MapONE

Software Testing Plan (Version 1)

April 1, 2022

Sponsors:
Planetary Geologic Mapping Program, USGS Astrogeology Science Center
Dr. Sarah Black, Research Physical Scientist
Marc Hunter, IT Specialist

Faculty Mentor:
Melissa D. Rose

Team Members:
Samantha Milligan
Michael Nelson
Ricardo McCrary
Jacob Stuck

Overview: The purpose of the Software Testing Plan document is to outline how the project's implementation satisfies the necessary functional and non-functional requirements.

Table of Contents

1. Introduction

The planetary science community develops, collects, and distributes cartographic research on the solar system. Scientists use planetary maps and related resources for various reasons - from surveying space exploration sites to collecting data on planetary elemental composition. The client, the United States Geological Survey (USGS) Planetary Geologic Mapping (PGM) Program, assists the community by developing tools and resources to better access and use planetary data for these purposes.

Unfortunately, the community faces challenges in collecting map products across multiple platforms. There are two venues for map publication: through USGS or in various online journal articles and conference papers. Because USGS mandates certain map standards, many non-USGS products are instead distributed in these online publications. Currently, scientists and researchers have to seek out these independent articles to locate map products. This is often time-consuming to view hundreds of science journals across the internet. Nevertheless, USGS is responsible for providing the community with data on all planetary maps regardless of how they are published. Thus, the project team's goal is to collect and display source data on these publications so researchers can quickly and accurately locate non-USGS maps.

The project team has developed MapONE, a web application that displays metadata (source name, link, map body, etc.) on these publications. To populate the application, the project team must first locate articles containing map products. The team uses a web scraper that searches through common science journals and pulls publication metadata when a map product is identified. This data is then passed to the application which can be viewed by users. Instead of wasting time locating map products, researchers can now view, save, and request maps using MapONE.

To ensure user satisfaction and product efficiency, MapONE must be tested for *user convenience, simplicity*, and *source accuracy*. Users should be able to locate source data with ease and in a timely manner. The application's features should be as simple and straightforward to users as possible. The data displayed should be accurate and beneficial to the user's workflow. These

principles will drive the software testing for all MapONE's components - *user interface*, *backend Application Programming Interface (API)*, and *web scraper.*

Software testing analyzes how effectively MapONE handles user requests and how accurately application features are executing. First, each component must be tested independently through *unit testing*. All lower-level functions used to create and configure the frontend and backend must be tested separately. If key functionalities do not work, the entire component will not work in its entirety. Splitting these tests into frontend, backend, and web scraper functions ensures that each component can work on its own. How does the website look without a running API? Does the web scraper pull accurate maps before passing to the backend? These are the key questions that should be answered in this phase.

Next, all components must be tested together through *integration testing* to ensure a fully working product. When connected, does the website display the correct data from the database? Are all backend API requests completed when called from the frontend? Is the right data pulled from the web scraper loaded into the database? These tests must show the correct results for all features in the system (export publications, notify users of map additions, save search history results, etc.).

Lastly, the project team must test MapONE on how well users respond to the application through *usability testing*. Conducting user studies and gathering feedback offers more insight and product improvement. Is it easy to save a publication? Is the user profile straightforward and easy to locate? Overall, these three testing phases ensure each component can work independently, together, and produce the correct output to users. After successful testing, this product should accurately identify map products and pass useful data to the planetary science community.

2. Unit Testing

In software development, unit tests analyze the behavior of a single component, independent of any other functionalities or features. Unit tests are often run for lower-level functions that support an overall system or class. MapONE's interface will be tested by each page - a login

page, main page, and user profile. The backend API unit tests will be divided into user, entry, and archive functions. Lastly, the web scraper tests will be split into keyword and URL generators and a publication result processor.

2.1 GUI

MapONE's interface is created using Flutter, a frontend development software. The frontend will have two test suites: a built-in Flutter unit testing framework and Mockito. Mockito is a mock test suite that creates mock objects from API calls. This mimics the behavior of the backend to test UI elements without an actual API. This is useful to test creating a new user account, changing a password, etc. without accessing and storing objects in the database. In the unit testing phase, each frontend element dependent on an API call can be replaced with data from this mock suite. All other tests will use Flutter's default testing software.

```
// Build our app and trigger a frame.
await tester.pumpWidget(MapOne());
```

Figure 1. An auto-generated unit test that renders the UI and ensures all elements build properly.

```
expect(find.text('0'), findsOneWidget);
expect(find.text('1'), findsNothing);
```

Figure 2. An auto-generated unit test that shows the assertion of values into a dummy widget (a button, table, sidebar, etc.).

The UI's user class performs the management operations of the user account system. To test the user profile, a Mockito object will be created to mimic the behavior of the backend API and test each available action. The following functions are necessary to test the user profile:

- **editPassword()**: Makes the appropriate call to the database through Django to edit the password.
- **getUserID()**: Makes a get request to the backend and serializes the user ID as a *string* type.
- **verifyLogin()**: Accepts the user's input password and sends it to the backend to be verified. Returns a *boolean* check.
- **SizedBox()**: Built-in method from the Flutter widget library that allows other widgets to fit on a specific page section.
- **Card()**: Built-in method from the Flutter widget library that creates a box that can contain a text field and act as a button.
- **CircleAvatar():** Built-in method from the Flutter widget library that creates a circle that can be filled in with a set of colors to represent a randomly generated profile picture.
- **BoxDecoration()**: Built-in method from the Flutter widget library that encapsulates DecorationImage() and AssetImage().

The EntryDataGridSource class is responsible for building the data table populated with publication data. A mock object will be created for the incoming stream of entry data. Unit tests will assert that each column and row have the proper data associated with it. The following functions will be tested using both unit and mock test suites:

- **getApiEntries()**: Fetches JSON data from the backend as future data. A mock object will be created and will mimic the return.
- **buildRow(DataGridRow row):** Builds the data table rows and verifies each row and column has the correct corresponding values.
- **getColumns():** Fetches the data table columns and confirms each column has the appropriate header.
- **buildDataGridRow():** Builds the data grid with row values to confirm each row has the appropriate matching data with the columns.

The login class is responsible for rendering the application's login page. The following functions will be tested in this class:

- **createUserAccount():** Submits an API request to create a new user using a mock object and confirms all status codes returned have a successful response code.
- **LoginAsUser():** Verifies user login and displays user profile data and confirms user data is properly rendered on the interface.
- **LoginAsGuest():** Logs the user in as a guest and pushes the guest profile onto the screen.

2.2 Backend API

MapONE uses Django, a web framework software, for the backend API. Django offers built-in testing modules for unit testing called *test cases*. The backend will be broken down into three test cases based on the API's user, entry, and archive class functions. As a note, to implement Django test cases that use database objects, objects must first be created in the test case initializer.

The user class is responsible for storing all user account information. The user test case will test the following functions in this class:

- **change_password()**: Ensures previously saved user objects have an updated password after calling the function with a new password.
- **check_existing_user()**: Verifies previously saved user objects exist in the database.
- **create_new_user()**: Creates a new user object and checks the database for addition.
- **delete_user()**: Deletes user object and checks database for deletion.
- **generate_user_id()**: Checks if the newly generated user ID is greater (by one) than the maximum user ID in the database. New users must be the next user in the database.
- **send_notification()**: Verifies email was sent to a given email address.
- **verify_email_address()**: Verifies email addresses in the mail server.
- **verify_password()**: Verifies if passwords follow security standards (character length, numbers, special characters, etc.).
- **verify_user()**: Ensures email address and passwords passed in belong to the same user saved in the database.

The entry class is responsible for storing all publication information. The entry test case will test the following functions in this class:

- **check_existing_entry()**: Verifies previously saved entry objects exist in the database.

- **create_new_entry()**: Creates a new entry object and checks the database for addition.
- **delete_entry()**: Deletes entry object and checks database for deletion.
- **filter_year()**: Ensures returned entry objects fall between two given years.
- **generate_entry_id()**: Checks if the newly generated entry ID is greater (by one) than the maximum entry ID in the database. New entries must be the next entry in the database.
- **get_all_entries()**: Ensures all entry objects are returned in the database.
- **search_keyword()**: Ensures all entries associated with a specific keyword are returned. No entries are left out and unnecessary entries are not returned.
- **verify_entry()**: Ensures entry does not already exist given an article title and author.

The archive class is responsible for storing all automated search information. The archive test case will test the following functions in this class:

- **create_new_archive()**: Creates a new archive object and checks the database for addition.
- **delete_archive()**: Deletes archive object and checks database for deletion.
- **generate_archive_id()**: Checks if the new generated archive ID is greater (by one) than the maximum archive ID in the database. New automated searches must be the next archive in the database.
- **get_entry_number()**: Ensures returned number is the same number of entry objects in the database under a keyword.
- **get_searches_by_frequency()**: Ensures all archives in the database with a given frequency are returned.
- **get_user_saved_searches()**: Ensures all archive objects under a user object in the database are returned.
- **update_frequency()**: Ensures previously saved archive objects have updated frequency after calling the class function with a new frequency.

2.3 Web Scraper

MapONE's web scraper must generate valid URLs using a set of keywords and scrape the metadata from the results. The URL generator and all supporting functions must be tested to ensure valid source information. Next, the publication processor must be tested to view data from

source information. Lastly, all data extraction functions must be tested to ensure accurate map products are located. Testing will be conducted using *AREPL*, a software extension in VS Code.

Provided with a set of keywords, the URL generator is responsible for scraping all web pages from a given library. Each URL is queried into a list and passed into the processor for further scraping. The first test case will run through the following functions regarding URL generation:

- **url_generator()**: Scrapes all possible webpages of a given set of keywords.
- **url_reader()**: Uses the *urllib* function with *urlopen* to extract the HTML code of a given page.
- **keyword_url_generator()**: Utility function that provides a starting URL and converts the keywords to a list of words for parsing.
- **argument_formatter()**: Splits the keywords argument string so it can be used across the rest of the script.

Following the generation of URLs, the publication processor scrapes each of the results in more detail, allowing the web scraper to analyze specific areas of each web page (title, date, abstract, etc.). Functions to test for the processor include the following:

- **processor()**: Main page-cycling function to scrape multiple resulting pages returned from Springer.
- **page_souper()**: Scrapes the web page elements and provides tags for scraping using Python's *BeautifulSoup* library.
- **abstract_id_scraper()**: Obtains ID PII number of each publication. Is part of the main URL for each page, but is separated to reduce the workload on the web scraper.
- **abstract_scanner()**: Crawls the page of results and accesses each publication.
- **delay_function()**: Reduces the number of pings sent to Springer servers in case of server issues.

Once each web page has been processed, the web scraper can now retrieve all of the desired planetary map publication information. This includes the title, date, author, body, and abstract. The functions to scrape each piece of data from the publications include the following:

- **abstract_date_scraper()**: Scrapes the date associated with each of the publications.

- **abstract_scraper()**: Scrapes the abstract from the soup and returns to the page scraper.
- **author_scraper()**: Scrapes the author associated with each of the publications.
- **title_scraper()**: Scrapes the title of the text from each of the publications.
- **aoi_scraper()**: Scrapes the area of interest (body) from each of the publications.

3. Integration Testing

Unlike unit testing, integration tests analyze the behavior of the *whole system* given all components, functionalities, and features. These tests monitor the interactions and communication between the system's main modules - in this case, the UI, API, and web scraper. MapONE's interface will be tested on how it makes API requests from the backend to complete user requests on the frontend. The backend API will be tested on both its interactions between the UI (via GET requests) and the web scraper (via the database). Lastly, the web scraper will be tested on its interaction with the API to save scraped publications in the database.

3.1 Website

For integration testing, the interface must be tested on how well it communicates with the backend API. Similar to unit testing, the interface's integration tests will be broken down into the main page, login page, and user profile features. However, unlike the mock suite, the objects created and gathered from the backend must reflect current database entries. First, the main page will be tested to ensure all publications listed in the backend's database are displayed on the application in alphabetical order. Next, the login page will be tested to see if a new user object has been created in the database and if all passed email addresses and passwords are associated with real user objects. Lastly, the user profile must display all saved automated searches associated with a user object in the database. All related features (exporting publications, creating new automated search, etc.) must be tested as well. Essentially, all UI elements in the frontend should be verified by real database objects in the backend.

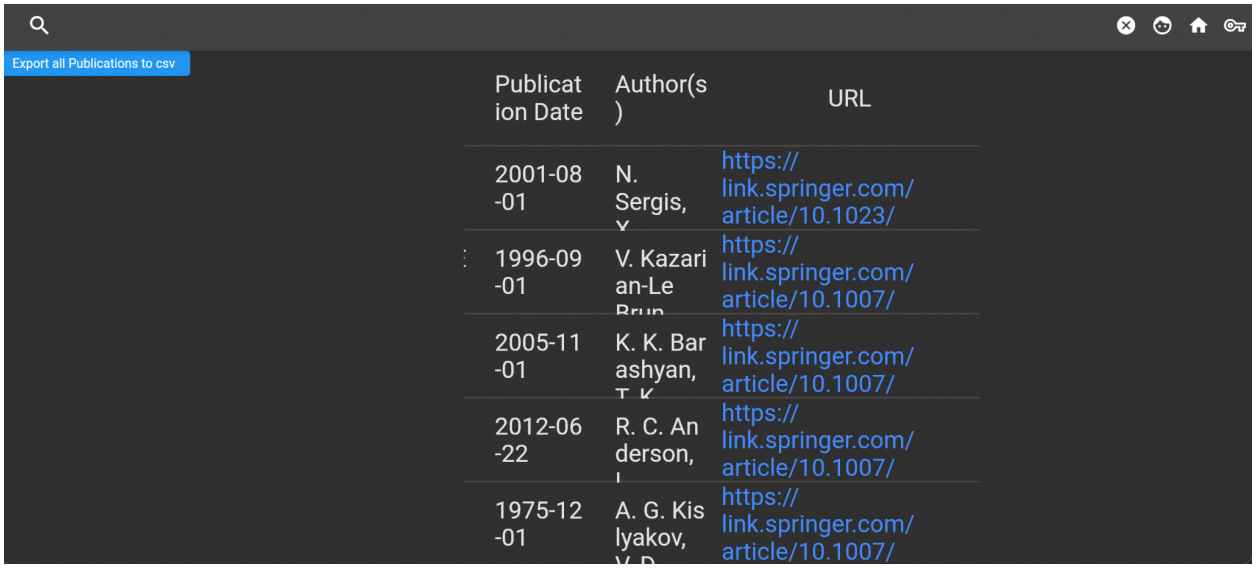

Figure 3. MapONE's main page when connected to the backend API.

3.2 API Requests

Similar to unit testing, Django test cases can be used to test API requests. MapONE uses three APIs to interact with the frontend: user, entry, and archive APIs. The following actions and necessary parameters can be used to access the user API (via website/user):

- **CREATE_USER**: action, email_address, password
- **LOGIN**: action, email_address, password
- **CHANGE_PASSWORD**: action, email_address, password, new_password
- **DELETE_USER**: action, email_address, password

These requests allow the frontend to create a new user, successfully log in a user, change a user's password, and delete a user account.

Likewise, the following actions and necessary parameters can be used to access the entry API (via website/entry):

- **MAIN_PAGE**: action

```
GET /entry/?action=MAIN_PAGE
HTTP 200 OK
```

```
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "entry_id": 1,
        "source_name": "Springer",
        "source_link": "https://link.springer.com/article/10.1023/A%3A1011861828314",
        "article_title": "Mars Map",
        "publication_date": "2001-08-01",
        "author_list": "N. Sergis, X. Moussas",
        "map_body": "Mars",
        "map_scale": null
    },
    {
        "entry_id": 2,
        "source_name": "Springer",
        "source_link": "https://link.springer.com/article/10.1007/BF00645774",
        "article_title": "Mars Map"",
        "publication_date": "1996-09-01",
        "author_list": "V. Kazarian-Le Brun",
        "map_body": "Mars",
        "map_scale": null
    }
]
```


Figure 4. Entry API request for action MAIN PAGE.

- **SEARCH_KEYWORD**: action, keyword
- **FILTER_YEAR**: action, first_year, second_year

These requests allow the frontend to display all entries in the database to the website's main page, look up entries based on a keyword, and filter publications by date.

Lastly, the following actions and necessary parameters can be used to access the archive API (via website/archive):

- **CREATE_ARCHIVE**: action, email_address, password, keyword, frequency
- **DISPLAY_USER_ARCHIVES**: action, email_address, password
- **DELETE_ARCHIVE**: action, email_address, password
- **UPDATE_FREQUENCY**: action, email_address, password, keyword

These requests allow the frontend to create a new automated search, display all saved archive objects under a user ID, delete an automated search, and update the frequency of a saved search.

3.3 Scraped Publications

Publications that have been retrieved by the web scraper can be written into a CSV file via the function *abstract_database_writer()*. This ensures that data can be extracted, printed, and stored in the backend. An example output of the CSV is provided in the following figure:

Springer, https://link.springer.com/article/10.1007/s11214-021-00834-7, 'Correction to: Meteorological Predictions for Mars 2020 Perseverance Rover Landing Site at Jezero Crater', 2021-06-14, (Jorge Pla-García, S. C. R. Rafkin, G. M. Martinez), Mars,

Springer, https://link.springer.com/article/10.1007/s11214-020-00739-x, 'Photogeologic Map of the Perseverance Rover Field Site in Jezero Crater Constructed by the Mars 2020 Science Team', 2020-11-03, (Kathryn M. Stack, Nathan R. Williams, Fred Calef III), Mars,

Springer, https://link.springer.com/article/10.1007/s11214-020-00788-2, 'Multi-model Meteorological and Aeolian Predictions for Mars 2020 and the Jezero Crater Region', 2021-02-08, (C. E. Newman, M. de la Torre Juárez, J. Pla-García), Mars,

Springer, https://link.springer.com/article/10.1007/s11214-020-00763-x, 'Meteorological Predictions for Mars 2020 Perseverance Rover Landing Site at Jezero Crater', 2020-12-14, (Jorge Pla-García, S. C. R. Rafkin, G. M. Martinez), Mars,

Figure 5. CSV file output of the main scraper function.

The project team must verify that all scraped publications are automatically added to the database. Since the web scraper's script is written into the backend, when the tool extracts publications, the database should reflect the same data as in the CSV files. How similar the files are to the database every time the web scraper is run determines the success of this testing phase.

4. Usability Testing

In the last phase of the test plan, usability testing is the process of obtaining feedback from potential users to gain insight into the overall product's quality. This testing phase sees how well users react to the product, and thus, is crucial to the team's understanding of how users would view the finished product. By collecting feedback, the development team will be able to improve and adjust the product for a better user experience.

This testing can be conducted through user studies. Since the application's intended users are researchers and scientists in the planetary science community, the project team will create a survey to send to the client and their colleagues. The goal is to collect fifteen to twenty surveys from the USGS PGM program staff. The survey will include a link to the website along with general information about MapONE's purpose and features. Essentially, the criteria for users to participate in this study are the following:

1. Must have access to a computer to test the web application.
2. Must be a member of the planetary science community (includes USGS PGM staff).

From there, the user will be asked to access the product's features and review the following example questions:

1. **Main Page**: What data is listed? What data is unnecessary? What data could be added? What do you think of the filters and search bar? Did the website show relevant and useful sources? Did the sources contain map products useful to your own research?
2. **Login Page**: How easy was it to create a user account and log in?
3. **User Profile**: How long did it take to locate the user profile? Is the icon intuitive?
4. **Other Feedback**: Gives users the opportunity to offer more feedback. What worked? What did not make sense? Any improvements or suggested features?

5. **Customer Satisfaction**: Users will have the opportunity to rank their experience on a scale of one to five stars.

Based on the responses, the team will use the feedback to refine the application. As a general rule, if 50% or more surveys offer the same suggestion, the feature must be modified or implemented. However, all refinements will be discussed and approved by the client first.

5. Conclusion

As a web application, MapONE's goal is to display publication metadata to researchers and scientists in need of source information for non-USGS map products. The system's main modules are the interface, backend API, and web scraper. Each component and its functions must be tested independently through unit tests. These tests determine the success or failure of the component's functionality as a single module. The project team will then use integration tests to view the system's entire system when completing user and API requests and storing objects in the database. Lastly, user studies will help the project team to gain insight into the user's perspective of the product to improve usability and ensure client satisfaction.

As the project team continues its refinement stage in fixing errors and system bugs, this testing plan will ensure MapONE satisfies all client requirements. The project is on track for product delivery in May 2022. The team will continue to demonstrate the improving prototype to the client on a weekly basis.