Team Ceres

Software Testing Plan

Version 1.0

March 25, 2021

Sponsored By: David Trilling, Michael Gowanlock

Team Mentor: Fabio Santos

Team Members: Javier Quintana, Joseph Sirna, Miles Barrios, Zach Messenger

Table of Contents

1. Introduction

Background

Every night astronomers across the globe participate in all-sky surveys, where the night sky is recorded in hopes to gain knowledge of the galactic entities that surround the Earth. These surveys can produce very large quantities of data and are useful for cataloging notable bodies, such as asteroids. The Zwicky Transient Facility (ZTF) in San Diego, California generates nearly 2 terabytes of data every night and that data alone is very difficult to examine due to the high rate that data is collected.

It is estimated that when the Vera C. Rubin Observatory is finished being built in 2021, 20 terabytes of data will be collected every night for the next 10 years. By the end of the Rubin Observatory's participation in these all-sky surveys, 73 petabytes of data will have been collected.

Our clients, Professor David Trilling and Professor Michael Gowanlock are interested in using this data for their personal research but many of the interfaces that use data from the ZTF, are either outdated or not user friendly. The main problem that our clients face, however, is that they have no interface available to them with functionality that is user-friendly and easy to navigate. Because of this, Professor Trilling and Professor Gowanlock are interested in the development of a new interface that uses data from the ZTF and provides easy-to-access data that doesn't overwhelm its users on the first view. Team Ceres' goal is to create this graphical user interface and to create it with all of the requested accommodations to make it a valuable tool for our sponsors.

Expectations

The purpose of this document is to outline Team Ceres' plan for testing the ZTF Asteroid Analysis Tool (ZAAT) and to provide insight into the tests that will be implemented to ensure that our product runs flawlessly. The team will run a series of unit, integration, and usability tests to find any weaknesses in the interface. While the components of this interface are not all very complex, it is important for us to test

everything because the interface will be hosted online for the use of many concurrent users. The tests that will be performed on the interface will be manually and automatically run depending on the components that are being evaluated.

Our unit testing will focus on isolating different components of the frontend and the backend of our interface. The frontend testing will mainly focus on the look and functionality of the widgets users will interact with, but it will also ensure that the proper frontend pages are being displayed when requested. The backend testing will focus on data delivery and ensure that the right information is being sent correctly to the frontend components.

The integration testing we intend to implement will mostly consist of double-checking the current modules and libraries we have working together already. Since our product is very near completion, the team has already taken steps to combine various technologies and libraries which are essential for the final design. To test our modules further, we intend to put the technologies under higher stress to see if this has any effect on the functionality of the interface.

And finally, usability testing will be handled by our clients and a select group of users whom they have selected based on their background experience and knowledge. This period of testing will act as a sort of “beta-test” to gather outside perspective and information to help improve our product.

2. Unit Testing

The purpose for unit testing the ZAAT interface is to ensure that various specifications of the final product are functioning as intended and not performing any more or any fewer tasks than they should be. It is important to ensure that each component of the frontend and backend of this application is doing just what it is expected to do. Especially because it is anticipated that multiple users will be potentially using the ZAAT interface at once so optimal performance is key to ensure functionality.

As mentioned previously, the unit testing of our application will be broken down into two main components of testing. The first component is the frontend of the application, which handles the user interactions and views. The second component is the backend of the application, which is responsible for managing the data being delivered to the frontend and holding user information privately.

Frontend Unit Testing

Our main user interface is built completely in React, which is a JS framework that uses node.js to run applications. Although we have been testing the functionalities of our application as we have been building it and making sure everything operates as we want it to, we're going to build more structured tests that we will be able to use for testing a larger chunk of our javascript code much more efficiently. Going along with this, we have laid out several test cases that will comprise our unit testing of the frontend.

> Note: For each of our cases we will be using Mocha.js or Jest.js as they interface with React and Node.js well for testing purposes.

Test Case 1: Performing a Query

One key requirement of our application is being able to perform queries on a database of asteroids based on the desired characteristics provided from our users. Without this key feature, our application would virtually not be doing anything. This is why it is important to test this functionality to ensure it works to the best of our ability.

Overall, this test should be pretty straightforward. What we need to ensure here is that our frontend can validate user provided search parameters, then using these parameters construct our query into the proper format. Once the user enters the desired search criteria, the application will validate the data entered, construct a query, and connect with the rest api. Depending on the type of query (i.e., insert vs select), we will also want to handle awaiting a response vs not receiving one.

One key problem to watch out for here improperly formatted queries and not being able to connect with the database. If either of these problems were to occur, it would block the entire application from running since we really rely on this asteroid data.

Test Case 2: Viewing Specific Data Relating to an Asteroid
Another key requirement is that we want our application to be able to access the data for individual asteroid pages. This is the bread and butter of our software, because once users have been able to query the database and get back the results they desire, it is all for naught unless they are able to view this interesting data that drew them to said asteroids.

The first component of this test would be covered by Test 1, and that is being able to query the database. From here, we can find an interesting asteroid and access its page, which pulls the data about the asteroid from the database once again. This test case should be pretty straightforward and once we validate the first test case, this one will be a piece of cake.

Test Case 3: Accessing and Saving Data to User Profiles
This case follows very similarly from the previous two cases, but will interact with new tables in our database that host information relating to the user. The first step of this test will be determining if our frontend is able to interact with our backend. Once this is verified, we'll have to test insert and select statements on this new table with user information.

This data will be relating to user bookmarks, display name, and search result preferences. All of this information will be secure and only accessible if the authentication service, Firebase, detects that a current user is signed in. One key problem to watch out for here is making sure that our queries cannot be broken/tampered with in any way that would allow unauthorized users to obtain data not relating to their user profile.

Backend Unit Testing

On the backend of our application, we have our database and REST API which are doing the heavy lifting when it comes to data storage and retrieval for our application. Again, we have spent a lot of time working with our application functioning as a whole in order to test how the backend is able to handle edge cases. We have still come up with a number of test cases to construct and work through to ensure that our application is avoiding all critical problems.

Since the REST API is written in C# .NET, the logical choice of unit test library is MSTest. We will write several unit tests using MSTest for each core functionality of the backend. These unit tests will be focused solely on internal consistency of method returns from a backend-only perspective; ensuring that non-nullable fields are, in fact, not null, for example.

Test Case 1: Retrieval of a Large Number of Asteroids from MySQL

One of the main focuses of the backend of the application is the ability to serve large amounts of data to the frontend. As a prerequisite, however, it must first be able to query large amounts of data from the database, which it then serializes. This unit tests involves querying every asteroid in the `asteroid` table from MySQL, as well as populating associated observations, and checking the results for anything that failed to populate.

Test Case 2: Updating User Preferences

The API is also responsible for being the intermediary between the frontend and the database in the context of update queries, not just data retrieval. A good example of this communication occurring in the opposite direction is the ability for a user to update their preferences (stored in SQL). In order to emulate this functionality, this unit test will call the method to update user preferences, not in the form of a post request, but a local call in C#. The result will then be checked before and after the update to ensure that all user preferences (results per page, etc) are able to be updated individually.

Test Case 3: Registering / Deleting a User
This unit test will ensure that the functionality to register a user / delete a user works as intended. This will be accomplished as above, by calling the public-facing API method with C# and checking the results in the database before and after each creation/deletion operation on an auto generated test user.

3. Integration Testing

The purpose of integration testing is to take the tests done in unit testing and stack these various pieces on top of each other to make sure that everything still works as expected. When performing integration testing on a component-based front-end such as this project, the testing can be a bit tricky. The approach that Team Ceres has decided to take with integration testing is to test the smaller components individually first and begin to stack these components on top of each other to generate the overall larger components that make up the web application.

The major modules to be tested in the code include the overall app component which embodies the web application as a whole as well as the following pages (components): **Home**, **Search**, **Bookmarks**, and **Account**. These four major pages represent a high-level overview of the components that make up our front-end. Each of these pages is made up of multiple smaller components and/or components that lead to a smaller page. The plan to perform integration testing on a single-page application is to use the popular testing library for React applications known as Jest. Since React uses the state to handle variables and data, we will be able to perform integration testing on components by having components make the various state changes and API calls needed. We will figure out what data should be expected to be in that state and test to see if that data matches when the component is loaded.

The app component (overall web application) gets broken down into the 4 components mentioned above. The app component will be the final piece of the integration testing that gets tested as it is made up of a large amount of smaller components. While the app component will be a little harder to test since there is a lack of state, the current

plan is to assume that by testing smaller components, we can assume that the app component works if it is built off of other tested components. In addition, the team can visually confirm if the component is rendered and acts correctly.

The Home component contains multiple histogram components that are generated using data retrieved from the database via an API call. These histogram components utilize a node package called Google Charts. This library allows for data to be sent to a component and then the corresponding component is generated and returned. In order to test our histogram component, we will make a call to the API to know what data should be returned, we will then use JEST to ensure that the data returned matches what we expected. This can be repeated for all the histogram components generated on the home page. By verifying these smaller components, we are able to verify that the home page is correctly generated.

The Search component contains a search filter column that is responsible for providing data to the frontend via a similar API call to the one used in the Home component. Once a call has been made retrieving the current stored asteroid data, it will be displayed in a sorted order (either ascending or descending based on the filter choice) based on the value. Since data is being transferred we can test this component like we do the Home component and use JEST to verify that the data we are querying is what we should be expecting. To be even more thorough, we can test each property filter in the search criteria. This will allow us to also verify that each and every value present in the query is accurate.

The Bookmark component is currently responsible for maintaining the list of favored user asteroids and has the very simple functionality of displaying a link to an asteroid data graph. While this could be tested with an automatic library, the team decided it may be best to manually test this component as it will be obvious which data needs to be delivered. Testing this component will be as simple as adding specific asteroids to our list of bookmarks, and then verifying that you can access that asteroid. The main reason being that this component does not require much digging to determine if it works.

The Account component works the most with Firebase, as Firebase is responsible for maintaining the user data and preferences. Since Firebase is responsible for the data we are using, we can test the user data storage similar to how we are able to test data provided to us from the home page using Jest. The team is already able to see who has created an account in the interface so the testing would not be as long due to the limited amount of accounts we have already. Testing would simply consist of verifying that a freshly created account matches the expected data we put in.

4. Usability Testing

The purpose of usability testing our interface is to ensure that the tools we have created are easily understandable and can be used by people with various degrees of technological knowledge. Testing with outside users who are not familiar with our program and have not been following along with its development makes them the perfect demographic for ensuring the usability of our program. While the users of our application will only be using the frontend to evaluate asteroid data, there are still plenty of tools on the frontend that we have created and want tested by users.

Below we have created a list of the actions and tools we want users to test during the usability trials:

User Actions

1. Navigate to the ZAAT Interface - Dependent on where the interface is hosted.
2. Navigate to the *Search* page, *Bookmarks* page, an individual *Asteroid's* page, and the *Account* page.
3. Create an account on the interface and log in.
4. Search for an asteroid by any property, then search for an asteroid by name.
5. Bookmark any asteroids then navigate to where that bookmark is located.
6. Go to an asteroid's page from the *Bookmarks* page.

7. Change the X and Y axes on the asteroid graph.
8. Download the asteroid graph and then view it on either the Antares or MARS interface web page.
9. Go back to any saved bookmarks and remove them.
10. Delete the user account being used.

The list we have created for testing has been written as such because it will ensure a number of things as the user processes the instructions. The instructions will first take the user through every single page our interface currently has to offer so they can be aware of what these pages look like for the following steps. After the user has become familiar with the location of these various pages, they will create an account so they can test the functionalities we have implemented that require a user to have an account. As such, they will continue by running queries on the interface and save bookmarks for later testing. After searching and navigating through individual pages of asteroids, the user will test undoing the bookmarks they just created and will finish off by deleting the account they have just created.

All of these instructions will expose the user to all of the functionality currently installed on the interface and have been designated as our baseline instructions to know that a user understands how to properly use our interface. And while we acknowledge that everything the user does is not the entirety of the functionality we have built-in, the important thing that the team is mainly concerned with is a user's ability to find these tools on their own. Once we have had users test these components, we can look for tester feedback to improve the interface.

5. Conclusion

As we approach the finalization of our product, we are taking the unit, integration, and usability testing very seriously. The product has thus far been functional but the testing will take it another step further and make it a much stronger bug-free interface. Once testing has been completed, we can feel confident that our product is one worth delivering to our clients and other end users. If any of our testings were to fail, we would

have potentially system-breaking errors being displayed to our users. These errors could halt the operation of the entire application or limit the users from using certain components while leaving others still up and running. This is why testing is important to us. We want to make sure that we have thought through almost all of the places our application could run into a wall, and make sure that it handles and problems elegantly and with the desired behavior.

Ultimately, we are feeling very confident in our ability to provide our clients with a product they will like and be able to use. As it currently stands, it will soon be impossible to perform analysis on the amount of data being pulled in about asteroids and other small bodies in space. The knowledge gained from this analysis is invaluable though; we could learn about the formation of our solar system and the conditions that led to our civilizations' formation. Additionally, we could better prepare for the inevitability of an asteroid impacting Earth.

That’s where Team Ceres comes in. We plan to give our clients, Professor Trilling and Professor Gowanlock, the tool to be able to perform this analysis. With this, our clients would be able to perform research with ease and efficiency. Currently, our clients work with another colleague in order to perform more complex analysis on the database of asteroids that exists, and this is incredibly inefficient. That means that once this database begins to grow exponentially, their research becomes nearly impossible to complete. And that is why our tool is necessary for them. The ZAAT would be some of the first technology of its kind and could be beneficial to astronomical observers all around the world. With this tool, researchers would be able to perform research at a much faster pace and come to conclusions in record times.