MapONE

Software Design (Version 2)

February 18, 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 Design document is to outline the project's architectural design.

Table of Contents

1. Introduction

The planetary science community develops, collects, and distributes cartographic research. Similar to geologic mapping on Earth, planetary maps identify key features such as craters, canyons, and fault lines on other planets. Scientists use these 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. Notably, the program outlined the landing/exploration site for the National Aeronautics and Space Administration's (NASA's) Mars 2020 Perseverance Rover mission. In this case, scientists used map products to gather information about Mars's surface.

Scientists publish their planetary maps and research either through USGS or various online sources (often in journal articles and conference papers). Because USGS mandates certain map standards, many publications are distributed throughout the internet instead. Nevertheless, these resources are valuable to the client to provide the community with all types of publications. However, USGS currently does not have an automated way to collect these existing resources and thus, lacks a centralized system for maps published in non-USGS venues.

To solve this integration issue, the project team will create a single system for non-USGS map publications. The envisioned product, MapONE, is a user-friendly interface accessible through the USGS's website. The interface will display map metadata for a selected region of interest and connect to a database containing online non-USGS publications collected from a web scraper. The following technologies will be used to create this product:

1. **Graphical User Interface (GUI)**: A GUI will be used to display map data on a single web application.
2. **Web Framework**: MapONE's web framework will filter metadata (author, source, region, etc.) from the database using user requests from the GUI.
3. **Web Scraper**: The project team will develop a data extraction tool, known as a web scraper, to automatically detect and gather map publication metadata (author, source, region, etc.) from online non-USGS sources.

4. **Central Database**: MapONE will store publication metadata in one centralized database.
5. **Remote Storage and Servers**: Cloud computing services will be used to store map data and run the product's web application.

At the system's center, the web scraper locates online map products and extracts publication metadata for the web framework to store in the database. The framework will then display the data to the client using a GUI. This way MapONE ensures these publications and content are available to the client. Together, these technologies will create a system for users to view and archive map publication data.

MapONE also has four main features. The **user account system** allows users to login into a USGS account to view previous search history and archived publications. The **search engine** allows users to view and filter map metadata. The **archive system** can automate searches periodically (monthly per client request). Lastly, the **notification system** informs users of new publications from automated search results. Some key requirements also include how accessible the GUI is to users and the response speed of automated searches (how often the web scraper pulls data). These are vital for MapONE's performance to efficiently pull and display data. The product must also be a web-based, open-source Python tool as requested by the client.

2. Implementation Overview

MapONE's success relies on the software integration of many technologies. The project team will use **Flutter**, a frontend web-based software, to create the GUI. **Django**, a Python full-stack web development software, will be used as the web framework which automatically sets **SQLite** as its default database structure. Lastly, **Keras**, a Python Machine Learning (ML) library, will be used to create the web scraper.

To connect the GUI (frontend) to the web framework (backend), the two will run on separate servers. The GUI can then execute an Application Programming Interface (API) request to the Django server to gather information from the backend and display to users on the frontend. The web framework uses a file called *models.py* to structure the database which can be called

anywhere within the framework including *views.py* where the API exists. This connection allows the GUI to access metadata from the framework's database. As shown in *Figure 1*, the web scraper and the framework's automated verification process are responsible for populating the database with correct article entries. Each entry the web scraper collects must be internally checked and verified to be stored in MapONE's database (Entry Class). This ensures that the web scraper remains active at all times (continuously collecting entries) and instead allows the database and other backend functions to verify entries separately. Overall, the product's main technologies (GUI, web scraper, and database) will remain separate systems to enforce modularity and ease of implementation.

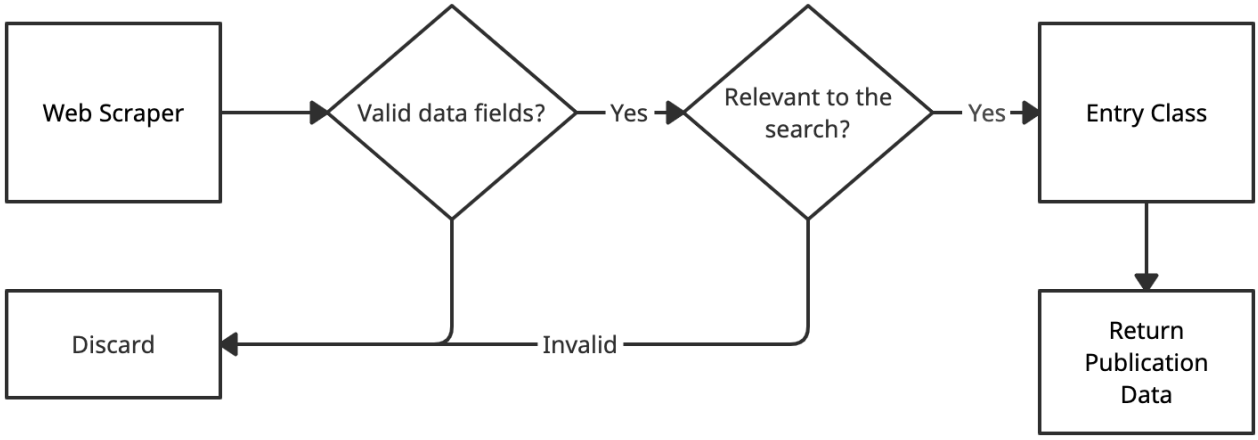

Figure 1. Entry verification process.

3. Architectural Overview

As shown in *Figure 2*, MapONE will run on two separate servers for the frontend and backend. The GUI will handle all user requests through the web application. These requests will be translated into API calls where the GUI requests information or invokes a specific operation from the backend. For example, when a user logins, the GUI will send the user's input (given email address and password) to the web framework in the form of an API request. The web framework will then verify the login information and send the GUI a successful or client error Hypertext Transfer Protocol (HTTP) response depending on if the user is valid or invalid. Essentially, the GUI and web framework will only communicate via API requests and use HTTP responses to pass data.

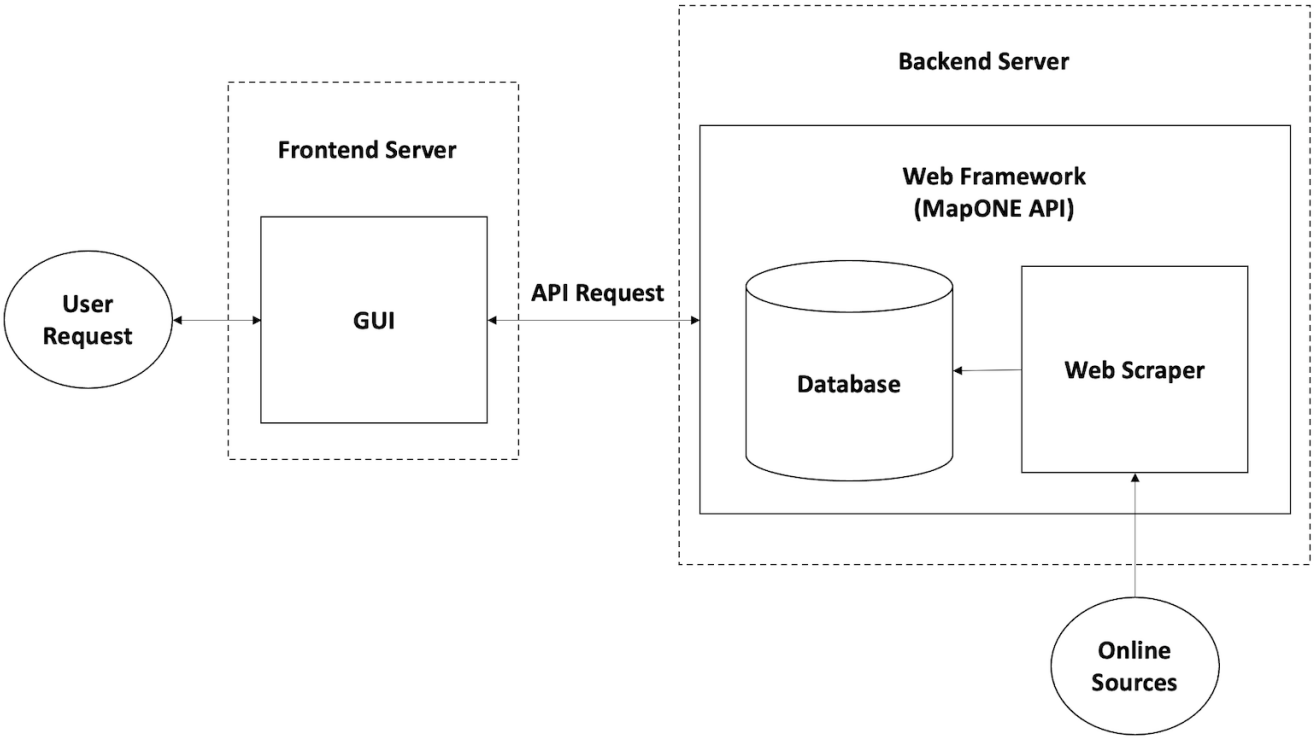

Figure 2. MapONE's high-level architectural diagram.

Within the web framework, the web scraper solely extracts metadata from online sources containing map products and passes entries to the database to be verified and added to MapONE's publications. This is a one-way communication link where the web scraper passes entry information and relies on the database structure (and other operations within the web framework) to verify map metadata. Thus, only the database handles entry and user information. Ultimately, all data and core operations provided to users are stored in the backend and can be accessed by the GUI given a user request.

4. Module and Interface Descriptions

4.1 GUI

As the front of the system, the GUI is responsible for handling all user requests. As previously mentioned, Flutter is used to build the system's frontend. The software uses the Dart language to construct classes and to implement UI functionalities (colors, buttons, etc.). In Flutter, every class inherits either stateful or stateless widgets. Stateful objects control the UI's output given a

change of state. Examples include text fields or checkboxes; the UI's response changes depending on the user's input. On the other hand, stateless widgets remain constant (ex. icons, theme colors, title names, etc.). As a note, the underscore symbol is used to denote a class managed by a stateful widget.

MapOne Class

This is the root class of the *main.dart* file where all stateless widgets are defined. The following class variables are necessary for all UI pages:

- **primaryColor**: Sets the color theme for the UI as a *ThemeData* type.
- **title:** Sets the title of the application as a *MaterialApp* type.

This function is also used to run the entire frontend application:

- **build(**BuildContext context**)**: Builds the UI as a stateless variation.

mapOneHomepage Class

This class builds the application's homepage and sends user input and output data to _MapOneHomePageState. The class uses the following functions to define the homepage:

- **super():** Default constructor allows the homepage to access the widget functions in the StatefulWidget class.
- **mapOneHomepage2():** Additional constructor that allows the execution of calls from the BackendCalls Class without requiring any initialization parameters.

_MapOneHomePageState Class

This class manages the state of the context builder which allows for changes in the application's view. Essentially, this class handles all dynamic actions on MapONE's homepage. The following class variables and functions are used to communicate and send data between the frontend and backend:

- **backendConsume**: Allows the BackEndClass to communicate with _MapOneHomePageState and populate the main data table.
- **fetchApiData()**: Converts data passed from the backend API to a *string* type.
- **Scaffold()**: The root of the application's view and contains several nested widgets such as appBar(), the topmost bar in the UI.

- **AppBar()**: Builds the search and menu bars to accept user text input.
- **Center()**: Assists Cascading Styling Sheets (CSS) to place UI items in the center view of the application.
- **Column()**: Places UI items in the center of the screen in a vertical manner.
- **DataTable()**, **DataColumn()**, **DataRow()**, and **DataCell()**: All functions used to construct and populate the data table icon where all the planetary mapping publication metadata will be displayed.

backEndCalls Class

This class facilitates the communication between the backend API and the GUI.

- **title:** Sets the title of the application as a *string* type.
- **consumeApi()**: Makes an API request to the backend.

Data Class

This class is responsible for managing and organizing the data that is received in BackendCalls into a data table. The following class variables and functions provide this functionality.

- **publicationNode**: Holds a reference to a list containing the serialized publication data from the backend.
- **inData**: Holds data from fetchApiData() as a *string* type.
- **populateDataRows(**source, link, body, scale, author, publicationInfo**)**: Returns a data row containing the specified publication metadata as a *DataRow* type.
- **serializeAsyncBuilderData(**asyncStr**)**: Converts serialized data into a *string* type.
- **PopulateLL(**SerializedStr**)**: Populates a list containing new data pulled from the backend.
- **IterateThroughLL(**serializedStr**)**: Iterates through the data list from backend API and extracts publication metadata.

User Class

The user class is responsible for displaying user profile information and available features (view profile picture, change password, etc.). The following class variables and functions are necessary to populate each profile page:

- **userid**: A generated unique identifier (ID) for each user as an *integer* type.
- **editPassword()**: Makes the appropriate call to the database through Django to edit the password.
- **createUserAccount():** Makes a rest call to create a new user in the backend.
- **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 be encapsulated to fit them onto a specific section of the page.
- **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() which is currently set to display a picture of the Mars Rover as shown in *Figure 3*.

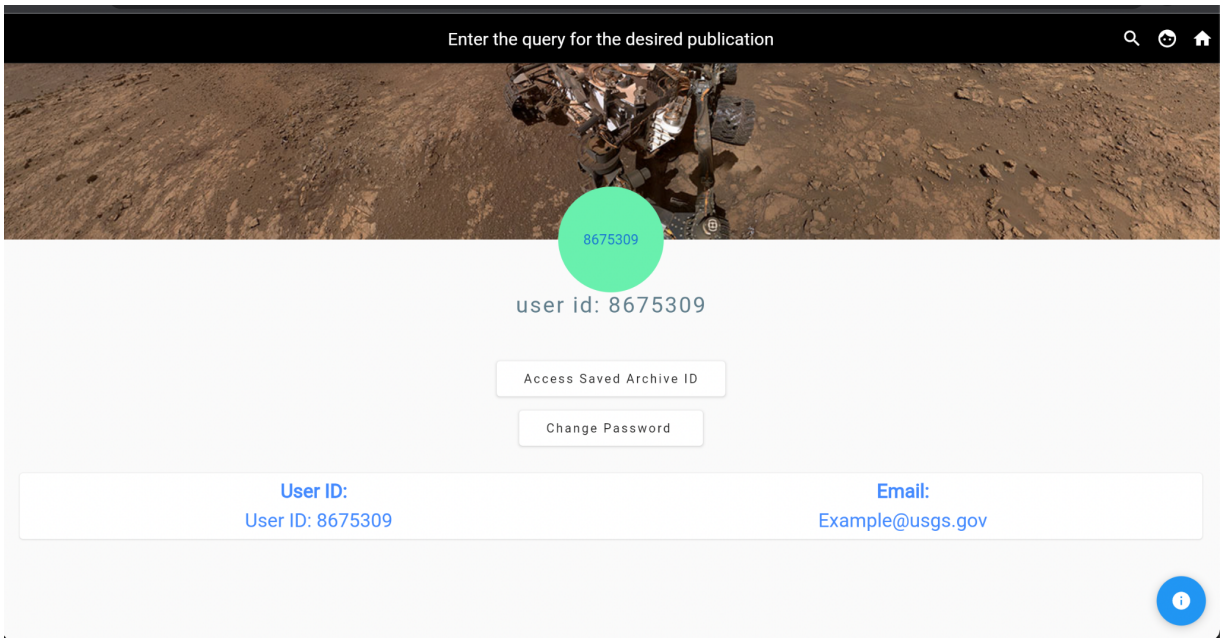

Figure 3. MapONE's user profile panel.

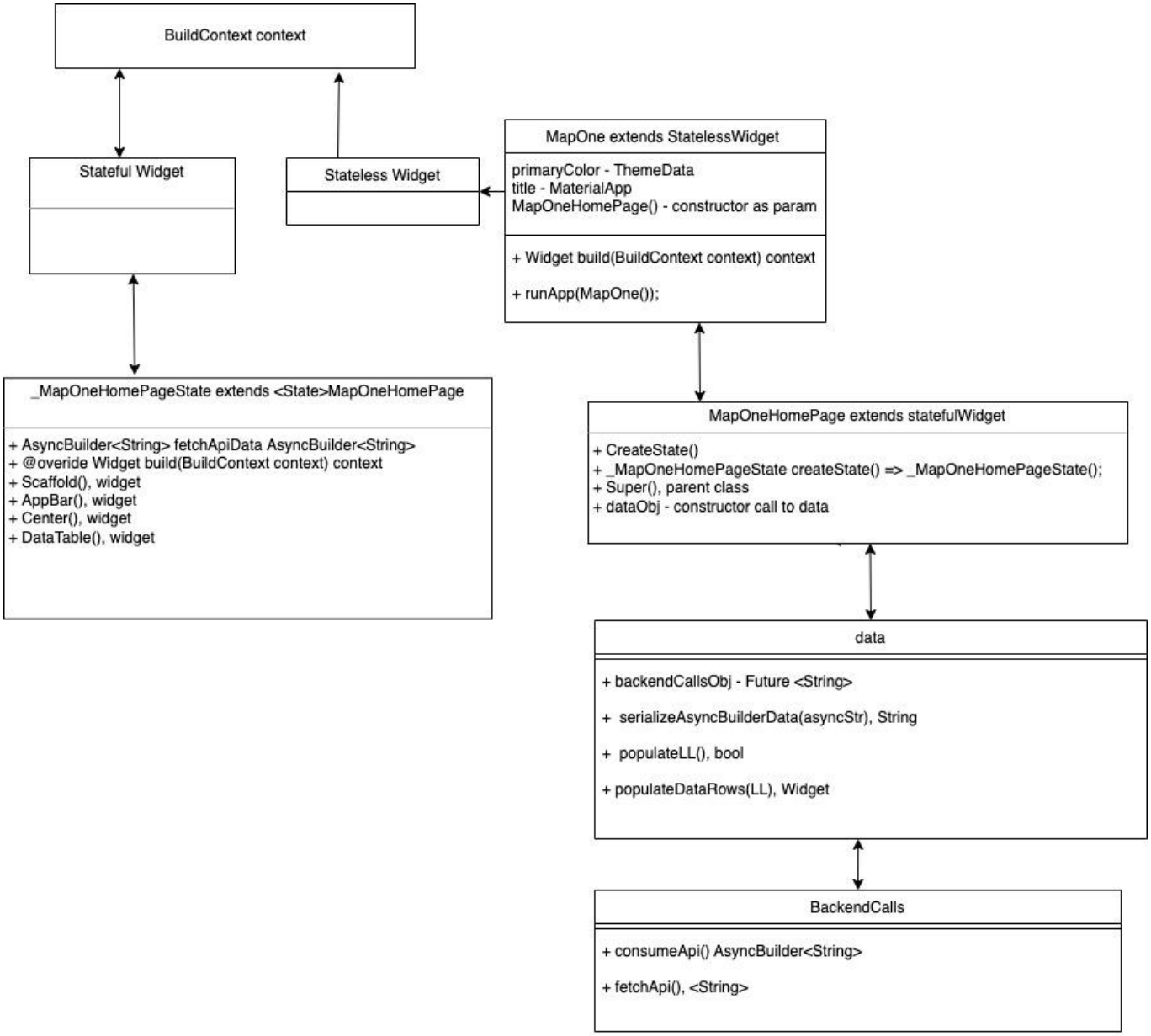

Figure 4. UML Diagram of MapOne's UI

4.2 Web Scraper

MapONE's web scraper extracts metadata from online publications to store in the system's database. This metadata is accessed via function calls that correspond to the location of

information within the article. Specifically, the web scraper identifies the title, author(s), area of interest, scale, and link associated with the online publication. The functions for each data pull as well as the necessary utility functions for scraping a website are listed below:

- **url_reader(**url**)**: Establishes a connection to the URL using the *urllib* library.
- **title_scraper(**title**)**: Returns *string* representation of the title within the publication.
- **author_scraper(**author**)**: Returns a *list* of authors from the text of the publication.
- **area_scraper(**area**)**: Pulls data related to the publication's planetary map of interest.
- **scale_scraper(**scale**)**: Returns *string* representation of the scale within the planetary map.
- **abstract_scraper(**abstract**)**: Returns the first body of text from the publication as a *string*.
- **url_scraper(**link**)**: Returns the link of the current URL as a *string*.
- **database_writer(**site_url_index**)**: Writes the publication data to a .txt/.csv file.

ML Model

To automate the web scraping process, a ML model runs scheduled monthly data pulls to check and add new publications. The algorithm currently pulls data from the Springer library, a popular science journal for map products. The project team continues to train the model to identify articles from other website libraries/catalogs as necessary. The model locates and logs publications based on keywords provided by user searches on the frontend. The content found will be presented within the logged folder generated by the *status_logger* function. This will help with organizing information for the database writer in the web scraper. Current functions for the ML framework are provided below:

- **url_generator(**urls_to_scrape**)**: Scrapes all publications/web pages of the related topic and returns a *list* of possible URLs to scrape.
- **processor(**keywords, urls_to_scrape**)**: Provides functionality to scrape multiple pages of results in a library.
- **status_logger(**log_names**)**: Prints scraping progress and details regarding each publication as a *list*.
- **pre_processor(**keywords**)**: Creates separate files to log keywords and their results.

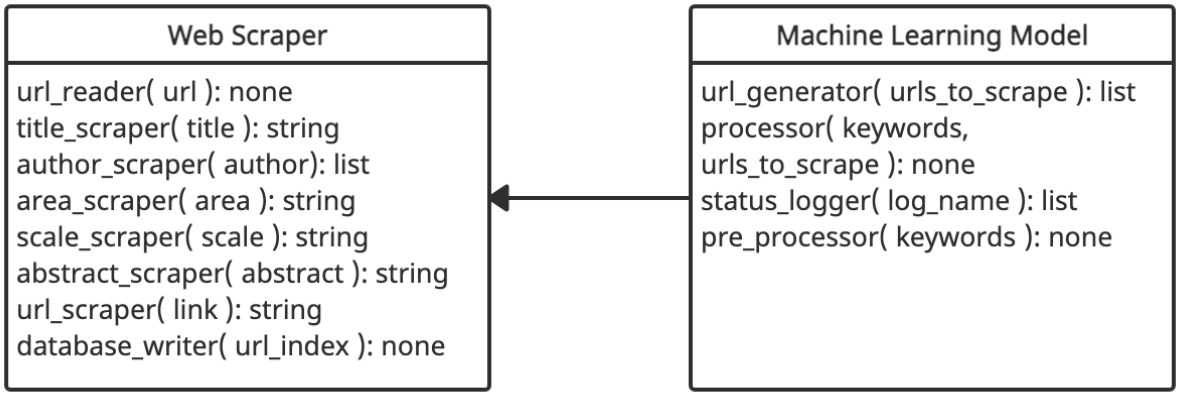

Figure 5. MapONE's web scraping utilities.

4.3 Database

MapONE's backend development relies on the modularity and accessibility of the database. Both the data gathered from users on the frontend and publication data from the web scraper can only be accessed and stored through the database. To create a transferable system as shown, the database must be broken down into user, entry, and archive classes (*Figure 6*) to support MapONE's four main systems.

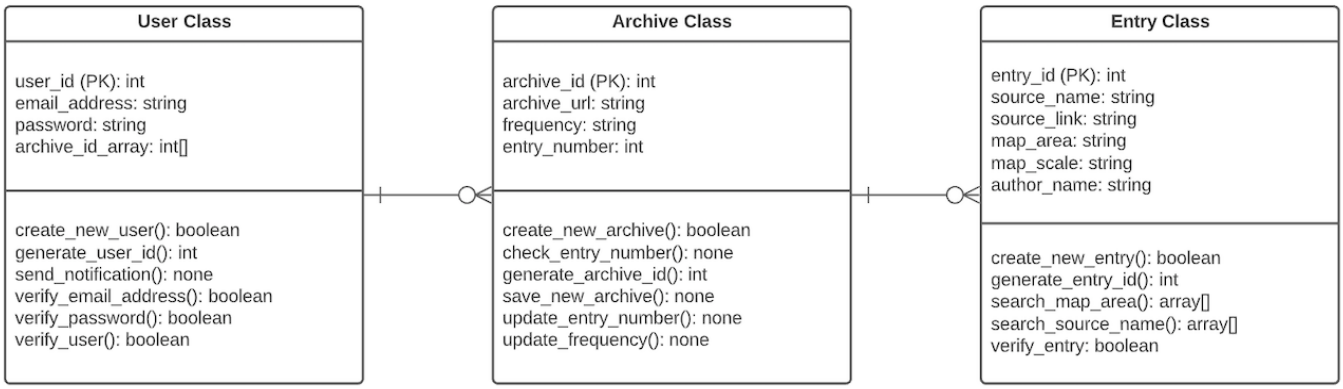

Figure 6. MapONE's database classes.

User Class

The user class is responsible for storing all user account information. This allows the system to directly communicate with users at the login and user profile levels. The following class variables are necessary for each user:

- **user_id** (PK): A generated unique identifier (ID) for each user as an *integer* type.
- **email_address**: A user's email address used as the username to login as a *string* type.
- **password**: A user's password used to login as a *string* type.
- **archive_id_array**: A list of archive IDs from automated searches requested by the user as an *integer array*.

Within the scope of the user class, MapONE must be able to verify, login, and send notifications to users using an authentication process. To implement these functionalities, the following class functions are necessary:

- **create_new_user(**email_address, password**)**: Creates a new user in the database and returns *Boolean* operation success.
- **generate_user_id()**: Creates a generated unique identifier (ID) for each new user and returns user ID as an *integer* type.
- **send_notification(**user_id, message**)**: Sends a notification to user's email address and returns *None*.
- **verify_email_address(**email_address**)**: Each new user must supply a valid email address to be a user. This function returns whether the given email address is valid or invalid as a *boolean* expression.
- **verify_password(**password**)**: Likewise, each new user must also supply a valid password. This function returns whether the given password passes the system's credentials (eight characters long and includes at least one special character) as a *boolean* expression.
- **verify_user(**email_address, password**)**: Verifies login input and returns *boolean* operation success.

Entry Class

The entry class is responsible for storing all publication information. This allows the system to directly communicate with users and the web scraper at the search engine level. The following class variables are necessary for each publication entry:

- **entry_id** (PK): A generated unique identifier (ID) for each publication entry as an *integer* type.
- **source_name**: Publication's source name as a *string* type.
- **source_link**: Publication's source Uniform Resource Locator (URL) as a *string* type.
- **map_area**: Publication's map area as a *string* type.
- **map_scale**: Publication's map scale as a *string* type.
- **author_name**: Publication's author name as a *string* type.

MapONE's web application must locate specific publication information requested by users through the search bar and filter features. To implement this functionality, the following class functions are necessary:

- **create_new_entry(**source_name, source_link, map_area, map_scale, author_name**)**: Creates a new publication entry in the database and returns *boolean* operation success.
- **generate_entry_id()**: Creates a generated unique identifier (ID) for each new entry and returns entry ID as an *integer* type.
- **search_map_area(**map_area**)**: Searches and returns all entries in the database with the given map area as a list of arrays. The following is an example:

```
[
        {
                "entry_id": 1234,
                "source_name": "Northern Arizona University",
                "source_link": "nau.edu",
                "map_area": "Mars",
                "map_scale": " ",
                "author_name": "John Smith"
        }
]
```

- **search_source_name(**source_name**)**: Likewise, searches and returns all entries with the given source name as a list of arrays.
- **verify_entry(**source_name, source_link, map_area, map_scale, author_name**)**: Each entry collected by the web scraper must be verified to be stored in the database. This function verifies all entry information and returns *boolean* operation success.

Archive Class

The archive class is responsible for storing all automated search information. This allows the system to directly communicate with users at the archive and user profile levels. The following class variables are necessary for each automated search:

- **archive_id** (PK): A generated unique identifier (ID) to locate automated searches and search history for each user as an *integer* type.
- **archive_url**: Web application's URL for saved results after an archived search as a *string* type.
- **frequency**: The frequency (weekly, monthly, yearly, etc.) of how often the automated search should be updated/checked for new added entries as a *string* type.
- **entry_number**: The number of entries returned from an archive URL as an *integer* type.

The product must be able to track automated searches designated by the user and recognize when new publications are added to the database. To implement these functionalities, the following class functions are necessary:

- **create_new_archive(**archive_url, frequency, entry_number**)**: Creates a new automated search in the database and returns *boolean* operation success.
- **check_entry_number(**archive_url, frequency, entry_number**)**: Checks current number of publication entries pulled from the application's URL. If the entry number is greater than the past stored number, the system updates the stored entry number and sends a notification to inform the user. This function returns *None*.
- **generate_archive_id()**: Creates a generated unique identifier (ID) for each new automated search and returns archive ID as an *integer* type.
- **save_new_archive(**user_id, archive_id**)**: Saves new archive ID in the user's profile (appends the archive_id to the user's archive_id_array) and returns *None*.

- **update_entry_number(**archive_id, entry_number**)**: Updates entry number as noted in check_entry_number() and returns *None*.
- **update_frequency(**archive_id, new_frequency**)**: Updates frequency of automated search under user profile as requested by the user and returns *None*.

5. Implementation Plan

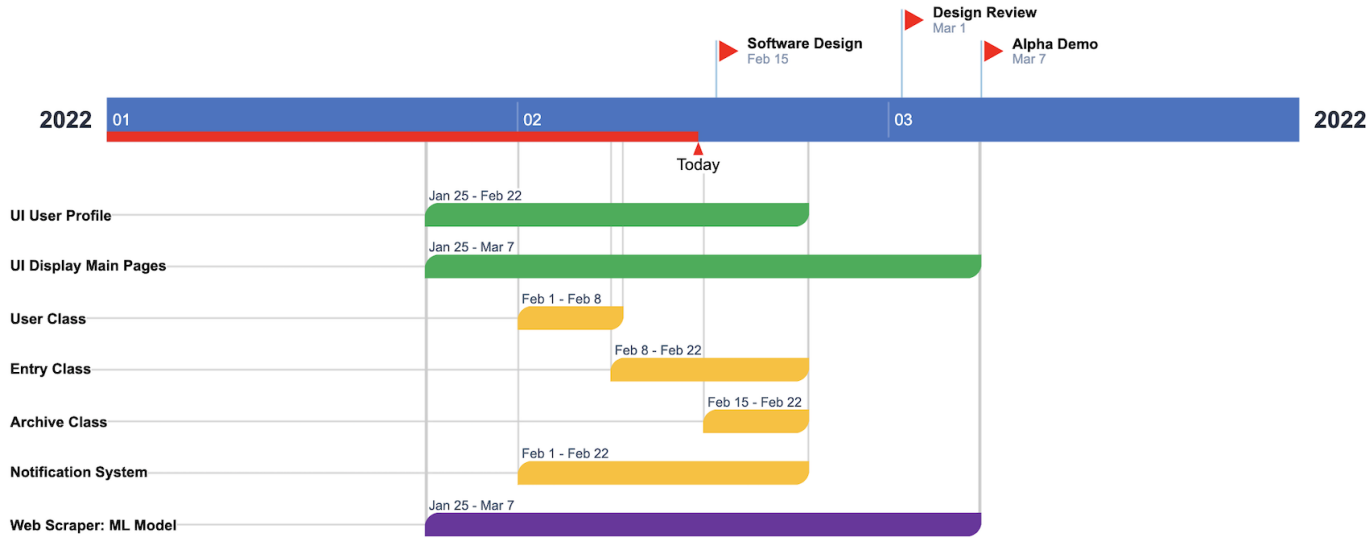

Figure 7. MapONE's progression plan.

The team plans to test and implement the above outlined systems to prepare for MapONE's final production in May 2022. Each stage of the progression plan will include design and test phases. The current project team's goal is to have a basis for all functionalities and operations by the alpha demo deadline, the week of March 7th. To achieve this, the UI must have a basic login page, main page, and user profile. The backend, specifically the database, must have user, entry, and archive classes. Lastly, the web scraper must be able to locate a subset of valid publications using a ML model.

In the current project phase, the team is creating the user account and publication entry (search engine) systems which must be implemented first as the foundation for the archive and notification systems. The UI and web scraper have been an ongoing development from the

previous year and will continue throughout the progression plan, specifically displaying the main UI pages and optimizing the ML model. After the alpha demo, a refinement period will occur until the project's release - final communication between the systems (UI, API/database, and web scraper). Once reviewed by the client, the team will work to optimize the workflow, presentation, and stability of the product.

6. Conclusion

The project team will continue developing MapONE, a single system for non-USGS planetary map metadata for final production in May 2022. The web application is a user-friendly interface that displays map metadata for a selected region of interest and connects to one database containing non-USGS publications. With this product, users will have access to view, save, and periodically automate searches to gather resources. This fits into the client's vision of a product capable of displaying scientific articles from other sources outside the USGS. The system will add to USGS's collection of map products while reducing the risk of researcher confirmation bias; by providing an easy outlet for scientists to explore both types of publications, USGS can further contribute to the planetary science community's research.

This document also discussed MapONE's three main modules - GUI, web scraper, and database - and their primary functions within the overall system architecture. Lastly, the team's project plan outlined the current project and the next steps for production. Overall, this document prepares the project team to implement all key software functionalities needed for the alpha demo.