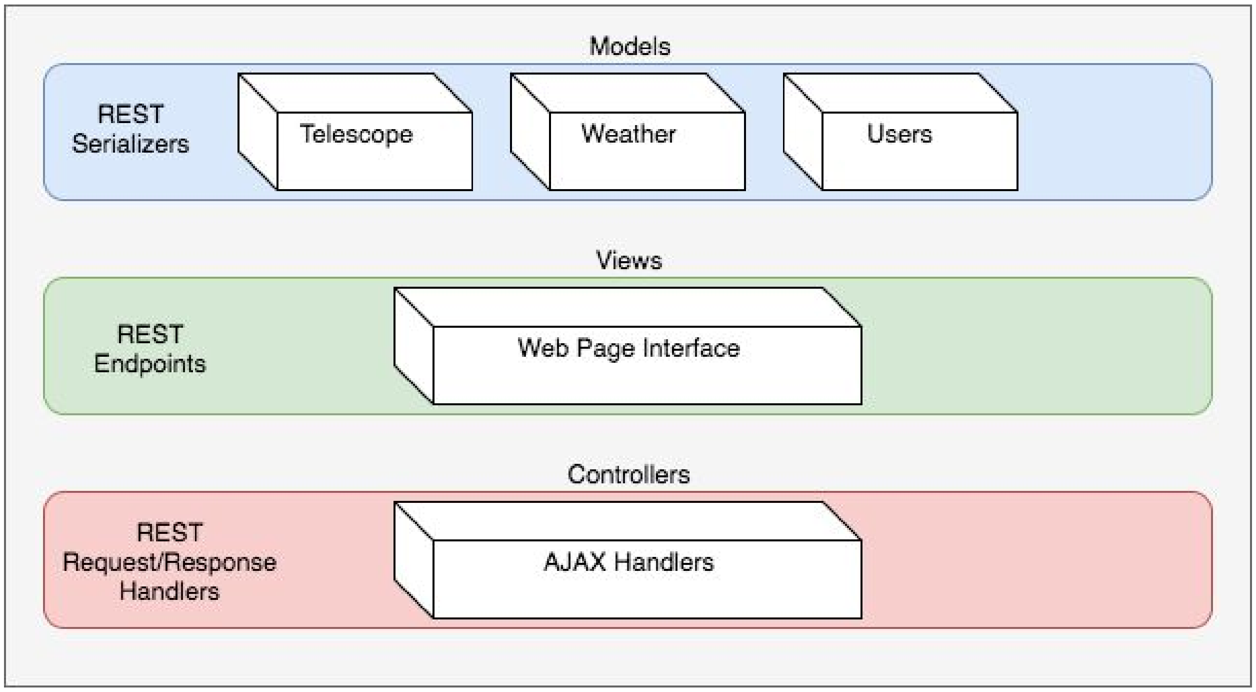


Figure 3: Layered View of FRoST Monitor Architecture

The FRoST Monitor application will use a fusion of two architectures Model, View, Controller architecture, and RESTful architecture. Inside our Django project you may recall that we have created two separate applications. The MVC will be used in the implementation of the web application interface and the RESTful architecture will be used for the jack frost application, which will handle http request to api endpoints including ajax request from the frost monitor web application and external request from data collecting gateway machines. An overview of how the RESTful architecture works with our standard Django MVC architecture can be seen below in Figure 4.

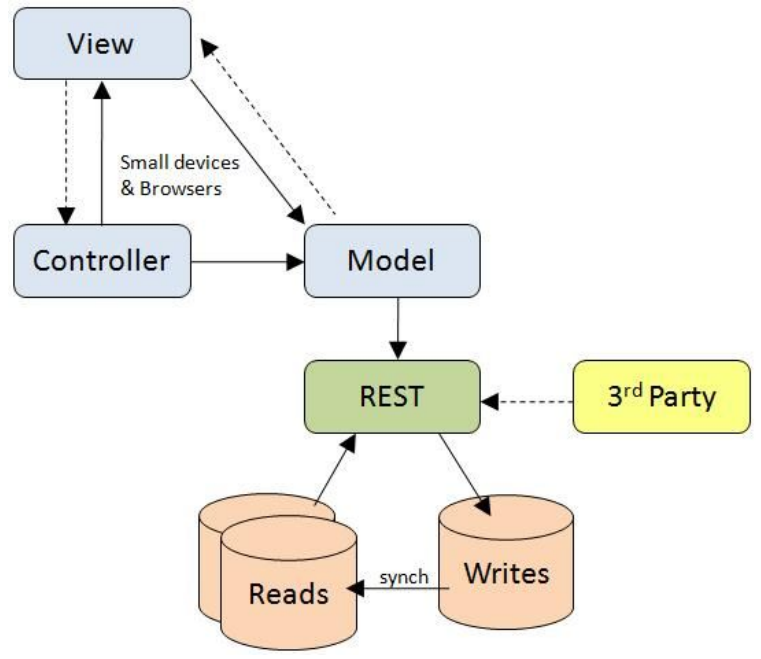


Figure 4: REST & MVC Architecture Diagram

Module and Interface Description

Django

MVC framework vs Django MTV framework.

Our project will be using django at its core. This allows us to set up the website using a commonly used MVC framework. The MVC framework consists of 3 main components; Model, View, and Controller. The model portion of the framework consists of all of the classes that we will need for the project. The View is basically what will appear on the webpage. The controller is what links the Model and the View together.

In Django the underlying MVC architecture is actually slightly different from the classic MVC approach. Models are still Models in Django, but a View is actually called a Template, and a Controller is actually called a View. This means that a Django Template is actually what you see on the webpage, and a View links the classes in a Model together with a Template.

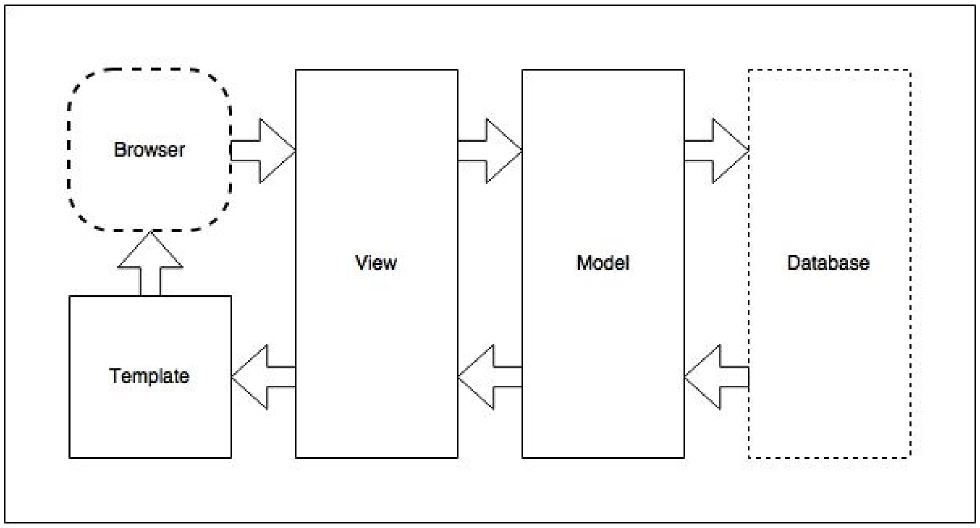


Figure 6.1: Django MTV Diagram

Flow of data in a Django MTV framework.

Figure 6 shown above showcases a brief overview of the flow of data in a basic Django application, beginning with a request from a browser and resulting in a web page produced back to the browser.

When a request is made to view a webpage provided by a Django application, it is first referenced in a list of url patterns located in a file "url.py". The url patterns in this file will link directly to the View portion of the MTV framework by accessing a file called "views.py"

The file "view.py" basically holds all of the functionality for the Django application (which explains why we reference these as "controllers"), and uses the classes defined in your Model to manipulate the data before sending the data to a template.

The Model keeps all of the models in a file labelled "models.py". Once a class is defined in this file, any objects created from each class will automatically be added to an SQLite database that is maintained inside the Django app. The requested data from the database will then be returned back to the View, and then returned to template.

Templates are used to dictate how the processed data will look on a webpage after it has been requested. A template consists of all of the basic utilities that can be included in any html document. Each page in a Django project will require its own template.

Urls.py

The url patterns in the "urls.py" file include "index", "admin", "login_admin", and "ajax". Each url pattern sends a request to the View which calls a function by the same name in the "views.py" file. For example, the url pattern for "index" uses a line of code called "views.index", and will call the function "index", from the file "views.py". This means that "views.py" will only consist of the 4 functions listed above. This "urls.py" file handles the url routing for the django application.

Models.py

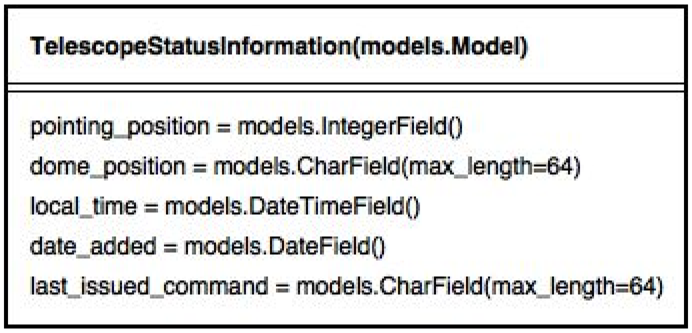
In Django, database tables are created via python classes. These classes are individually referenced as a "model" and all together we call our database entries the "Models". In the FRoST Monitor System we currently have three models which are the Telescope Data, Weather Data, and the User Data.

Telescope Status Class

The fields for the Telescope Data are "pointing_position", "dome_position", "local_time", "date_added", and "last_issued_command". This model is displayed within our template so that our users can view the telescope data at anytime.

The "last_issued_command" field will be useful in the case of any disruptions in communication between the application and the dome, as well as any unforeseen failures, because it will be possible to see what the telescope was doing during a point of failure. The "date_added" field gets stored along with all of the other information in the class when a new instance of "TelescopeStatusInformation" is created in the database. This allows the View to request the most recent instance of the class in the database, or request any amount of instances, and order them by date, which would be handy for building a chart that tracks status over time. For example, it would be possible to view the last issued command given to the telescope, or even view a list of the last 30 commands on a specified date range, by using both "last_issued_command", and "date_added" fields.

The "dome_position" and "pointing_position" fields are important entries for the class. These fields allow us to trace the facing direction of both the telescope and the dome, as well as compare the orientation of each with each other. A mismatch in alignment between the telescope and the dome would result in black images taken by the telescope, because it is actually photographing the inside of the dome rather than the night sky.



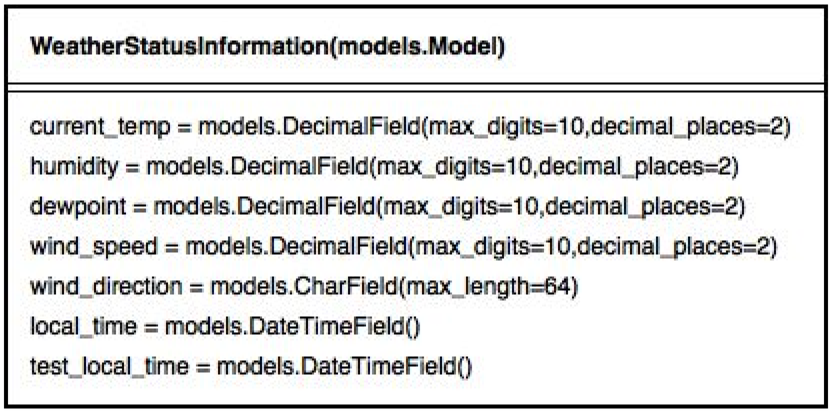
```
TelescopeStatusInformation(models.Model)

pointing_position = models.IntegerField()
dome_position = models.CharField(max_length=64)
local_time = models.DateTimeField()
date_added = models.DateField()
last_issued_command = models.CharField(max_length=64)
```

Figure 6.2: Telescope Status Class UML

Weather Status Class

The Weather Data model is made up of "current_temp", "humidity", "dewpoint", "wind_speed", "wind_direction", and "local_time". These are the attributes of our weather model that allows users to view important weather information within our application.

As mentioned above, the time and date fields will allow whether information to be requested and displayed across a wide range of time. This will allow us to display the weather data in different ways. For example, a graph could be constructed that would show the temperature over the course of the previous 3 days, or all of the temperature readings could be gathered in the range of an entire day to produce a high and low temperature for that day.

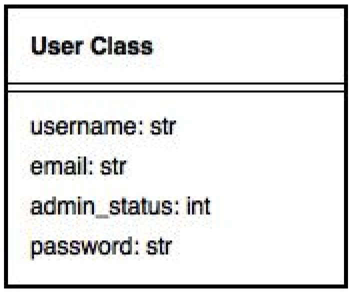


```
WeatherStatusInformation(models.Model)

current_temp = models.DecimalField(max_digits=10,decimal_places=2)
humidity = models.DecimalField(max_digits=10,decimal_places=2)
dewpoint = models.DecimalField(max_digits=10,decimal_places=2)
wind_speed = models.DecimalField(max_digits=10,decimal_places=2)
wind_direction = models.CharField(max_length=64)
local_time = models.DateTimeField()
test_local_time = models.DateTimeField()
```

Figure 6.3: Weather Status Class UML

User Class

The User model is actually a model that is built in to Django and does not require setup or explicit definition. The default fields for a user in Django are "username", "email", "admin_status", "password". These fields are all we need for our user models so we intend on keeping them the way they currently are.



```
User Class

username: str
email: str
admin_status: int
password: str
```

Figure 6.4: User Class UML

Views.py

"Views.py" will consist of 4 functions; "index", "admin", "login_admin", and "ajax". These functions return data to specific web pages within our web application. Each of the functions in the View are referenced by the “urls.py” file, and are called after being requested by the corresponding URL address in the Django application.

The View is not responsible for how the data is displayed, but rather what data will be displayed. The View is the section in which all of the functionality for the web application will reside. It is responsible for requesting queries from the database, and then manipulating and organizing the data before passing it off to a template for displaying to a bowser. For example, the request for weather information in a date range to be displayed in graph for would start in the “views.py” file.

Index
The index function currently generates the most recent weather data to display on the index page. This will eventually be expanded to include extra data such as the latest all-sky-cam image, last given telescope command, and any other pieces of data that would be relevant to have on the front page.

Login_Admin
This function simply calls “login.html” template which requires no additional data from the database. This login web page allows users to input a username and password.

Admin
This function checks to see it the correct login information is given. If the login information does not match the information saved in the database, then the user is referred back to the previous login screen. If the information provided is correct, then the user is referred to the “admin.html” page, which will contain the emergency shutdown button for the telescope.

Ajax
The purpose of the “ajax” function is to gather the most recent weather information. This information will be updated periodically and reflected on the webpage without the need for a page refresh by referencing the “ajax” output in a template.

Templates
The templates used in the Django application are typical of any webpage in that they can be written in HTML, and use tools to augment the page such as CSS and JavaScript. The fields in the Model classes are called and manipulated by the View which results in a query of instances from the Model classes. These instances can then be utilized in a template by referencing the returned results from the View. This makes the content displayed in the webpage to be dynamic.

In a Django application, the html content you view is not always static, which is great for modern dynamic web applications. Templates can be fed dynamic queries that are requested by the View before the pages are actually displayed to the user. For example, every time a user visits the “index” template within our project, Django’s templating system will go and fetch the latest database information to be loaded into our template. This is a very powerful tool, because this allows our index page to always display the latest data from the database, without the need for a POST request or a page refresh.

We intend on continually leveraging the template system to make our application as useful as possible.

Testing Overview

Project Background & Motivation

We are developing an online dashboard which displays vital information and metrics for the Flagstaff Robotic Survey Telescope (FRoST). The purpose of FRoST is to carry out follow up research on near-earth objects (NEOs) so that researchers can better analyse and track NEOs. With more than 15,000 already discovered NEOs, there is a great need for research to be done on the already discovered objects - most importantly so that we might be able to predict impacts with these objects and Earth.

Our online dashboard, known as the Frost Monitor System, displays various information about FRoST and weather local to the telescope. Using our the Frost Monitor System, researchers and operators of FRoST will be able to quickly see information that would previously require many steps to reach. Therefore, the main motivation for developing the Frost Monitor System is that it will be able to save researchers time when looking for the information that FRoST works with.

General Software Testing Overview

As with all software being developed, there comes a time when the developers must be sure that the software behaves as it should and that the software meets the requirements that were initially designed before development began. Validating and verifying software is done in many ways, and this process is more generally known as "software testing". Software testing can be done automatically using testing software, or manually by preparing tests of the system and verifying the results of the tests. Many times software testing is carried out by using various automated tests as well as manually creating tests specific to the software being tested. Both automated and manually tests aim to find errors, faults, and failures within the software being tested.

Testing Plan Overview

The Frost Monitor System has been in the development stage for about 90 days, and the system's requirements are nearly all implemented and designed. The next stage of development for the Frost Monitor System is to carry out software testing. Our plans for software testing involve unit testing, module integration testing, as well as usability testing.

We decided to use unit tests to validate the functionality of our software. Due to our application being developed with Django - a python web development framework - we will be able to easily carry out unit tests since python features an already created 'unittests' testing package. Because the Frost Monitor System is comprised of two separate applications, we will be carrying out unit tests within both of these applications separately. Using unit tests, we will be able to verify that individual units of our software are working as per requirements and are error free. Further detail will be given in the following "Unit Tests" section of this document.

Given that the Frost Monitor System is comprised of various modules, validating these various modules work together as they should will be carried out via integration testing. We felt that integration testing was necessary due to the fact that if one module in our system fails, our system may not behave as it was intended. More detail regarding our integration tests can be found later within this document.

The Frost Monitor System is a web-based application that users directly interact with and view when using. Because our application aims to save the user's time, we must carry out usability tests. Usability testing will allow us to make sure that our software is as user-friendly and simple as possible, while still being highly effective. Usability testing involves having actual users test the software by interacting with the system as a whole. By carefully analysing how users interact with our system, we can then make changes to improve upon the system and its functionalities. More details regarding our usability tests can be found within our "usability testing" section further down in the document.

The following section of this document will describe in detail our unit tests for the Frost Monitor System and its applications.

Unit Testing

Testing Approach

We have divided the unit testing into two main sections. The sections are logically divided between the Frost Monitor front end application and the JackFrost back end application - the two applications that comprise the Frost Monitor System. This ensures that each application will be tested according to its own functionality. This will also allow for quick identification of problem areas and will afford for each application to be tested individually without affecting the other.

Each application will be divided into modules of similar functionality. Each module will have test to evaluate its general functionality for all submodules. Each submodule will then have test to evaluate unique critical elements within the submodule.

The unit test will be organized as follows:

1. **Frost Monitor – front end**
   - 1.1. Views
     - 1.1.1. Admin
   - 1.2. Templates
   - 1.3. Urls
2. **JackFrost – back end**
   - 2.1. Views
     - 2.1.1. View sets
   - 2.2. Models
     - 2.2.1. Data Models
     - 2.2.2. Serializers
   - 2.3. Urls
     - 2.3.1. jackfrost/

This top down approach to unit testing the entirety of both applications will provide adequate test coverage while reducing redundancy. Many of the methods and classes within the submodules perform similar task with similar data. Unit test with a low test coverage that still test the entire set of implemented data types,objects, and classes will produce the same results as having tests for each individual method within the submodules.

Both the Frost Monitor (front end) application and JackFrost (back end) application are built upon Django, an open source framework that is actively maintained and heavily tested. A majority of the functionality of both applications are implemented using the built-in objects of the framework. Testing that we instantiated the built-in objects within our application correctly is important, while testing that the built-in objects themselves are implemented error free is unnecessary within the context of this project. This is a guiding principle we used in establishing our unit test cases.

We will be using Django's built in test suite which is built on top of 'unittest' module in the Python standard library. Django's test-execution framework will provide all the necessary utilities to perform unit test for both the front end and back end application. With Django's test framework we can easily perform, simulate and automate complex web application interaction as well as integrate other web based testing software such as Selenium into the test suite. To determine our code coverage we will be using Coverage.py - a testing tool that also integrates within Django's test suite. Coverage.py allows us to visualize where, what and how much code our predetermined test are actually covering. This will help us decide if we need to implement, modify or change test based on the statistics provided by Coverage.py.

Testing Plan

1. FROST MONITOR

1.1 Views.py

The views are responsible for handling HTTP Post and returning the HTML with the rendered request.

- **All views can :**
  - Return rendered HTML
  - Receive HTTP POST
- **Test will:**
  - Assert correctly rendered returns
  - Assert correctly received HTTP POST

The table below shows sample test inputs with their respective outputs when unit testing Views.py within the Frost Monitor System.

Test Inputs	**Test Outputs**
Malformed HTTP POST	Error
Correct HTTP POST	True
No HTTP POST	None
Method call with request	HTTP Status code 200 & HTML doc with correct title
Method call with no request	HTTP Status code 200 & HTML doc with correct title

1.1.1 admin view

The admin view handles the authentication page for admin and super users for the Frost Monitor Application.

- **The admin view can:**
  - Authenticate users
  - Reroute users based on authentication
- **Admin view test will:**
  - Assert only Authenticated user can login

- Assert correct reroute of URL

The table below shows the test inputs with their respective outputs when unit testing Admin.py within the Frost Monitor System.

Test Inputs	**Test Outputs**
Super User Username & Password	True & 302 & correct URL redirect
Super User Username only	False & 302 & correct URL redirect
Super User Password only	False & 302 & correct URL redirect
Unregistered Username & Password	False & 302 & correct URL redirect
Unregistered Username only	False & 302 & correct URL redirect
Unregistered Password only	False & 302 & correct URL redirect

1.2 Templates

The templates are the static HTML pages that can include django tags that are dynamically loaded by the views.py

- **All templates:**
  - Have valid and well-formatted HTML
  - Have valid external and internal links
- **Test will:**
  - Assert correctly formated HTML
  - Assert there are no broken links

The table below shows test inputs with their respective outputs when unit testing template files within the Frost Monitor System.

Test Inputs	**Test Outputs**
Load page from ‘client’	True
Search for broken links	False

1.3 Urls.py

The urls.py is responsible for routing the URLs to the correct views corresponding to those uri.

- **All URLs can :**
    - Return the corresponding view
- **Test will:**
    - Assert received HTTP Response
    - Assert the correct view is returned

The table below shows sample test inputs with their respective outputs when unit testing Urls.py within the Frost Monitor System.

Test Inputs	**Test Outputs**
Malformed HTTP GET	HTTP Status code 418
Correct HTTP GET	HTTP Status code 200 & HTML doc with correct title

2. Jack Frost

2.1 Views.py

The views are responsible for handling HTTP Post and returning the HTML with the rendered request.

- **Refer to 1.1**

2.1.1 ViewSet

ViewSet is a view that is used in the django-rest framework used to display sets of data models in JSON format in an HTML document and is also responsible for initializing the database query for GET request and the Model serializer for POST request.

- **All viewsets can :**
    - Return data from a HTTP GET request
    - Receive data from a HTTP POST request
- **Test will:**
    - Assert correctly returned HTTP GET request
    - Assert correctly received HTTP POST request
- Note:
    - Each viewset can have methods that are unique to their response requirements, as individual response methods are implemented their corresponding test will follow these same guidelines

The table below shows sample test inputs with their respective outputs when unit testing Views.py within the Frost Monitor System (Specifically, JackFrost).

Test Inputs	Test Outputs
Malformed HTTP POST	HTTP Status code 400
Correct HTTP POST	HTTP status code 200
Malformed HTTP GET	HTTP Status code 400
Correct HTTP GET	HTTP Status code 200
ANY restricted HTTP request	HTTP Status code 405

2.2 Models.py

The models are python classes that represent the data used in the Frost Monitor System. These data models can be divided into the following two categories.

2.2.1 Data Models

The Data Models are python classes that represent the data that is gathered from the FRoST. The Model class is a built-in object from Django framework. We will ignore all of the class methods that are not overrode in the child class and focus on correct parameterization while instantiating class variables.

- **All models.py can :**
  - Instantiate Class variables that are data fields
  - Access Class variables
- **Test will:**
  - Assert correctly instantiated class variables
  - Assert correctly assigned class variables
- Note:
  - Each class variable can have a unique data type field. The test for the different types of fields can follow these guidelines.

The table below shows sample test inputs with their respective outputs when unit testing Models.py within the Frost Monitor System.

Test Inputs	Test Outputs
Malformed models.*Field() parameters	Error
Correct models.*Field() parameters	True

Out of bounds models.*Field() parameters	Error
Correct data type class variable assignment	Assigned value
Incorrect data type class variable assignment	TypeError

2.2.2 Serializers.py

The Serializers define JackFrost API representation of the models. The serializers is a built in class of the Django-rest framework. The serializer is used to abstract the conversion of python data types into JSON or XML responses and vise versa for request. In our use case we need to ensure that the fields of our implemented serializers match to the implemented models.

- **All serializers.py can :**
  - Access their base model
  - Access the fields
  - Validate the fields with the fields of the base model
- **Test will:**
  - Assert the validation of the base model fields with the serializer fields

The table below shows sample test inputs with their respective outputs when unit testing Serializers.py within the Frost Monitor System.

Test Inputs	**Test Outputs**
Validate instantiated serializer	Validated Values

2.3 Urls.py

The urls.py is responsible for routing the URLs to the correct views corresponding to those uri.

- **Refer to 1.3**

2.3.1 jackfrost/

./jackfrost/ is the entry point in the Frost Monitor domain for the JackFrost REST API. All URI's that follow jackfrost/ should be directed to their corresponding view.

- Note:

- As models and data changes so will the entry points for API views. As individual REST API response methods are implemented, their corresponding test will follow these same guidelines

The table below shows sample test inputs with their respective outputs when unit testing Urls.py within the Frost Monitor System (Specifically, JackFrost API).

Test Inputs	**Test Outputs**
Malformed HTTP POST	HTTP Status code 400
Correct HTTP POST	HTTP status code 200 & correctly formatted response JSON/HTML
Malformed HTTP GET	HTTP Status code 400
Correct HTTP GET	HTTP Status code 200 & correctly formatted response JSON/HTML
ANY restricted HTTP request	HTTP Status code 405

The results of these test will be studied along with the analysis of our code coverage. More unit test will be developed and successfully completed until we are satisfied with the results. We will then move onto integration testing with the confidence of the stability of our core functionality.

Integration Testing

General Explanation

Our system consists of three main modules: The Nuthatch Data Collection script, the JackFrost REST API, and the Frost Monitor web dashboard application. Each of these represents a stage in the path that data takes on its journey to being displayed within the Frost Monitor web dashboard. It is important to make sure that each module plays its part correctly by communicating in the correct fashion with the other modules. This means that the messages they pass between each other need to be formatted correctly, and no important pieces of messages are left out. Error messages must be sent and handled gracefully so that the system is resilient to complete system failure.

Our first module in our flow of data is the Nuthatch script. This is a python script that will reach out to the nearby Lowell Observatory's weather station in order to grab weather data, and for everything else, will communicate with the FRoST Robotic Telescope. It interprets and packages the data it receives into a format that is accepted by the JackFrost REST API.

When the JackFrost application receives data from the Nuthatch script, it stores the data into the Frost Monitor System's database according to the predefined models. The validity of the data is checked automatically, as Django provides mechanisms for verifying input into the database. For example, if the JackFrost application attempted to post model data that was malformed, Django would automatically detect this and disallow the post from continuing.

The Frost Monitor application (front end) is the what the users are supposed to see; the web dashboard containing all the charts, tables, and images. This is the main interface to the system, providing a graphical way of viewing the information stored in the database. The application sends requests to JackFrost in order to populate the web page, thus putting the carefully gathered data to use. Both the Frost Monitor and JackFrost applications are responsible for sending valid messages to each other in order to communicate and give each other information.

Testing the Interfaces

Nuthatch and JackFrost

Nuthatch will send data to JackFrost according to the models we have implemented. Django will automatically check the sent data against these models, and if there are any extra fields, or missing fields, or incorrectly formatted fields, it will return with an error. Incorrectly formatted fields could be a number where a string is expected.

The tables below specify the fields required for each model.

Weather				
Temperature	Dewpoint	Humidity	Wind Speed	Wind Direction
Number	Number	Number	Number	Number

Telescope Status			
Pointing Position	Dome Position	Local Time	Last Command
Number	Number	Date Time	String

Telescope Log	
Log Type	Log Content
Number	Number

All Sky Image		
Image Link	Image Name	Date Taken
Number	Number	Number

Target Information				
Target Name	ra_deg	ra_dec	Scheduled Start	Scheduled By
String	Number	Number	Date Time	String

For each of these models, if the JackFrost application does not receive all fields specified, it will return an error and not add any data to the database, which will protect the validity of the stored data.

JackFrost and Frost Monitor Applications

For data sent between the JackFrost and the Frost Monitor applications, the model schemes as defined above are also applied for this interface. The data must be formatted the same way, and with the same required fields. The same errors will be sent back, and all behaviors of JackFrost are the same, such as not accepting incomplete data. This lets the two interfaces be consistent with each other, and so we will not have to manage two different kinds of data formats.

Given that we can now have confidence in how effectively our modules communicate with one another, we can move onto testing how well the system and the users interact during usability testing.

Usability Testing

General Explanation

Our project is essentially a single web page with various modules that will showcase weather and telescope data. The look and layout for these modules must meet certain standards and criteria set by our client. The page must load in a reasonable amount of time, must be in the designated layout, and must be able to communicate information quickly and effectively. Not only is this important to meet the client's standards, but will be important for accommodating any additional system users in the future.

Another important facet of user testing will include security. We obviously don't want unauthorized access, but it would be really nice if the intended users were able to successfully log into the website to use features that are not available to anyone else.

For the most part this testing phase will be walking the client through all of the functionality that the website and getting feedback on any changes that must be made. The client will use the website and review it for usability, information accuracy, and navigation. Necessary changes may be made before the client may test further.

The time frame for user testing will begin as soon as testable features are added to the website once deployed, and will continue until the project deadline has lapsed. The test subjects for the user tests will include our client and any additional users from their team. This will include anyone that will continue to use the completed website long term.

Any and all problems found will be documented and communicated to our team, either via email, or during weekly meetings. We will create a detailed list of any issues that arise so that we can easily look back on them. Our team will then make the necessary updates, and inform the client on the changes so that testing may continue further. Essentially, our usability testing will be conducted in an iterative fashion where we will: test, gather results, implement changes, then repeat the previous steps until completion.

Usability Testing Plan

1. Front Page Testing
    a. Module Layout
    b. Module Information and Accuracy
    c. Ease of Use

2. Security Features Testing
    a. Password
    b. Emergency Shutdown Button
    c. Target List Editor

Front Page Testing

Module Layout

The layout of modules on the web page will soon reflect the design that was given to us by the client. It is possible that after the client sees the finished product, that he may wish to make updates to the overall design. This will be tested as needed and may be ongoing until the project is complete. Any suggestions for an updated layout may be corrected and pushed back to the client for review.

Additionally the user will be testing the layout of the modules across a breadth of various devices in order to see if the modules are sized correctly and appear in the correct order on different sized displays such as a tablet or a cell phone display. Modules may appear in any size of our choosing, but must be scalable to appear correctly on any size display, and when the modules are all stacked one on top of the other on a smaller display, they must appear in the correct order. As the client tests the front page across various screen sizes, they will make note of aspects of the modules that may not be to their liking and we can make changes accordingly.

Module Information

Various modules on the front page will display information such as temperature, wind speed, and dew point. Depending on the amount of data that is pulled in to populate the graphs, they may become easily cluttered, or may not have enough data at all. Perhaps headings or labels may be missing.

Each of the modules on the front page are very customizable. Our client will be reviewing these modules individually to test for errors in displayed information or to simply update the displayed format. For example, the client may not be able to actually see the data on the temperature graph because the information is too condensed, or perhaps the layout is correct, but the graph is too small. After testing, we would have to make the necessary changes before allowing the client to test again.

Ease of Use

This is just a general mark of quality for the page itself. We will have the client click around and interact with modules on the front page to see is the overall page is easy for the client to use. Minor tweaks to the page can be made to keep the design satisfactory for the client. The client in this case just has to use the site for a bit to see if any general settings or changes should be made to suit their needs.

Security Testing

Password Testing

Obviously the password given to the client must work and allow the user to get behind the wall of security to access features. We will have the client test out the password until they are satisfied that it works and ensure that everything functions as intended.

At this point the password may be changed to whatever the client would like, and they can test it to make sure it works.

Emergency Shutdown Button

After the client has accessed the button, they will be able to test out the button to see if the satisfactory functionality is present. The implementation of the button functionality will be completed by the client at a later date, which means the button will simply be an endpoint to transfer information of the clients choosing later, but it must be accessible and function as the client expects.

We will have the client test out the button and ensure that the endpoint produces the desired output, if any output is desired.

Target List Editor

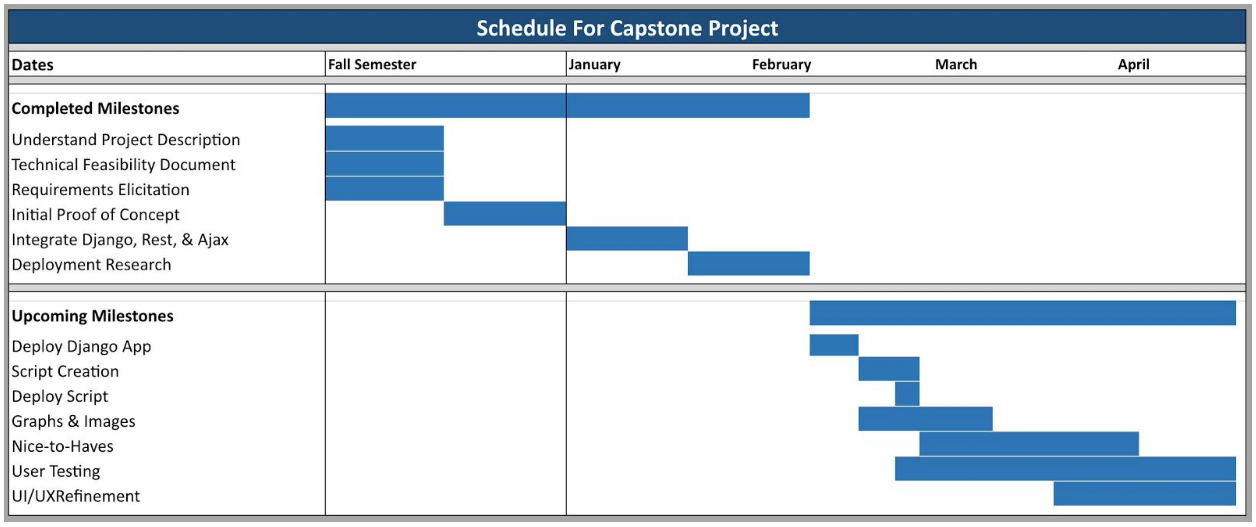
This user test will be more in depth than the others. Rather than just visually inspecting elements, the client will have to interact with the target list editor and ensure that the proper output is provided once editing the list is complete. The actual target list is on the front page, and the editor is behind security on a separate page. This means that the client will have to make changes on one page, then inspect the output of the changes on another page and check for accuracy in the changes made to the information.

The target list editor will also be an endpoint to transfer the changes the any necessary locations off site. We will not be implementing the transfer of information off site. That will be left for the client. This means that the target list endpoint must produce the output needed by the client to make the necessary implementation later.

Testing will continue and be checked for accuracy for the remainder of the project. As long as any discrepancies in information exist, we will be prepared to fix them as long as testing continues.

Project Timeline

Throughout the course of our project, we created many iterations of a project schedule that we tried our hardest to follow. Below you can see the latest version of our projects Gantt chart.



Some major milestones that are outlined above are:

- Technical Feasibility document
- Initial Proof of concepts & prototypes
- Deployment of application
- Testing of the application

The major milestones discussed above were completed in a team-oriented way. For each milestone in our project each member of our team completed various pieces of the work that needed to be done. This method of breaking up the work seemed to work quite well, and we were able to complete many milestones following this structure

---

Future Work

Although our team completed all the major requirements that were laid out by our sponsor, we still have ideas for future work and features that could be implemented in the 2.0 version of the Frost Monitor System. Some of the future work & features we have thought about implementing are:

- Algorithms for predicting weather patterns
  - By predicting weather patterns, our application may be able to more accurately display the chances of bad weather occurring around FRoST.

- Send out alerts when weather exceeds some threshold
    - Sending an alert when certain weather thresholds are met would allow admins of FRoST to quickly get an understanding that FRoST may be in danger.
- Provide an interface to actually control telescope
    - By implementing a way to actually control the telescope using the Frost Monitor System, our application would then contain all necessary features for FRoST researchers to use.

The features discussed above would add even more functionality to our application, and would help our clients even more.

Conclusion

As a team, we feel that this project has great potential to very useful to our client. It will enable them to more effectively use their telescope to scan possible impactors and learn about the NEOs, and thus making a large impact in their productivity.

This project consists of three main parts: the Nuthatch machine, the Django Rest framework, and the FRoST Monitor web page itself. This makes the project as simple as possible which makes it easy to design and implement. In the end, our client will be maintaining it, so having a simple design is best for them and make them happy about having this product.