SSC21-P1-27

Lessons Learned from Development of Web-Based Mission Operations Software

Nathan Chow, Logan Pulley, Michael Lembeck
University of Illinois at Urbana-Champaign – Laboratory for Advanced Space Systems at Illinois
104 S. Wright St., Urbana, Illinois 61801; (703) 226 9277
nwchow2@illinois.edu

ABSTRACT

The Laboratory for Advanced Space Systems at Illinois (LASSI) is responsible for the development of, and mission operations for, a series of student built CubeSats. The Steven R. Nagel Mission Operations Center (MOC) provides the facilities and software systems required to track and communicate with the satellites. Mission operations software has been designed with a flexible and familiar user experience supporting not just traditional workstations but also mobile devices such as smartphones and tablets. This paper discusses the motivation behind choosing a web-based architecture for the scalable, full-stack design underlying the MOC software system. Also presented is the deliberate selection of mainstream languages, tools, and frameworks for containerization and web-based delivery – a philosophy that enables future enhancements and fosters maintainability. Processes that facilitate student workflow patterns such as comprehensive code reviews and development operations (DevOps) automation are presented.

INTRODUCTION

The number of CubeSats launched each year has continued to increase with the commercial sector driving much of the growth in the last decade.[1] Supporting this growth is decreasing launch costs due to commercialization and competition, the increase in commercial of the shelf parts, and the improvement of technology, such as power and control systems, that increase the capabilities of CubeSats.[2,3] Most missions utilizing the CubeSat platform are unique in their diverse objectives and relatively short mission life. Consequently, the mission operations software that support these missions must not only be robust to handle downlinked bus and payload data but also flexible enough to support the objectives of present and future CubeSat missions.

The mission operations center (MOC) software system currently under development within the University of Illinois at Urbana-Champaign's Aerospace Engineering Department's Laboratory for Advanced Space Systems at Illinois (LASSI) utilizes a web-based application architecture to meet the unique requirements of CubeSat mission operations. Earlier mission operations solutions supporting LASSI CubeSats were decentralized and dedicated to individual missions. The new MOC software system is designed with several primary goals: to incorporate all aspects of mission operations and control, from commanding to data visualization and analysis; support for traditional workstations as well as mobile device like smartphones and tablets to increase data accessibility; and maintainability, expandability, and flexibility to maximize capability and to support evolving requirements for all present and future LASSI CubeSats.

This paper discusses the features of web-based applications that carry out the primary goals of the MOC software system and presents a high-level description of the software architecture at its current stage of development. Also discussed is the development pipeline which strives to implement parts of the Agile-like design philosophy for continuous, rapid, and responsive development.

WEB APPLICATION FEATURES

Expandable and Flexible

LASSI has a number of CubeSats under development, some of which might potentially be operated simultaneously. A configurable MOC software system is required in order to avoid major rewrites for every mission.

In a web-based application, heavy duty data processing and storage is executed on a high-performance web server, while the less resource intensive data presentation is run on a browser on the user's machine.[4] This separation allows the independent development of front-end and back-end logic and allows for easier changes to the user interface and presentation when required. While the overall data storage method and

associated logic layer are expected to remain largely static between missions, the front-end and external applications interacting with and utilizing the data are dynamic and constantly evolving based on payloads and customer feedback. Web-based applications neatly separate the server-side back-end and client-side front-end operations.

A critical feature of front-end and back-end separation is a standard data access mechanism. Database access is performed through a REST API using a HTTP request, while live data is distributed from server to client through the WebSocket communications protocol. The usage of a standard API allows developers of server-external applications, such as downlink tools, like a parser or front-end interfaces, to develop their application without much knowledge of the inner workings of the server side layers.

Maintainable

Software maintainability is an important consideration in any large application. Maintenance issues arising from requirements changes, bugs, and feature additions contribute to a large portion of development time.[5,6] In a university setting, frequent personal changes to the MOC software development team can be challenging. Aspects of a web-based application helps minimize these problems.

LASSI's MOC software maximizes usage of mainstream and open-source languages, tools, and libraries. It is developed on the popular Django high-level Python web framework which provides much of the base code and services needed to run a web server. Front-end static files are written using the common trio of JavaScript, HTML, and CSS. These tools should be familiar to many web-developers, and, because they are open source, extensive documentation, tutorials, and discussion are publicly available online.

The separation of layers that makes web-applications flexible and expandable also makes them maintainable. It breaks down the application for more straightforward refactoring, bug tracking, and bug fixing. This also introduces more opportunities for unit and regression testing.

Accessible

Due to the prevalence of computer browsers, most people are accustomed to the look and feel of web applications. Many modern desktop applications use a web-based framework for their GUI. The MOC software system web page uses Bootstrap, a front-end framework library developed at Twitter, to control a modern and responsive user interface. It contains a large library of components designed for intuitive user interactions with both desktop and mobile browsers in mind.

Web-based applications also increases the accessibility of information. Data access and visualization through the MOC webpage are not restricted to consoles in the lab. Users with authorized accounts can access satellite data anywhere they have an internet connection and are attached to the university VPN. Other than a web browser, the user does not need to install additional software on their client computer or mobile device. This is valuable for student operators who are often relocating between semester breaks as well as during the COVID-19 pandemic when much work had to move remote.

ARCHITECTURE

Server-Side

The primary job of the MOC web server is to handle incoming telemetry downlink from the parser and handle WebSocket and HTTP requests from web page clients. Figure 1 shows a high-level block diagram of the current MOC server-side software architecture. The web server is built around the Django web framework. Models define PostgreSQL database tables and relationships, described as Python classes in the Django framework. The Django REST framework is used to build a web API for handling the HTTP requests from the parser, client web pages, or other external applications. Raw telemetry from the downlink chain is sent to the parser, which makes POST requests to the web server with timestamped data. The Django framework processes the POST requests and store the information in the database. It also pushes telemetry immediately to the client through the WebSocket connection. The client web page can also make HTTP requests for submitting form data or requesting telemetry stored in the database.

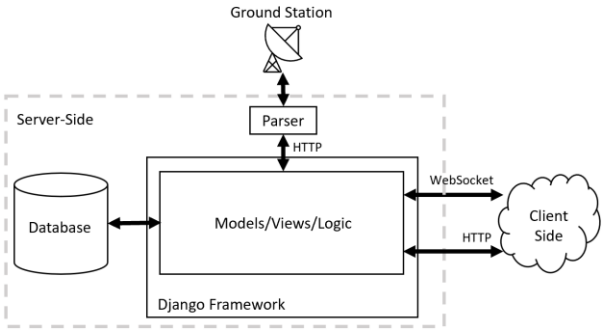

Figure 1: High-level diagram of the server-side architecture.

Client-Side

The client-side web page is the presentation layer and main interface between LASSI operators and data downlinked from CubeSats. It supports visualization of data from user selected satellite, subsystems, and time range. Data is currently viewable in scatterplot and table formats. A form page also allows operators to create alarms and set their trigger range for detecting and highlighting unexpected values of a subsystem.

The client-side architecture separates functionality into several layers to improve organization and maintainability. The UI layer consists of the functionality describing the user interactive elements. These include selection dropdowns, forms, tables, and charts.

The data transfer layer makes HTTP requests and receives HTTP responses. It also manages WebSocket connections and handles requests and responses through the socket. A cache is also located in the data transfer layer to minimize HTTP requests which are relatively slow. If a user requests previously requested data, the data transfer layer does not make a new HTTP request and instead provides the cached data.

The interface layer connects the UI layer to the data transfer layer. It formats data from the stored format to one that a corresponding element, such as a chart or table, can use.

A publish and subscribe (pub/sub) bus facilitates communication between the different layers. With this system, modules do not directly talk to each other and instead pass information, such as UI events and data requests, through messages. The pub/sub bus removes tightly integrated inter-modular communication, increasing the modularity and scalability of the front-end architecture.

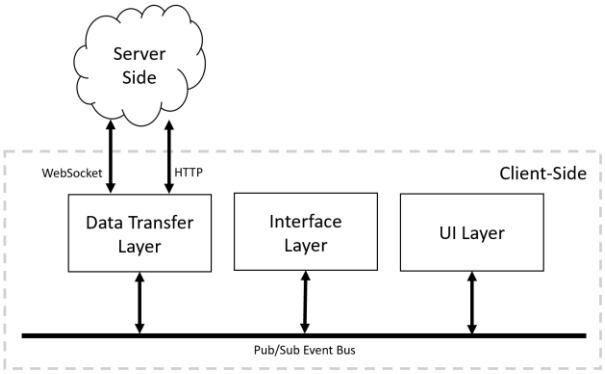

Figure 2: High-level diagram of the client-side architecture.

DEVELOPMENT OPERATIONS

Development Pipeline

Development on the MOC software system follows an Agile-like process. Source control, issue tracking, and automated CI/CD are primarily executed using GitLab tools. GitLab's issue tracking features are used to manage and track work on new features and bugs. Every issue receives its own merge request.

Figure 3 shows the typical development process for an issue. Issues move into code review following the complimentary and concurrent development and development testing phases. We introduced this step to improve overall code health and give more developers a greater understanding of the larger code base. When reviewers approve work on an issue and after it is merged into the development branch, GitLab triggers the automated CI/CD pipeline process. The pipeline runs automated tests and builds the images that are deployed to the development server. After ensuring the new software version is working in the development environment, the development branch is copied into the master branch and the pipeline pushes the images to the production server. If at any stage an error is found, the issue moves back into development and the process restarts.

Our development pipeline evolves as the project grows. Our initial process involved only development testing. As the code base and team grew, code review, automated testing, and linting were introduced to ensure quality and robustness. Our development pipeline is now a critical part of the speedy and consistent delivery of the MOC software system.

Containerization

MOC utilizes Docker to containerize MOC applications. The platform streamlines the process of building, deploying, and running applications and services.[7] During the build stage of the CI/CD pipeline, applications and services that are a part of the MOC software system are packaged into images and pushed to either the development or production server Docker registries. From there, the latest images are pulled, created into containers, and run.

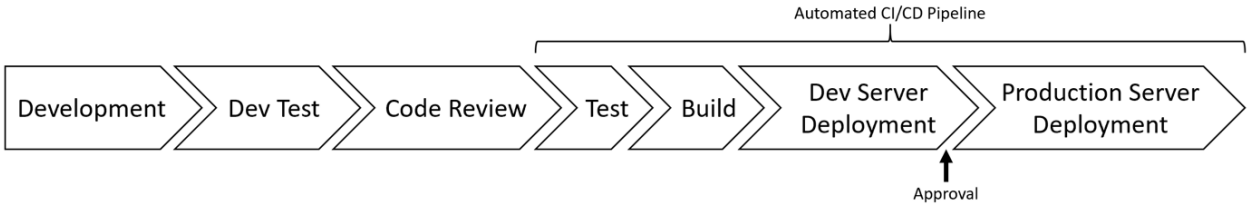

Figure 3: Typical development and CI/CD pipeline.

ROADMAP

Development of the MOC software system is a work in progress with ongoing development of new features and bug fixes. While we have not yet implemented the full intended feature set, the current version is ready to support operations of LASSI's upcoming missions. The infrastructure for handling downlinked telemetry from the CubeSat bus systems and payloads, as discussed in the section on server-side architecture, is in place. On the client-side web page, a data visualizations page for viewing requested telemetry in scatterplot and table format is operational.

Ongoing and future work include optimizations on existing features such as the data visualizations page and better support for viewing and interaction on smartphones and tablets. An interface allowing LASSI operators to send spacecraft commands from the MOC software system web page is also planned. Work to support mission-specific requirements will also continue alongside the development of enhancements and new features.

CONCLUSION

Building the MOC software system as a web-based application allows it to be flexible, maintainable, and accessible. The separation of layers and modular design eases the process of adding new capability and refactoring old code. The web page allows data access to any device with an internet browser and the REST API gives external applications a simple and standard way to interact with data.

Our incremental development philosophy means that the MOC software system is able to support LASSI's upcoming missions as simultaneous development continues on implementing features to place a centralized, modern, and easy-to-use mission operations and control solution at the hands of LASSI's operators.

References

1. Shkolnik, E.L. (2018). On the verge of an astronomy CubeSat revolution. *Nature Astronomy, 2,* 374–378. https://doi.org/10.1038/s41550-018-0438-8
2. Jones, H.W. (2018). The recent large reduction in space launch cost. *Proceedings of the 48th International Conference on Environmental Systems.* NASA Technical Reports Server. https://ntrs.nasa.gov/citations/20200001093
3. Liddle, J.D., Holt, A.P., Jason, S.J., O'Donnell K. A., Stevens E. J. (2020). Space science with CubeSats and nanosatellites. *Nature Astronomy, 4,* 1026–1030. https://doi.org/10.1038/s41550-020-01247-2
4. Bourne, K.C. (2014). Chapter 2: Design. In K.C. Bourne (Ed.), *Application administrators handbook: Installing, updating, and troubleshooting software* (pp. 12-19). ScienceDirect. https://doi.org/10.1016/B978-0-12-398545-3.00002-9
5. Velmourougan, S., Dhavachelvan, P., Baskaran, R., Ravikumar, B. (2014). Software development life cycle model to improve maintainability of software applications. *Proceedings of the 4th International Conference on Advances in Computing and Communications* (pp. 270-273). IEEE. https://doi.org/10.1109/ICACC.2014.71
6. Saraiva, J. (2013). A roadmap for software maintainability measurement. *Proceedings of the 35th International Conference on Software Engineering* (pp. 1453-1455). IEEE. https://doi.org/10.1109/ICSE.2013.6606742
7. Preeth, E. N., Mulerickal, J. P., Paul B., Sastri, Y. (2015). Evaluation of Docker containers based on hardware utilization. *Proceedings of the International Conference on Control Communication & Computing India* (pp. 697-700). IEEE. https://doi.org/10.1109/ICCC.2015.7432984