MapONE

Final Report

May 5, 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 Final Report document is to summarize the project's development.

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.

2. Process Overview

To begin the project, the team defined certain rules and expectations. The team agreed to communicate over Discord and emails. All code material was stored on a GitHub repository. All team documents were stored on Google Drive. Team meetings were also held weekly on

Discord. Mentor meetings were held weekly in person. Lastly, client meetings were held weekly on Zoom. Additionally, the following roles and responsibilities were assigned to each team member:

Description	Assigned Role
Team Leader: Coordinates task assignments, ensures project progression, and leads team meetings. Responsible for all deliverable submissions.	Samantha Milligan
Customer Coordinator: Communicates and presents project updates to clients. Monitors customer satisfaction. Promotes customer collaboration.	Samantha Milligan Ricardo McCrary
Recorder: Ensures product features are documented. Collects notes on all team, mentor, and client meetings. Responsible for creating task reports.	Michael Nelson
Architect: This team member is primarily responsible for ensuring that core architectural decisions are followed during implementation**.**	Jake Stuck
Release Manager: Coordinates project versioning and branching, reviews and cleans up commit logs for accuracy and readability. Responsible for merging product changes. Ensures that any build tools can quickly generate a working release.	Michael Nelson
Coder: Produces code and implements product software features.	Samantha Milligan Michael Nelson Ricardo McCrary Jake Stuck

The team also agreed to conduct themselves in a professional manner at all stages of the product's development. This included responding to team messages within a 48-hour period and submitting individual deliverables sections 48 hours before the deadline.

3. Requirements

To begin development and software design, the team outlined MapONE's key system requirements. During client meetings, the team created the following list of **domain** requirements:

1. Login into an account on the USGS website.
2. View and filter planetary map publication metadata (source name, source link, map body, map scale, article title, author, and publication date).
3. Download all publication entries.
4. View and save search history results.
5. Automate searches periodically.
6. Receive notifications on new publications from automated search results.

At the time of product delivery, all requirements (except for requirement 3, see section 7 for more information) were implemented as features of MapONE. Additionally, the following **functional** requirements were included in the final delivery:

1. **User Account System**: A system that allows users to log into the application and access user privileges (creating automated searches and receiving email notifications).
2. **Search Engine**: A system that can search and filter publications based on map body or publication year.
3. **Web Scraper**: Configure the data extraction tool to locate and collect verified planetary map publications.
4. **Notification System**: A system that can send email notifications to users.

For **performance** requirements, the team had to ensure MapONE's interface was accessible to all users and that the web scraper was able to pull new publications on a weekly basis. Lastly, the

only key **environmental** constraint the team faced was that MapONE would be a web-based, open-source Python tool as requested by the client.

4. Architecture & Implementation

MapONE consists of three modules: a Graphical User Interface (GUI), backend Application Programming Interfaces (APIs), and a web scraper. These modules are divided into two domains, the application's frontend (GUI) and backend (APIs and web scraper). The domains run on two separate remote servers where the frontend completes user requests through API calls to the backend.

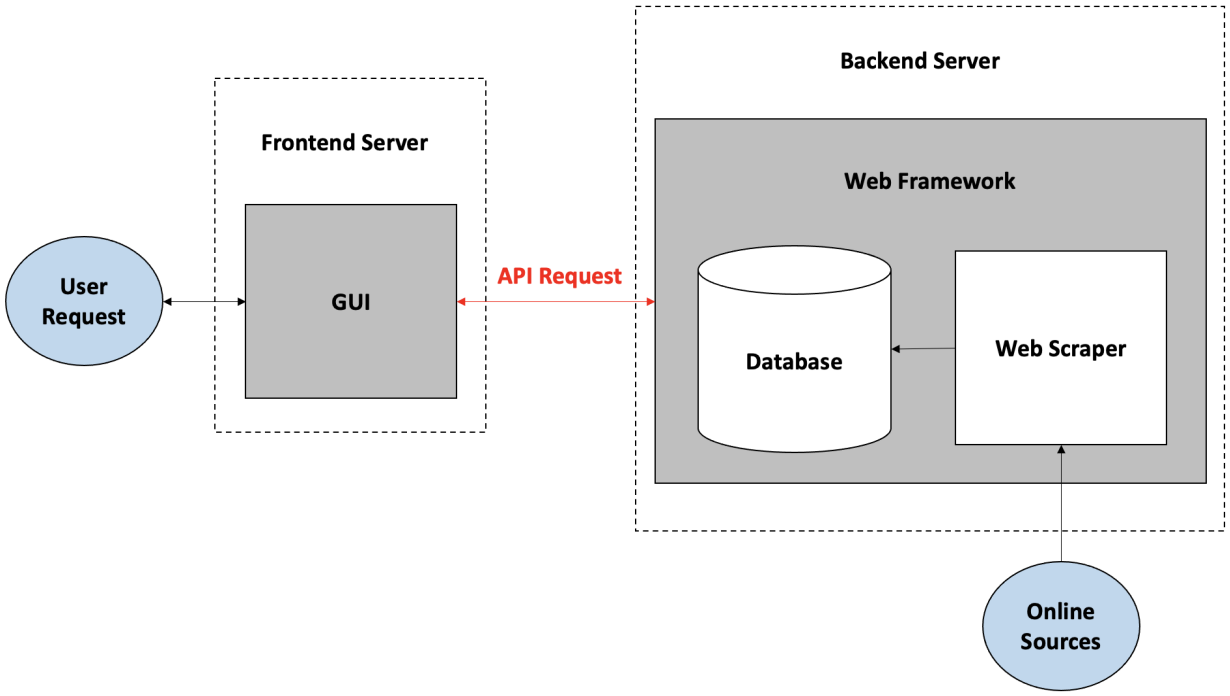

Figure 1. MapONE's software architecture.

The web scraper is configured to locate a series of online science journals and search a specific map body or keyword among published articles. Once the articles are gathered, the web scraper verifies if each publication contains a planetary map. If so, the source data is extracted and stored in MapONE's database.

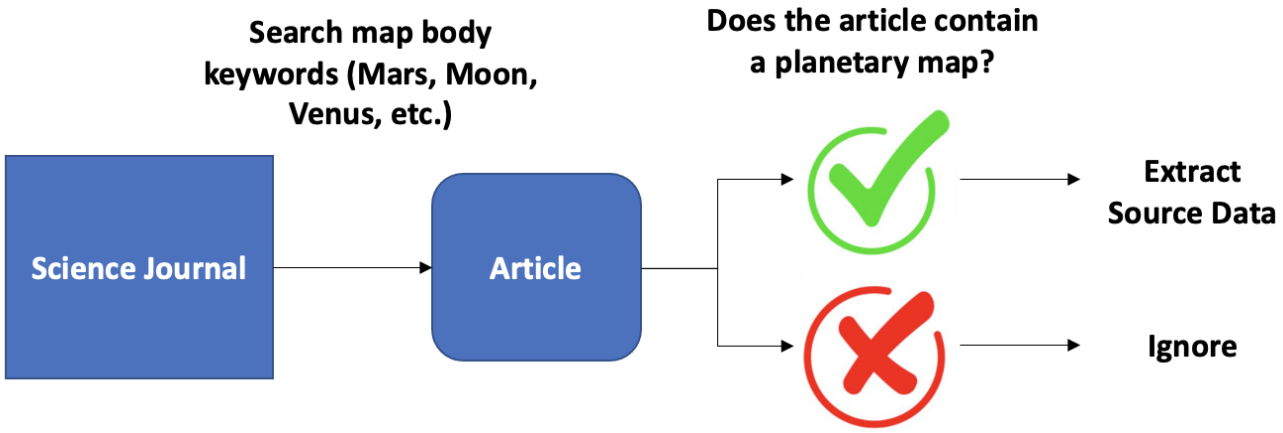

Figure 2. Web scraper workflow.

For technologies, the project team used Flutter, a frontend web-based software, to create the GUI. Django, a Python full-stack web development software, was used as the web framework which automatically sets SQLite as its default database structure. Lastly, Keras, a Python Machine Learning (ML) library, was used to create the web scraper. At the time of product delivery, the backend system was containerized using Docker, and both the frontend and backend servers currently run on Heroku, a hosting platform.

5. Testing

The project conducted three phases of software testing on MapONE. First, each module (GUI, backend APIs, and web scraper) had to pass **unit tests** to determine if each component could run independently from the rest of the system. The GUI used a mock testing suite to mimic the behavior of the backend to test UI elements without an actual API. The backend APIs' individual class functions also had to be tested separately. Three Django test cases were created to test user, entry, and archive functionalities. Lastly, the web scraper used AREPL, a software extension, to run tests on the web scraper's process, scan, and extract functionalities.

Next, the modules were tested together during **integration testing**. The GUI was checked to ensure successful API requests to the backend would display properly on the interface. Similarly, all backend APIs were tested under a Django test case which made real-time API requests. The

test case ensured all HyperText Transfer Protocol (HTTP) response codes and messages from the APIs were as expected. Lastly, the web scraper was tested to ensure scraped publications were displayed in the database and interface in real-time.

Lastly, MapONE's full system was tested by users during the **usability testing** phase. A user survey was conducted and distributed to members of the planetary science community. The survey allows users to offer feedback on the system's overall performance. Collectively, the users confirmed that the system displayed correct planetary map publications. However, the users recommended that the interface format be changed to a larger viewport (see section 7 for additional information).

6. Project Timeline

During the course of MapONE's development, the project team defined specific intervals for each module creation. During the first half, the project team created the following:

1. Interface's user profile and main pages
2. Backend's user, entry, and archive APIs
3. Web scraper's machine learning model to collect publication data

The second half of the development period included the frontend and backend deployment to Heroku as well as software testing and overall system refinement. MapONE was delivered to the clients the week of May 2nd (on schedule).

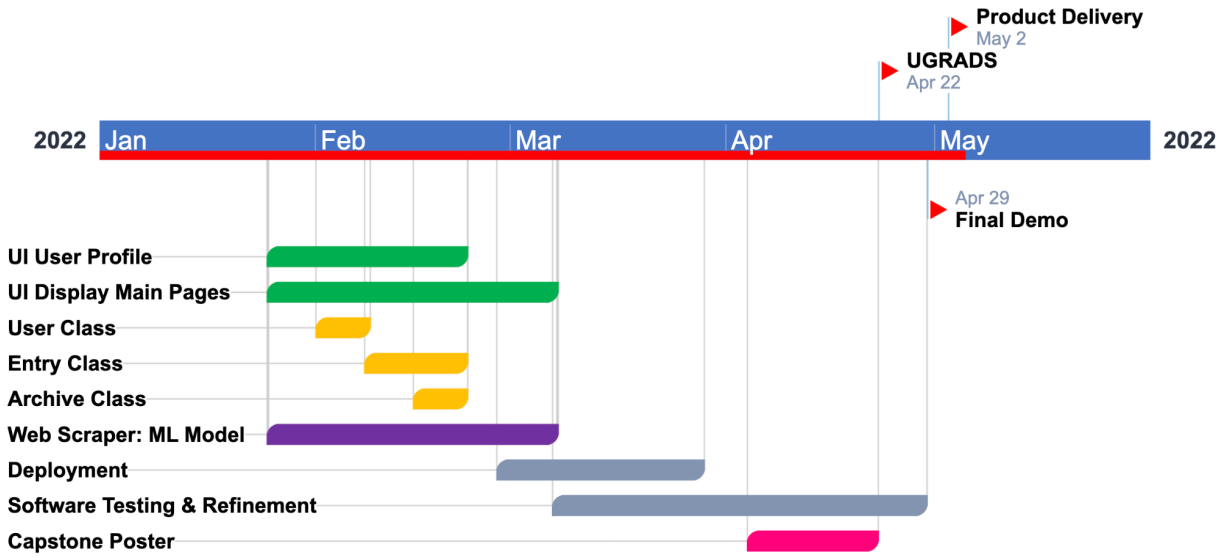

Figure 3. MapONE's project schedule.

7. Future Work

As previously mentioned in section 3, the only requirement not implemented in MapONE's system was the ability to download publications. To implement this feature, users should be able to export publications displayed on the main page to a Comma-Separated Values (CSV) format. This allows users to easily gather and store a set of publications for later use.

Also, given the results of the user survey (see section 5), MapONE's main page could be further improved. The current table format offers a small viewport of the publication data where only a few columns are in view at a time. This is difficult for users to read all source data without constantly scrolling back and forth across the main page.

8. Conclusion

This document outlines MapONE's development stages throughout the course of the project. Initially, the project team created team standards and outlined key functional and non-functional requirements of the system. As the product progressed, the team created the software design and conducted tests on each main module (GUI, backend APIs, and web scraper). From there, the

team was able to deliver the product to the clients. For future projects, users should be able to export publications along with improvements to MapONE's interface and table layout.

9. Appendix

This section serves as an appendix to the final report to discuss the development environment and toolchain used during MapONE's development.

9.1 Hardware

The majority of MapONE's development was done in macOS and Windows. The processing power of each system varied, but the average hardware of each system consisted of ~8 GB of memory and a 6-Core Intel processor. Although not required, it is recommended that a system meets this average for effective development.

9.2 Toolchain

The primary list of software tools consists of Python, Django, Flutter, Heroku, and Docker. A brief description of each tool and purpose in the system are listed below:

- **Python:** A general-purpose programming language used to develop the web scraper and backend API components.
- **Django:** A Python-based architecture used to manage the backend framework as well as the database that holds all of the web scraper data.
- **Flutter:** An open-source UI software used to create the frontend of MapONE's system.
- **Heroku:** A cloud hosting platform as a service, implemented to build and operate the system entirely on the cloud.
- **Docker:** A containerization tool utilized for easier deployment of the system.

9.3 Setup

For more detailed information on installation and configuration, please refer to the team's User Manual. At the time of product delivery, the frontend server should run at https://mapone-interface.herokuapp.com/ and the backend server at

https://mapone-api.herokuapp.com/. All files and systems discussed in this section can be accessed at https://github.com/samantha-milligan/MapONE. Basic setup instructions for the frontend and backend are listed here:

9.4 Frontend Installation:

1. **Install** Flutter at https://docs.flutter.dev/get-started/install depending on the operating system. Here, users can download all system requirements to run a Flutter application. Follow all installation instructions including downloading the latest release of Flutter SDK and updating the package source path. Additionally, macOS and Linux users can use Homebrew, a software package management system, to install Flutter at https://formulae.brew.sh/cask/flutter.
2. **Download** the frontend directory from GitHub, "mapone_frontend," onto the client's local machine.
3. **Change** into the frontend directory, "mapone_frontend," using a command-line tool.

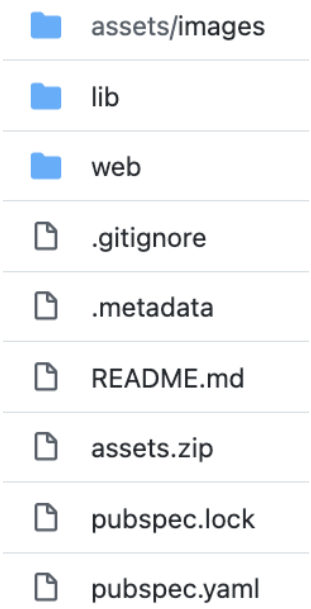

Figure 1. Frontend directory view.

4. **Run** *Flutter run* to run MapONE's interface on localhost. MapONE's interface should now be displayed on localhost on a pop-up browser (this will be automatically opened by the command).

5. **Make** changes to the source material as needed and repeat step 4 to see results.
6. **Install** Heroku at https://devcenter.heroku.com/articles/heroku-cli#install-the-heroku-cli depending on the operating system.
7. **Log** in to the existing Heroku account by running *heroku login* using a command-line tool.
8. **Run** *heroku git:clone -a mapone-interface* to clone the frontend server's source code to the local machine.
9. **Change** into the "mapone-interface" directory where it was cloned in step 8.
10. **Replace** the files in "mapone-interface" with the new changes in "mapone_frontend."
11. **Run** *git add .* and *git commit -m "write-message"* to update changes.
12. **Run** *git push heroku master* to push changes to https://mapone-interface.herokuapp.com/. The frontend should now be updated and deployed remotely.

9.5 Backend Installation:

1. **Install** Docker at https://docs.docker.com/get-docker/ depending on the operating system. Here, users can download all system requirements to run a Docker container.
2. **Install** Heroku (see step 6 in section 2.4).
3. **Download** the backend compressed file from GitHub, "mapone_backend_docker.zip," onto the client's local machine.
4. **Unzip** the file.
5. **Change** into the backend directory, "mapone_backend_docker/mapone-api/" using a command-line tool.

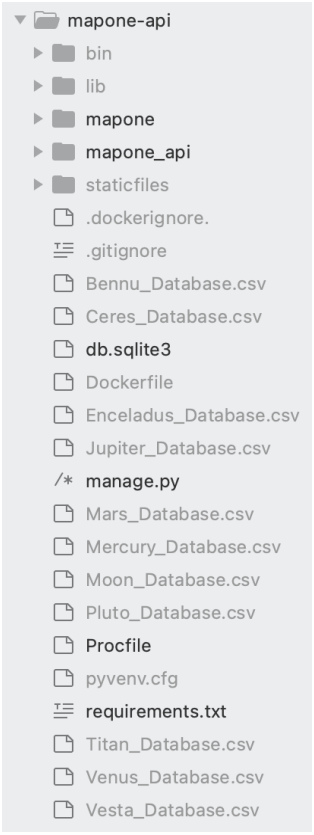

Figure 2. Backend directory view.

6. **Add** the password for EMAIL_HOST_PASSWORD in "mapone_backend_docker/mapone/settings.py". This will allow users to receive emails from gs-g-wr_astro_map_search@usgs.gov.
7. **Make** changes to the source material as needed.
8. **Run** *pip install -r requirements.txt* to install all necessary packages. For this step, it is recommended the client use a virtual environment. For more information, visit https://virtualenv.pypa.io/en/stable/.
9. **Run** *python3 manage.py runserver* to run the backend server locally at http://localhost:8000/.
10. **Log** in to the existing DockerHub account by running *docker login* using a command-line tool.
11. **Run** *docker container stop $(docker container ls -aq)* to stop all Docker containers as this is an existing container.
12. **Run** *docker container rm $(docker container ls -aq)* to delete all containers.
13. **Run** *docker rmi $(docker images -q)* to remove all Docker images.

14. **Run** *docker images -a* and *docker ps -a* to ensure container and image lists are empty.
15. **Run** *docker build -t mapone-api .* to create a new Docker image.
16. **Log** in to the existing Heroku account by running *heroku login* using a command-line tool.
17. **Run** *heroku container:login* to log in to the container.
18. **Run** *heroku container:push web* to update changes.
19. **Run** *heroku container:release web* to push changes to https://mapone-api.herokuapp.com/. The backend should now be updated and deployed remotely.
20. **Enter** https://mapone-api.herokuapp.com/timer/ into any browser to start the system's internal timer (see section 3.2 for more information).

9.6 Production Cycle

An IDE of choice is recommended for editing any of the material in this system. For reference, the project team used Visual Studio Code for Python-related material and Android Studio for the Flutter GUI. Following the steps above, users can disregard the installation process (assuming completion) and make changes to the source material as needed. Once changes have been made and deployed, please ensure all new material is pushed to the GitHub repository.