TEAM SKYWARD

Frost Monitor Project

Software Test Plan

Version 1.0

3/23/2017

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

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.

Conclusion

NEO research is a multi-million dollar research area that could potentially predict potential impacts with NEOs and Earth. The Frost Monitor System we have developed aims to help NEO researchers quickly view important information being transmitted and used by the Flagstaff Robotic Survey Telescope.

The Frost Monitor System development has reached a point where we now need to carry out software testing to validate functionality and make sure our software is free of errors, faults, and failures. To carry out our software testing, we plan on carrying out unit testing, integration testing, and usability testing. We believe that our plan discussed above will be able to prove whether or not our software is ready for operation within a production environment.