Module Interface Specification for Software EngineeringMCT

Team #12, Lower Earth Orbiters
Diamond Ahuja
Rishi Vaya
Buu Ha
Dhruv Cheemakurti
Umang Rajkarnikar

April 4, 2024

1 Revision History

Date	Version	Notes
Jan. 18, 2024	1.0	Finished MIS doc
April 3, 2024	2.0	Updated documentation to reflect current implementation of the application and updated MIS based on feedback

2 Symbols, Abbreviations and Acronyms

See SRS Documentation at this link. This section records information for easy reference.

2.1 Abbreviations and Acronyms

symbol	description
AC	Anticipated Change
DAG	Directed Acyclic Graph
M	Module
MG	Module Guide
OS	Operating System
R	Requirement
SC	Scientific Computing
SRS	Software Requirements Specification
UC	Unlikely Change
FIFO	First In First Out
MCT	Mission Control Terminal
SSL	Secure Socket Layer

Contents

3 Introduction

The following document details the Module Interface Specifications for MCT.

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at this link.

4 Notation

The structure of the MIS for modules comes from Lower Earth Orbiters, with the addition that template modules have been adapted from Dr. Spencer Smith's capstone template repository. The mathematical notation comes from Chapter 3 of David Gries and Fred B. Schneider: A Logical Approach to Discrete Math. For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1|c_2 \Rightarrow r_2|...|c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by MCT.

Data Type	Notation	Description
character	char	a single symbol or digit
integer	$\mathbb{Z}$	a number without a fractional component in (-∞, ∞)
natural number	$\mathbb{N}$	a number without a fractional component in [1, ∞)
real	$\mathbb{R}$	any number in (-∞, ∞)
Command	Object	Object representing command sequence record from database.

The specification of MCT uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, MCT uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

Level 1	**Level 2**
Hardware-Hiding	N/A
Behaviour-Hiding	Satellite Module Schedule Module User Module Log Module Database Module Authentication Module BackendService Module
Software Decision	SatelliteQueue Module

Table 1: Module Hierarchy

6 MIS of Database Module

6.1 Overview

This module utilizes the date and collectionType, formatted to align with the MongoDB database, to locate and retrieve stored data from the specified location. Additionally, it has the capability to store provided data using the inputs of date, collectionType, and data. Lastly, it can store edited data by utilizing date, collectionType, id, and data as inputs.

6.2 Uses

The database module is used to manage all of the data stored for the satellites, users, commands, logs and scheduling. This involves addition of new data, editing/modifying existing data and deleting values.

6.3 Syntax

6.3.1 Exported Constants

N/A

6.3.2 Exported Access Programs

N/A

6.4 Semantics

6.4.1 State Variables

N/A

6.4.2 Environment Variables

Name	Type	Description
DB_USER	String	Login information for MongoDB Cluster
DB_PASSWORD	String	Login information for MongoDB Cluster
DB_URI	String	Connection path to MongoDB instance

6.4.3 Assumptions

MongoDB is an ACID compliant database and ensures persisted data is consistent.

6.4.4 Access Routine Semantics

N/A

6.4.5 Local Functions

Name	Input	Output	Description
insert()	{databaseId: string, values: list of strings}	N/A	Adds new values to the database
update()	{databaseId: string, rowselector: string, values: list of strings}		Updates an existing entry in the specified database
read()	{databaseId: string, selector: string}	{result: list of strings}	Returns values from the specified database based on the selectors and filter provided.
delete()	{databaseId: string, rowselector: string}	N/A	Deletes an entry from the specified database.

Formal Specification - insert()
Let D be the set representing our database, where each element $r_i \in D$ is a record within the database. The state of the database before any insert operation can be represented as:

$$D = \{r_1, r_2, \ldots, r_n\}$$

The *insert* operation on the database can be formalized as adding a new record r_{new} to the set D. This operation results in a new set D':

$$D' = D \cup \{r_{\text{new}}\}$$

Hence, after the insert operation, the state of the database is represented as:

$$D' = \{r_1, r_2, \ldots, r_n, r_{\text{new}}\}$$

Formal Specification - update()
Assume our database D is a set of records, where each record is represented as a tuple $(id, data)$, with id being a unique identifier and $data$ representing the record's content. The state of D before the update operation can be defined as:

$$D = \{(id_1, data_1), (id_2, data_2), \ldots, (id_n, data_n)\}$$

To update a record with identifier id_i to new data $data_i'$, we define the update operation as follows:

$$D' = \{(id, data) \in D | id \neq id_i\} \cup \{(id_i, data_i')\}$$

This operation results in a new state of the database D' where:

$$D' = \{(id_1, data_1), \ldots, (id_i, data_i'), \ldots, (id_n, data_n)\}$$

After the update operation, D' includes the updated record $(id_i, data_i')$ while retaining all other records unchanged.

Formal Specification - read()
Given a database D consisting of records, where each record is a tuple $(id, data)$, the read operation for a record with a specific id_i can be formalized as follows:

$$\text{read}(id_i) = data_i \quad \text{for} \quad (id_i, data_i) \in D$$

The operation retrieves $data_i$ corresponding to the unique identifier id_i. If the record exists in D, the output is $data_i$; otherwise, the operation returns a null or error indicating that the record does not exist.

Formal Specification - delete()
Consider D to be the set representing our database, with each element being a record defined as a tuple $(id, data)$. The state of the database before any delete operation is:

$$D = \{(id_1, data_1), (id_2, data_2), \ldots, (id_n, data_n)\}$$

To delete a record with identifier id_i from D, we formalize the delete operation as removing $(id_i, data_i)$ from D, resulting in a new set D':

$$D' = D - \{(id_i, data_i)\}$$

Thus, the state of the database after the delete operation is represented without the record $(id_i, data_i)$:

$$D' = \{(id_1, data_1), \ldots, \nexists (id_i, data_i), \ldots, (id_n, data_n)\}$$

This operation ensures that the specified record is no longer part of the database post-deletion.

7 MIS of Log Module

7.1 Overview

The Log module implements logging functionality which displays the runtime logs of the commands sent to a satellite and its response.

7.2 Uses

The log module is used to view information regarding the commands transmitted to a satellite. This information contains data pertaining to the commands sent during a particular schedule, their run time and the response received.

7.3 Syntax

7.3.1 Exported Constants

N/A

7.3.2 Exported Access Programs

Name	Input	Output	Exceptions
/getLogs	{satelliteId: string}	{satelliteId: string, commandId: string, message: string, createdAt: dateTime}	N/A

7.4 Semantics

7.4.1 State Variables

N/A

7.4.2 Environment Variables

Name	Type	Description
DB_USER	String	Login information for MongoDB Cluster
DB_PASSWORD	String	Login information for MongoDB Cluster
DB_URI	String	Connection path to MongoDB instance

7.4.3 Assumptions

MongoDB is an ACID compliant database and ensures persisted data is consistent.

7.4.4 Access Routine Semantics

- **/getLogs**
  - *Input:* `satelliteId`

- *Output:* Logs- { satelliteId: string, commandId: string, message: string, createdAt: dateTime }
- *Semantics:* This function fetches and returns all log entries associated with the given `satelliteId`. The returned `logs` are in a format that includes the command response, the time it was sent, and the time it was updated.

7.4.5 Local Functions

Name	Input	Output	Description
fetchLogs()	{satelliteId: string}	{satelliteId: string, commandId: string, message: string, createdAt: dateTime}	Returns the logs associated with every command for the specified satellite.

Formal Specification - fetchLogs()

Consider our logging system as a database L, where each log entry is represented by a tuple $(satelliteId, logEntry)$, with $satelliteId$ identifying the satellite and $logEntry$ representing the content of the log. The complete set of logs is represented as:

$$L = \{(satelliteId_1, logEntry_1), (satelliteId_2, logEntry_2), \ldots, (satelliteId_n, logEntry_n)\}$$

The *fetchLogs* operation for a specific satellite ID, $satelliteId_i$, can be formalized as retrieving all log entries associated with $satelliteId_i$:

$$\text{fetchLogs}(satelliteId_i) = \{logEntry \mid (satelliteId, logEntry) \in L \text{ and } satelliteId = satelliteId_i\}$$

This operation results in a subset of L, L', containing only the logs related to $satelliteId_i$:

$$L' = \{logEntry_1, logEntry_2, \ldots, logEntry_m\} \quad \text{for all} \quad (satelliteId_i, logEntry) \in L$$

Where:

- L' is the set of all log entries related to $satelliteId_i$.
- m is the number of logs related to $satelliteId_i$.

8 MIS of Satellite Module

8.1 Overview

The Satellite module is responsible for all information related to a satellite, including calculating positional data, future overpasses, and generating polar plot data.

8.2 Uses

The satellite module is used to provide relevant information regarding a satellite, including detailed information regarding its position, future overpasses, and relevant polar plot data for each overpass.

8.3 Syntax

8.3.1 Exported Constants

N/A

8.3.2 Exported Access Programs

Name	Input	Output	Exceptions
/getSatelliteInfo	noradId: string	positionEci: $\mathbb{R}$, velocityEci: $\mathbb{R}$, longitude: $\mathbb{R}$, latitude: $\mathbb{R}$, height: $\mathbb{R}$, azimuth: $\mathbb{R}$, elevation: $\mathbb{R}$	N/A
/getPolarPlotData	startDate: Date, endDate: Date	[azimuth: $\mathbb{R}$, elevation: $\mathbb{R}$]	N/A
/getMaxElevation	startDate: Date, endDate: Date	maxElevation: $\mathbb{R}$	N/A
/getNextPasses	startDate: Date, endDate: Date	type: string, time: string, azimuth: $\mathbb{R}$, elevation: number	N/A

8.4 Semantics

8.4.1 State Variables

Name	Type	Description
tleLine1	String	TLE information for satellite tracking
tleLine2	String	TLE information for satellite tracking
defaulttleLine1	String	Default TLE information for satellite tracking
defaulttleLine2	String	Default TLE information for satellite tracking

observerGd	{
	longitude: Number
	latitude: Number
	height: Number
	}
		Observer location data

8.4.2 Environment Variables

Name	Type	Description
SPACE_TRACK_USERNAME	String	Login information for SpaceTrack API
SPACE_TRACK_PASSWORD	String	Login information for SpaceTrack API

8.4.3 Assumptions

Assuming that external APIs (Spacetrack) and libraries (SGP4) are always accurate for measurements.

8.4.4 Access Routine Semantics

Name	Transition	Output	Description	Exceptions
/getSatelliteInfo	Retrieves the current state information of the satellite.	An object containing satellite's position and velocity in ECI coordinates, longitude, latitude, height, azimuth, and elevation.	Accesses the latest telemetry data for the satellite and computes its current state, including position in Earth-Centered Inertial (ECI) coordinates and ground track information.	N/A
/getPolarPlotData	Generates data for a polar plot of the satellite's overpasses.	An array of objects, each containing the azimuth and elevation of the satellite during an overpass.	Calculates the satellite's position relative to the observer's location on Earth for a series of future overpasses between the specified start and end dates.	N/A
/getMaxElevation	Calculates the maximum elevation of the satellite's overpass.	A real number indicating the maximum elevation angle.	Determines the highest point in the satellite's trajectory, relative to the observer's horizon, for the next overpass.	N/A
/getNextPasses	Retrieves information on the next passes of a satellite within a specified date range.	A collection of objects each containing the type, time, azimuth, and elevation of a pass.	Computes the upcoming overpasses of a satellite based on its orbital data and the provided date range, returning detailed pass information.	N/A

8.4.5 Local Functions

Name	Input	Output	Description
getSatelliteInfo()	(date : Date, tleLine1: string, tleLine2: String)	{ positionEci : number, velocityEci : number, longitude : number, latitude : number, height : number, azimuth : number, elevation : number, rangeSat : number, }	Returns relevant satellite information given a certain date. Used by /getSatelliteInfo, /getPolarPlotData, /getMaxElevation.

Formal Specification - getSatelliteInfo()
Assume the satellite information system is retrieved from an online database S, where each record is a tuple containing satellite identification and its orbital parameters:

$$S = \{(id_1, \boldsymbol{O}_1), (id_2, \boldsymbol{O}_2), \ldots, (id_n, \boldsymbol{O}_n)\}$$

Here, id_i represents the unique identifier of a satellite (noradID), and $\boldsymbol{O}_i$ represents the set of orbital parameters for satellite i, such as position ($\boldsymbol{p}$), velocity ($\boldsymbol{v}$), and other relevant state variables.

The *getSatelliteInfo* operation, for a specific satellite ID, id_i, is formalized as retrieving the orbital parameters $\boldsymbol{O}_i$ associated with id_i:

$$\text{getSatelliteInfo}(id_i) = \boldsymbol{O}_i \quad \text{for} \quad (id_i, \boldsymbol{O}_i) \in S$$

The output $\boldsymbol{O}_i$ can be further detailed as:

$$\boldsymbol{O}_i = \{\text{position } \boldsymbol{p}_i, \text{velocity } \boldsymbol{v}_i, \ldots\}$$

This operation effectively returns the current state of the satellite, including its position, velocity, and other specified parameters, crucial for tracking and communication purposes.

9 MIS of Schedule Module

9.1 Overview

The Schedule module is responsible for all logic pertaining to storing, updating, and executing command sequence(s) for a satellite target.

9.2 Uses

The Schedule module is used to schedule execution of command sequences for a satellite system during a specified overpass. This includes reading, adding, and updating command sequences as well as executing scheduled commands.

9.3 Syntax

9.3.1 Exported Constants

N/A

9.3.2 Exported Access Programs

Name	Input	Output	Exceptions
/schedule/addCommand	scheduleId: string, satelliteId: string, userId: string, commandSequence: string	command: Command	Ensure command sequence is included in both user and satellite system's permission list criteria.
/schedule/updateCommand	commandId: string, scheduleId: string, satelliteId: string, userId: string, commandSequence: string	command: Command	Ensure command sequence is included in both user and satellite system's permission list criteria.
/schedule/cancelCommand	commandId: string, userId: string, scheduleId: string	void	Ensure the user has required permissions to cancel the command. Ensure the overpass has not passed.
/getNextPasses	startDate: Date, endDate: Date	type: string, time: string, azimuth: $\mathbb{R}$, elevation: number	N/A

9.4 Semantics

9.4.1 State Variables

Name	Type	Description
overpassQueueMap	[key:string]: SatelliteQueue	A hashmap of key-value pairs. Each key is the ID of a satellite record and corresponds to a value of type SatelliteQueue, which represents the list of command sequences to be executed in the next overpass for a satellite system.

9.4.2 Environment Variables

Name	Type	Description
DB_USER	string	Login information for MongoDB Cluster
DB_PASSWORD	string	Login information for MongoDB Cluster
DB_URI	string	Connection path to MongoDB instance

9.4.3 Assumptions

- MongoDB is an ACID compliant database and ensures persisted data is consistent.

9.4.4 Access Routine Semantics

Name	Transition	Output	Description	Exceptions
executeOverpass()	Indexes state variable overpassQueueMap, then unloads a queue of type SatelliteQueue. Then, loads queue with command sequences scheduled for the satellite's next overpass.	N/A	Executes all command sequences scheduled for the overpass in a FIFO order and records logs of the responses in the database	If a connection cannot be established with the satellite system, the indexed queue shall not be unloaded and therefore, will not be executed.
/schedule /addCommand	Adds a command to the schedule of the satellite.	A `Command` object representing the scheduled command.	Adds a command to a schedule list	If the command sequence is not in the user and satellite system's permission list.
/schedule /updateCommand	Updates an existing command in the satellite's schedule.	A `Command` object representing the updated scheduled command.	Replaces the existing command sequence with the new one provided, while maintaining its scheduled execution time.	If the command sequence is not in the user and satellite system's permission list.
/schedule /cancelCommand	Cancels a command from the satellite's schedule.	`void` (no output)	Removes the specified command from the schedule	If the user does not have permission to cancel the command or if the overpass has passed.
/getNextPasses	Retrieves information on the next passes of a satellite within a specified date range.	A collection of objects each containing the type, time, azimuth, and elevation of a pass.	Computes the upcoming overpasses of a satellite based on its orbital data and the provided date range, returning detailed pass information.	N/A

9.4.5 Local Functions

Name	Input	Output	Description
executeOverpass()	satelliteId: string, overpassId: string	void	Executes all command sequences scheduled for the overpass in a FIFO order and records logs of the responses in the database

Formal Specification - executeOverpass()

The *executeOverpass* operation is crucial for managing the execution of scheduled commands for a satellite during its overpass. Consider the set C representing all scheduled commands for a given overpass, where each command c_i has associated parameters indicating its scheduled execution time, target satellite, and specific action:

$$C = \{c_1, c_2, \ldots, c_n\}$$

For a specific overpass identified by $overpassId$, the *executeOverpass* operation is formalized as sequentially executing each command $c_i \in C$ related to that overpass. This process can be represented as a function f applied to the set of commands C, resulting in a new state C' reflecting the execution status of each command:

$$C' = f(C) = \{execute(c_i) \mid c_i \in C\}$$

Where the function $execute(c_i)$ denotes the execution of command c_i, altering its state to reflect completion or failure based on the execution outcome. The set C' represents the updated state of scheduled commands post-execution, with each element indicating the result of its execution process (e.g., success, failure, pending).

The operation ensures that all commands scheduled for the specific overpass are executed in accordance with their scheduling parameters and that their execution results are accurately captured:

$$executeOverpass(overpassId) = C' \quad \text{for} \quad overpassId \,\text{associated with}\, C$$

This process underlines the system's ability to handle scheduled commands efficiently, ensuring that each command intended for execution during a satellite's overpass is processed accordingly.

10 MIS of SatelliteQueue Module

10.1 Overview

The SatelliteQueue module is responsible for managing future execution of command sequences for all satellite systems.

10.2 Uses

The SatelliteQueue module is used to store execution of command sequences for a satellite system during a specified overpass.

10.3 Syntax

10.3.1 Exported Constants

N/A

10.3.2 Exported Access Programs

N/A

10.4 Semantics

10.4.1 State Variables

Name	Type	Description
overpassQueueMap	[key:string]: SatelliteQueue	A hashmap of key-value pairs. Each key is the ID of a satellite record and corresponds to a value of type SatelliteQueue, which represents the list of command sequences to be executed in the next overpass for a satellite system.

10.4.2 Environment Variables

N/A

10.4.3 Assumptions

- Node.js is single-threaded and does not permit two requests to modify the state variable, overpassQueueMap to occur simultaneously.

10.4.4 Access Routine Semantics

Name	Transition	Input	Output	Description
push()	Modifies the overpassQueueMap and adds a command sequence to the queue of type SatelliteQueue.	{commandId: string, userId: string, createdAt: dateTime}	N/A	Adds a command sequence to the queue of type SatelliteQueue.
pop()	Modifies the overpassQueueMap and removes the top item from the queue of type SatelliteQueue.	N/A	{commandId: string, userId: string, createdAt: dateTime}	Removes the top command sequence from the queue of type SatelliteQueue.

Formal specification - push()

The *push* operation adds a new command to the queue of scheduled commands for a satellite. Let Q represent the queue of commands, where $Q = \{c_1, c_2, \ldots, c_n\}$ and each c_i is a command scheduled for execution.

The push operation, for a new command c_{new}, can be formalized as appending c_{new} to the end of the queue Q:

$$Q' = Q + \{c_{\text{new}}\}$$

After the operation, the updated queue Q' represents the original queue with c_{new} added as the last element:

$$Q' = \{c_1, c_2, \ldots, c_n, c_{\text{new}}\}$$

Formal Specification - pop()

The *pop* operation removes the first command from the queue Q and returns it. This operation is represented as removing c_1 from Q:

$$(c_{\text{pop}}, Q') = \text{pop}(Q)$$

Where c_{pop} is the command being removed and Q' is the queue after removal:

$$Q' = \{c_2, c_3, \ldots, c_n\}, \quad \text{and} \quad c_{\text{pop}} = c_1$$

The pop operation thus dequeues the first command for execution or processing and updates the queue to reflect this removal.

10.4.5 Local Functions

N/A

11 MIS of User Module

11.1 Overview

The User module is responsible for storing information about users in the MIST application. This includes creating new users and updating information for existing users.

11.2 Uses

The User Module in the MIST application is designed to manage user-related data. Its primary functions include creating new users, updating information for existing users and getting a list of all the current operators.

11.3 Syntax

11.3.1 Exported Constants

N/A

11.3.2 Exported Access Programs

Name	Input	Output	Exceptions
/createUser	{Email: string, role: string}	New user	N/A
/getAllOperators	N/A	Returns all the operators	N/A
/updateOperatorRole	{userId: string, adminId: string, role: string	Updates the user's role	N/A

11.4 Semantics

11.4.1 State Variables

N/A

11.4.2 Environment Variables

Name	Type	Description
DB_USER	String	Login information for MongoDB Cluster
DB_PASSWORD	String	Login information for MongoDB Cluster
DB_URI	String	Connection path to MongoDB instance

11.4.3 Assumptions

MongoDB is an ACID compliant database and ensures persisted data is consistent.

11.4.4 Access Routine Semantics

Name	Transition	Output	Description	Exceptions
/createUser	Registers a new user in the system.	Confirmation of the creation of a new user.	Takes an email and a role, and creates a new user account assigning the specified role.	N/A
/getAllOperators	Retrieves a list of all operator accounts.	A list containing all the operators in the system.	Queries the database for all users with an operator role and returns their details.	N/A
/updateOperatorRole /:userId	Updates the role of an existing operator.	Confirmation of the updated role for the user.	Accepts a userId and a new role, and updates the specified user's role in the system's records.	N/A

11.4.5 Local Functions

N/A

12 MIS of Authentication Module

12.1 Overview

This module will employ the user's email and password for authentication purposes and to fetch their complete information, a prerequisite for accessing the satellite and scheduling

data.

12.2 Uses

The Authentication Module is a key part of the system, using the user's email and password to confirm their identity. After verification, it allows access to important features like satellite and scheduling data, requiring the user's complete information. This module ensures that only authorized users can interact with the system, enhancing security and controlling access to different functions.

12.3 Syntax

12.3.1 Exported Constants

N/A

12.3.2 Exported Access Programs

N/A

12.4 Semantics

12.4.1 State Variables

Name	Type	Description
email	String	The email address that operator uses to login
password	String	The operator's password associated with their email

12.4.2 Environment Variables

Input Name	Input Type	Description
DB_USER	String	Login information for MongoDB Cluster
DB_PASSWORD	String	Login information for MongoDB Cluster
DB_URI	String	Connection path to MongoDB instance

12.4.3 Assumptions

- MongoDB is an ACID compliant database and ensures persisted data is consistent.
- OAuth0 is the service used for authentication

12.4.4 Access Routine Semantics

N/A

12.4.5 Local Functions

N/A

13 MIS of WebSocket Module

13.1 Overview

The WebSocket module provides a real-time communication channel between the MIST application and the ground station. It is responsible for establishing a secure WebSocket connection, handling incoming messages, and sending data to the ground station.

13.2 Uses

The WebSocket Module is designed to enable the MIST application to maintain a persistent connection with the ground station, allowing for bi-directional communication. This module is primarily used for sending command sequences to the ground station and receiving acknowledgments or data in return.

13.3 Syntax

13.3.1 Exported Constants

N/A

13.3.2 Exported Access Programs

Name	Input	Output	Exceptions
/connect	N/A	WebSocket connection instance	N/A
/sendData	{Command: string}	Acknowledgment or response in term of string based on the Command sent from ground station	WebSocket not connected
/disconnect	N/A	Confirmation of disconnection	N/A

13.4 Semantics

13.4.1 State Variables

Name	Type	Description
clientSocket	WebSocket	The active WebSocket connection

13.4.2 Environment Variables

Name	Type	Description
KEY_CERT	File Path	Path to the SSL certificate file
KEY_PATH	File Path	Path to the SSL key file

13.4.3 Assumptions

- The ground station is continuously running and can accept WebSocket connections.
- The SSL certificate and key are valid and have not expired.
- The environment variables for the SSL certificate and key paths are set correctly.

13.4.4 Access Routine Semantics

Name	Transition	Output	Description	Exceptions
/connect	Establishes a connection to the ground station.	A WebSocket connection instance.	Initiates a WebSocket connection using the provided SSL certificate and key.	N/A
/sendData	Sends a command sequence through the open WebSocket connection.	Acknowledgment or response in term of string based on the Command sent from ground station	Transmits a command to the ground station and waits for a response.	WebSocket not connected
/disconnect	Closes the WebSocket connection.	Confirmation of disconnection.	Terminates the WebSocket connection with the ground station.	N/A

13.4.5 Local Functions

N/A

14 MIS of Backend Service

14.1 Overview

This module bridges the front end user interface with all the other aforementioned modules that deal with the functionality of the application. It oversees the initiation of the application's backend processes.

14.2 Uses

The Backend Service module is responsible for ensuring the information from backend modules that deal with user authentication and managing user accounts, and logging of satellite commands, responses and errors is accurately displaying in the UI according to requirements.

14.3 Syntax

14.3.1 Exported Constants

N/A

14.3.2 Exported Access Programs

N/A

14.4 Semantics

14.4.1 State Variables

N/A

14.4.2 Environment Variables

Name	Type	Description
DB_USER	String	Login information for MongoDB Cluster
DB_PASSWORD	String	Login information for MongoDB Cluster
DB_URI	String	Connection path to MongoDB instance

14.4.3 Assumptions

The only assumption that Backend service module depends on would be for the front-end and back-end to be running properly.

14.4.4 Access Routine Semantics

N/A

14.4.5 Local Functions

N/A

15 Appendix

15.1 User Interface Design - Figma

Figma link for the user interface design

15.2 Component Diagram

Link to Component Diagram

15.3 Reflection

1. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better?

The current limitations of the project stem from difficulties in verification and testing, particularly in connection with an operational ground system for scheduling on actual satellites. Additionally, if we had unlimited resources, we would create our own fully operational ground server and test satellite architecture to ensure validity. However, the application's requirement is to be hosted on the MIST linux based server making communication challenging as functionalities like pinging are restricted. Having unlimited resources would allow us to establish our server capable of sending commands directly to the satellite. Another limitation of the project is the testing of command scheduling and satellite communication features. Currently, we lack a satellite for direct communication, so we must rely on a mock server for our tests. If unlimited resources were available, launching a test satellite to create a real-like environment for satellite communication testing would be the approach. Given unlimited resources, we can also enhance the authentication process by implementing multi-factor authentication for an extra layer of safety and rate limiting to prevent brute-force attacks. Better visualization of satellite path and related information would also enhance the user experience.

2. Give a brief overview of other design solutions you considered. What are the other benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design?

Given the set of requirements imposed by external stakeholders, our design solutions were somewhat constrained. However, we actively managed complexity by decomposing our solution into distinct modules and implementing a high cohesion-low coupling approach to enhance the independence of these modules. Specifically, we ensured high cohesion by organizing modules to perform closely related tasks, such as command scheduling, communication protocols, and data processing, which minimized dependencies and promoted clarity within each module. Low coupling was achieved by designing interfaces between modules to be minimal and abstract, allowing for easier maintenance, testing, and future

modifications without impacting other parts of the system. This design approach aligned with good software engineering practices, particularly in terms of architectural considerations, as a component-based architecture was well-suited to our project requirements. Since the project description provided us with most of the functional and nonfunctional requirements, there was limited room for exploring alternative design ideas for our application. While we did contemplate design decisions related to the application's user interface, the core functionalities remained consistent. Other design decisions such as experimenting with different communication protocols or architectural patterns, had to be ruled out early on due to the need for compatibility with existing systems. Our project's predefined requirements for both functionality and technology stack further limited the scope for exploring diverse implementation strategies at the outset of the project.