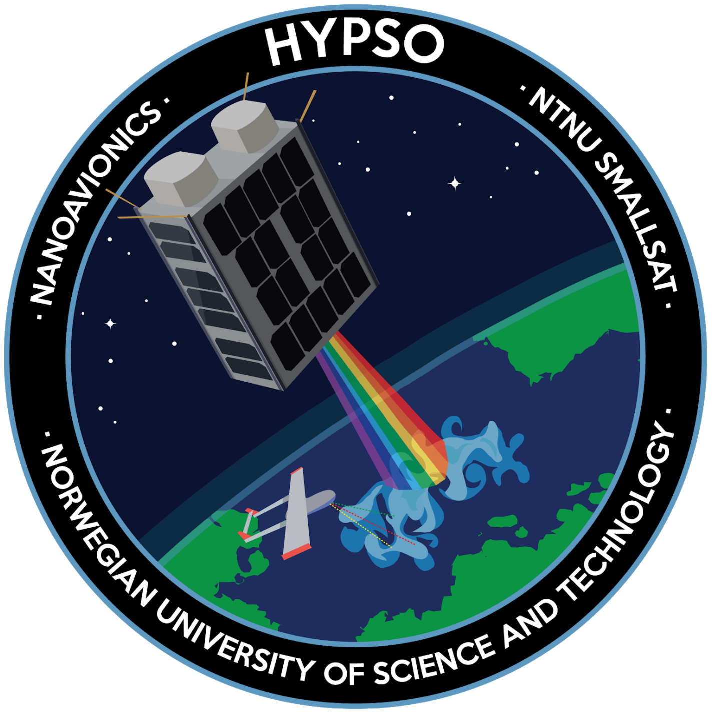
Open MCT Integration

HYPSO-DR-015



Prepared by:	HYPSO Project Team
Reference:	HYPSO-DR-015
Revision:	5
Date of issue:	2020-06-22
Status:	Preliminary
Document Type:	Design Report

Table Of Contents

Table 1: Table of Changes

Rev.	**Summary of Changes**	**Author(s)**	**Effective Date**
0.1	*Initial Draft*	*J L Garrett*	
		R Birkeland	
0.2	*Document outline, title change*	Audun V. Nytrø	
0.3	*Documented system architecture and plan ahead*	Audun V. Nytrø	2020-03-27
0.4	*Adding new class diagrams and documentation for v1.0 release*	Audun V. Nytrø	2020-06-20
0.5	*Added server software name*	Audun V. Nytrø	2020-06-22

1 Overview

The HYPSO Mission will primarily be a science-oriented technology demonstrator. It will enable low-cost & high-performance hyperspectral imaging and autonomous onboard processing that fulfill science requirements in ocean color remote sensing and oceanography. NTNU SmallSat is prospected to be the first SmallSat developed at NTNU with launch planned for Q4 2020 followed by a second mission later. Furthermore, vision of a constellation of remote-sensing focused SmallSat will constitute a space-asset platform added to the multi-agent architecture of UAVs, USVs, AUVs and buoys that have similar ocean characterization objectives.

1.1 Purpose

The purpose of the HYPSO-DR-015 Open MCT Integration Design Report is to provide an overview of MCT Depot, a telemetry depot and web server package for viewing and browsing historical and live telemetry data using NASA's Open MCT software package.

1.2 Scope

This document covers the use and implementation of MCT Depot itself and the associated customized Open MCT instance and its plugins.

It does not cover any external users of the data used by the telemetry web server, such as the ground station software suite, but does describe how these users can interface with the telemetry web server to get historical and real-time data for use outside Open MCT.

1.3 Summary

This document describes the implementation details, usage and status of MCT Depot for displaying and analyzing telemetry data from the HYPSO spacecraft in Open MCT.

MCT Depot is an Express-based web server running on Node.js with SQLite, which allows users to get real-time and historical telemetry data in Open MCT, or using a separate HTTP or WebSockets client.

The document currently consists of the following:

- Chapter 2: System description, which contains implementation details and explanations for how the overall system is intended to work
- Chapter 3: System usage, which contains usage instructions and recommendations
- Chapter 4: Current status, which explains what work remains and the plan ahead for getting that work done

1.5 Applicable Documents

The following table lists the applicable documents for this document and work.

Table 2: Applicable Documents

ID	Author	Title
[AD01]	ECSS Secretariat	ECSS-M-ST-10C: Space management: Project planning and implementation (tailored)
[AD02]	Norwegian Space Agency	NSC-Payload-PAQA (tailored)
[AD03]		
[AD04]		
[AD05]		

1.6 Referenced Documents

The documents listed in Table 3 have been used as reference in creation of this document.

Table 3: Referenced Documents

ID	Author	Title
[RD01]	Mariusz Grøtte	Telemetry Format Spreadsheet https://docs.google.com/spreadsheets/d/1DCzhNcp5J5GWeUGTtZC0nzXIwu4XpLFEQB6uAN2jq2Y/edit#gid=0
[RD02]		
[RD03]		
[RD04]		
[RD05]		

2 System description

2.1 Quick Introduction to Open MCT

Open MCT - short for Open Mission Control Technologies - is a new mission control framework developed and used by NASA for visualization of various types of data inside a web browser, both on mobile and desktop devices.

It is very flexible and extensible, allowing for many different types of data to be integrated and easily accessible on one single website for mission planning and telemetry data analysis. It reduces the need for mission operators to switch between many different applications to view all necessary data.

More information can be found on Open MCT's official website: https://nasa.github.io/openmct/.

2.2 Overview

At its core, MCT Depot consists of a web server that gets data from NanoAvionics' server, stores it locally and hosts the Open MCT frontend along with code that allows Open MCT to understand the telemetry data format used by NanoAvionics. A summary and early version of the system structure and its connections can be seen in the high-level system block diagram shown in figure 2.1.

The data is stored in a local database to abstract away the connection to NanoAvionics, allowing us to manipulate and access the data as we want without having to depend on their server being available, and to avoid constantly doing requests to it for data we've downloaded before. This reduces our load on their server, and allows us to easily support multiple instances of Open MCT (or other services that need telemetry data) running and accessing telemetry data simultaneously if required.

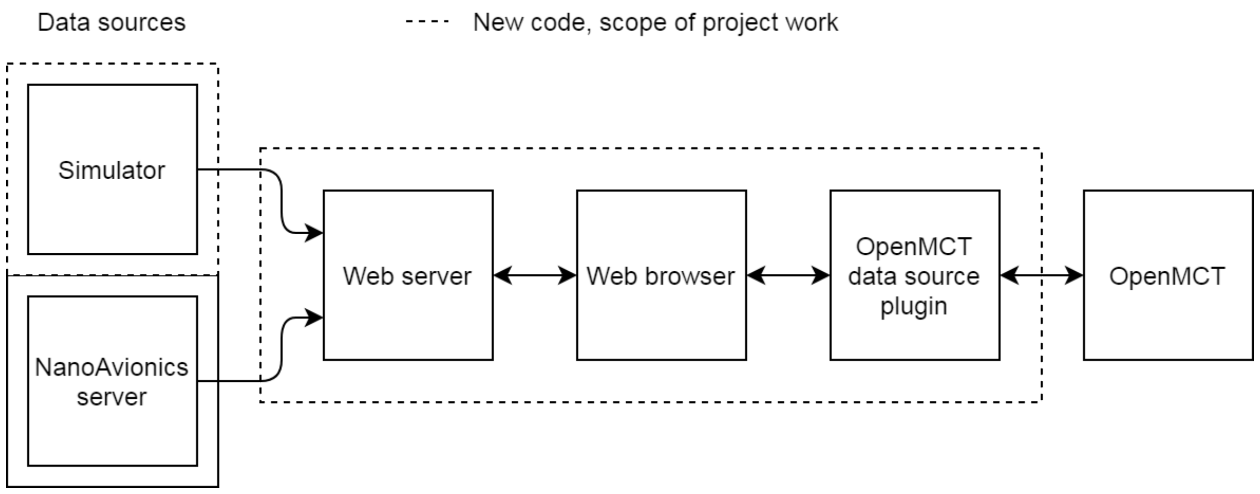


Figure 2.2 High-level system block diagram

2.3 Software architecture

To implement the system outlined above, it was decided to use JavaScript (in the form of ECMAScript 6) for the backend since it's already used in the Open MCT frontend, to avoid introducing more than one extra programming language for the HYPSO project to support in relation to this system, and to encourage code sharing between the frontend and backend.

Seeing as we want the system to be independent of the data source - whether it's directly to NanoAvionics' server, a file-based backup, or another server providing some kind of relevant data, it was natural to split the project into three more or less independent parts: The telemetry fetching system, the data management and storage system, and the telemetry serving system. The data management and storage system is the link between the otherwise independent telemetry fetching and telemetry serving systems.

Seeing as using JavaScript was wanted for the backend, it was decided to use Node.js running an Express-based web app to host and manage each of these subsystems. Express was chosen since it is one of the most used and well-documented web server frameworks for Node.js, and has already been used in multiple other successful implementations of Open MCT telemetry servers including the current official Open MCT server tutorial.

Based on the block diagram above a data flow diagram for the full MCT Depot system was created to get an idea of the types and amounts of data that would be going between each subsystem. This can be found in Figure 2.3.1.

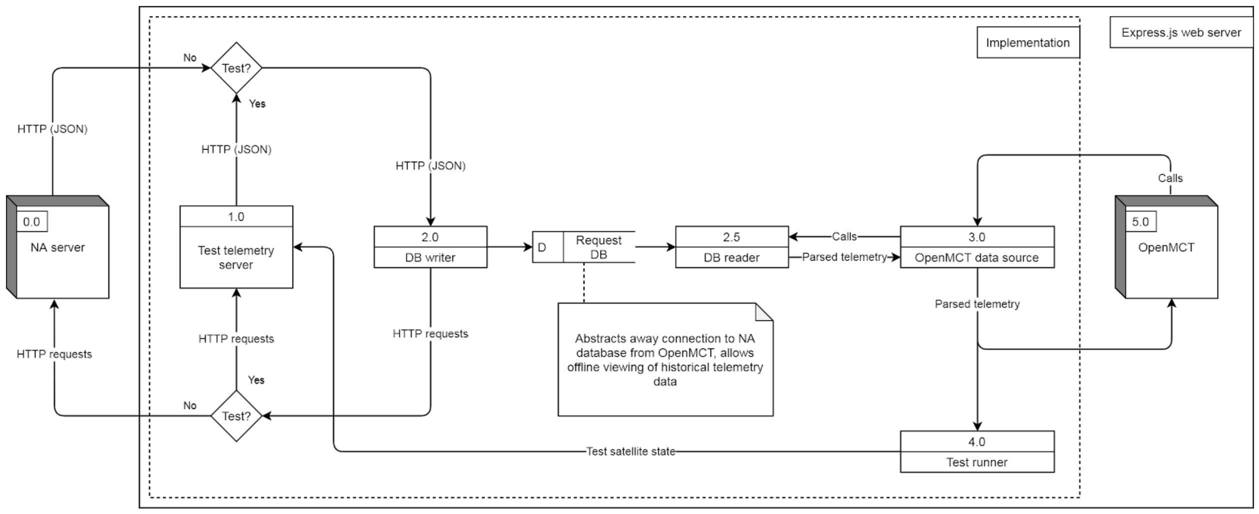


Figure 2.3.1 Level 0 Data Flow Diagram for MCT Depot

After this a full class diagram with the full extent of modules and descriptions of their local variables and external methods was created; this will be referenced actively in the section below outlining the functionality and decisions behind the implementation of each module. This can be found in Figure 2.3.2, with cutouts for each subsystem found in Figures 2.3.3 to 2.3.7.

Attempting to read Figure 2.3.2 directly is not recommended, and it is mostly included as a way to get a quick overview of the connections between the subsystems.

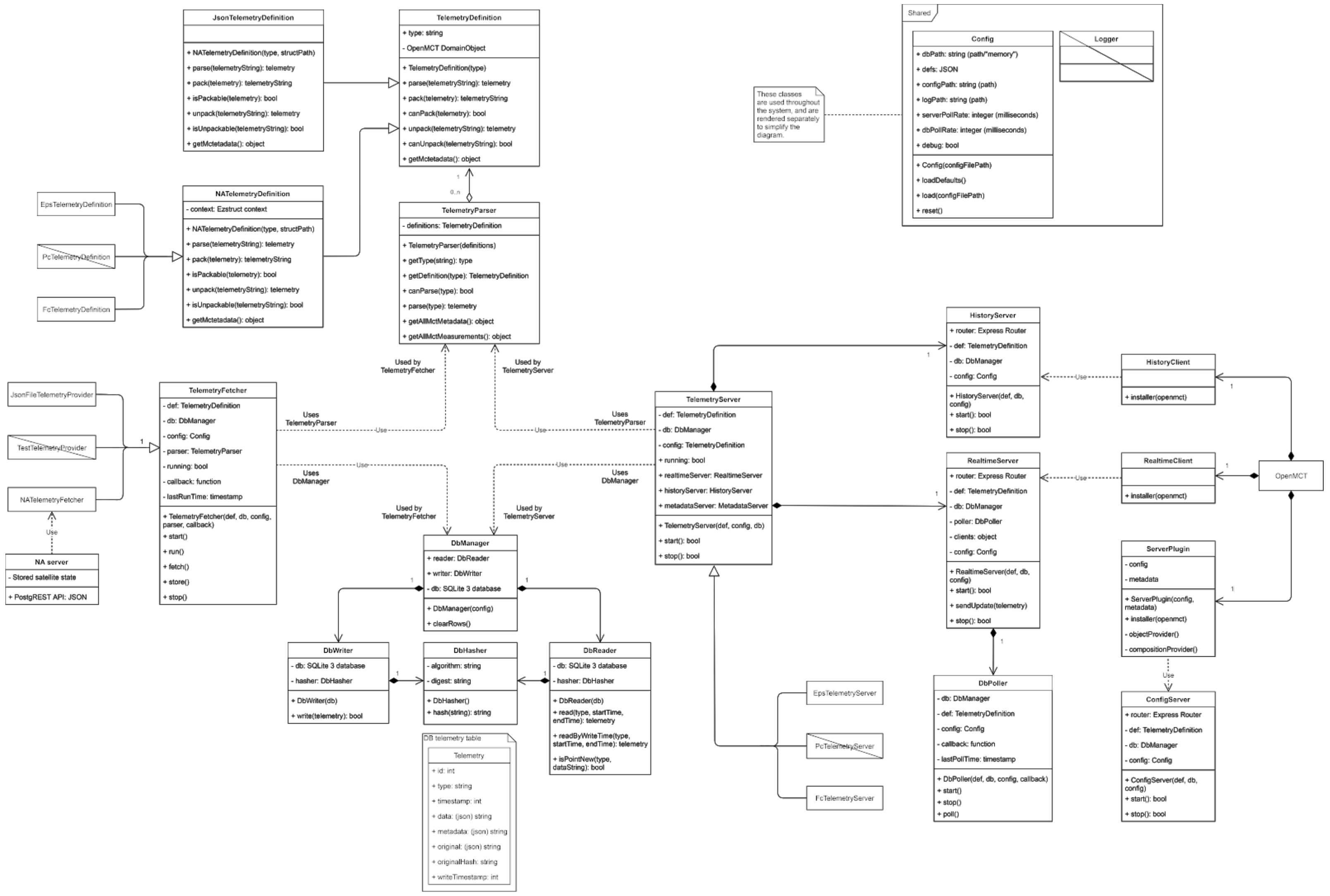


Figure 2.3.2 Class Diagram for full system

2.3.1 Shared modules

These provide simple but important utility functions, such as managing the current configuration of the system and allowing it to be imported/exported to a file, plus providing a system log.

The class diagram for these modules can be found in Figure 2.3.3.

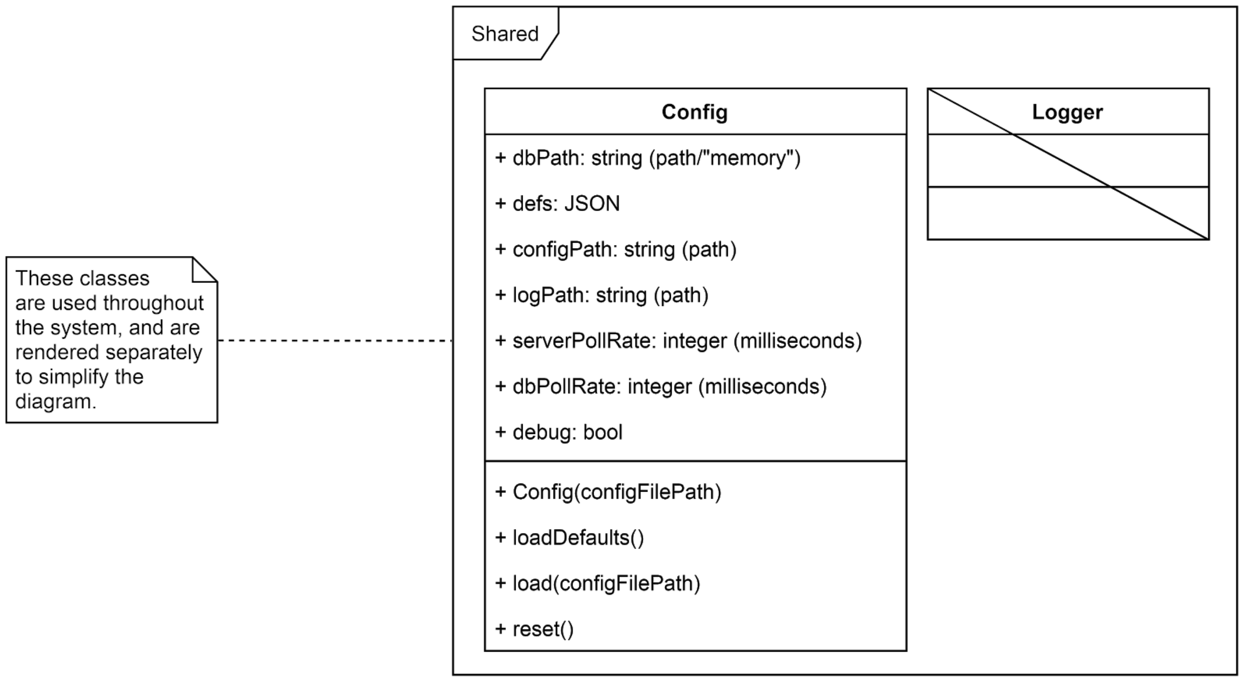

Shared
Config
+ dbPath: string (path/"memory")
+ defs: JSON
+ configPath: string (path)
+ logPath: string (path)
+ serverPollRate: integer (milliseconds)
+ dbPollRate: integer (milliseconds)
+ debug: bool
+ Config(configFilePath)
+ loadDefaults()
+ load(configFilePath)
+ reset()
Logger
These classes are used throughout the system, and are rendered separately to simplify the diagram.

Figure 2.3.3 Class Diagram for shared modules

2.3.2 Telemetry fetching subsystem

This subsystem provides an interface for getting telemetry data from external sources implemented by the module `TelemetryFetcher`, which is realized for various specific external sources as `FileTelemetryProvider`, `TestTelemetryProvider`, and `NATelemetryProvider`. As the names imply, these get telemetry data from a file, a test spacecraft, and NanoAvionics' servers respectively.

The default implementation of all of these are simple polling services that check their sources at a regular interval to see if theres any new data; if so this data is written to the connected `DbManager`. A connection to a `TelemetryDefinition` via `TelemetryParser` may be required if the telemetry type, timestamp or any metadata to be stored cannot be determined directly from the received data without it being unpacked and parsed first. These modules will be introduced in the next section.

The class diagram for these modules can be found in Figure 2.3.4.

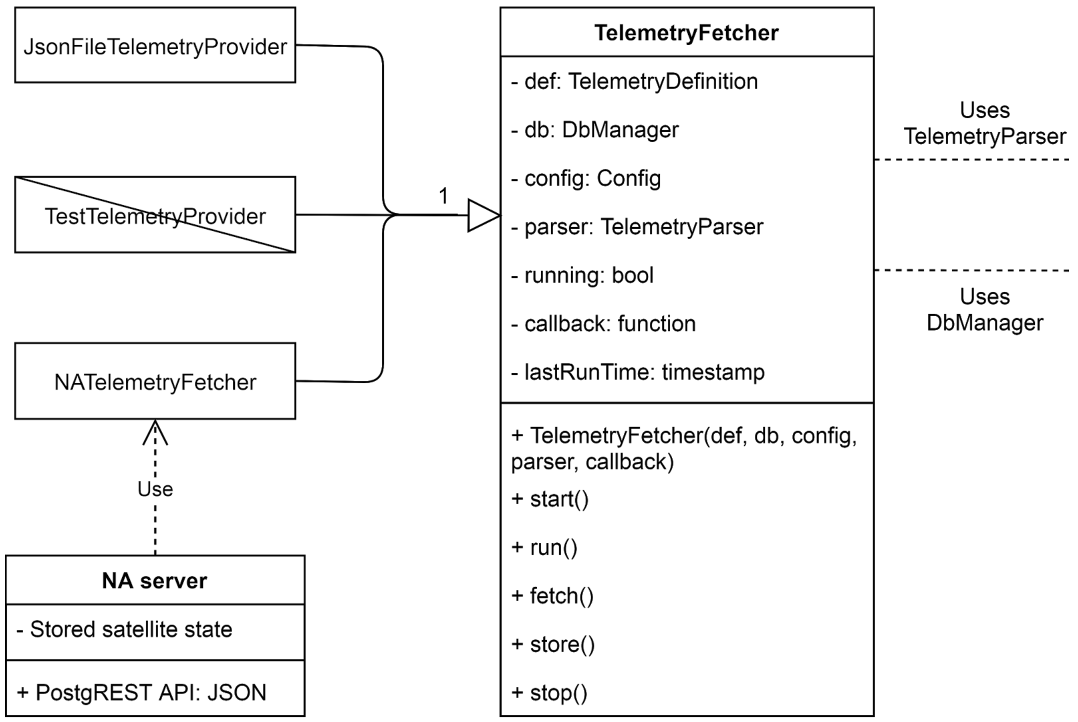


Figure 2.3.4 Class Diagram for telemetry fetching section

2.3.3 Data management and storage subsystem

This subsystem consists of two main modules; the `TelemetryParser` and `DbManager`.

The first of these, `TelemetryParser`, provides methods for configuring, parsing and unpacking various types of telemetry data, with a general class `TelemetryDefinition` that has a special subclass `NATelemetryDefinition` that provides a quick way to set up the `TelemetryDefinitions` for each type of telemetry we get directly from NanoAvionics' server.

The data that arrives from NanoAvionics' from their PostgREST-based HTTP API is in the form of JSON with some basic metadata and, more importantly, a string that contains a packed C-style struct. In `NATelemetryDefinition` we use a third-party module called EzStruct that allows us to quickly convert this to JSON based directly on the struct header definitions we get in text files from NanoAvionics, speeding up the process of implementing and updating the telemetry definitions greatly. Updating or adding a new `TelemetryDefinition` for parsing telemetry data from NanoAvionics' is usually as simple as downloading the struct header text file and updating the configuration to include it.

The class diagram for this subsystem can be found in Figure 2.3.5.

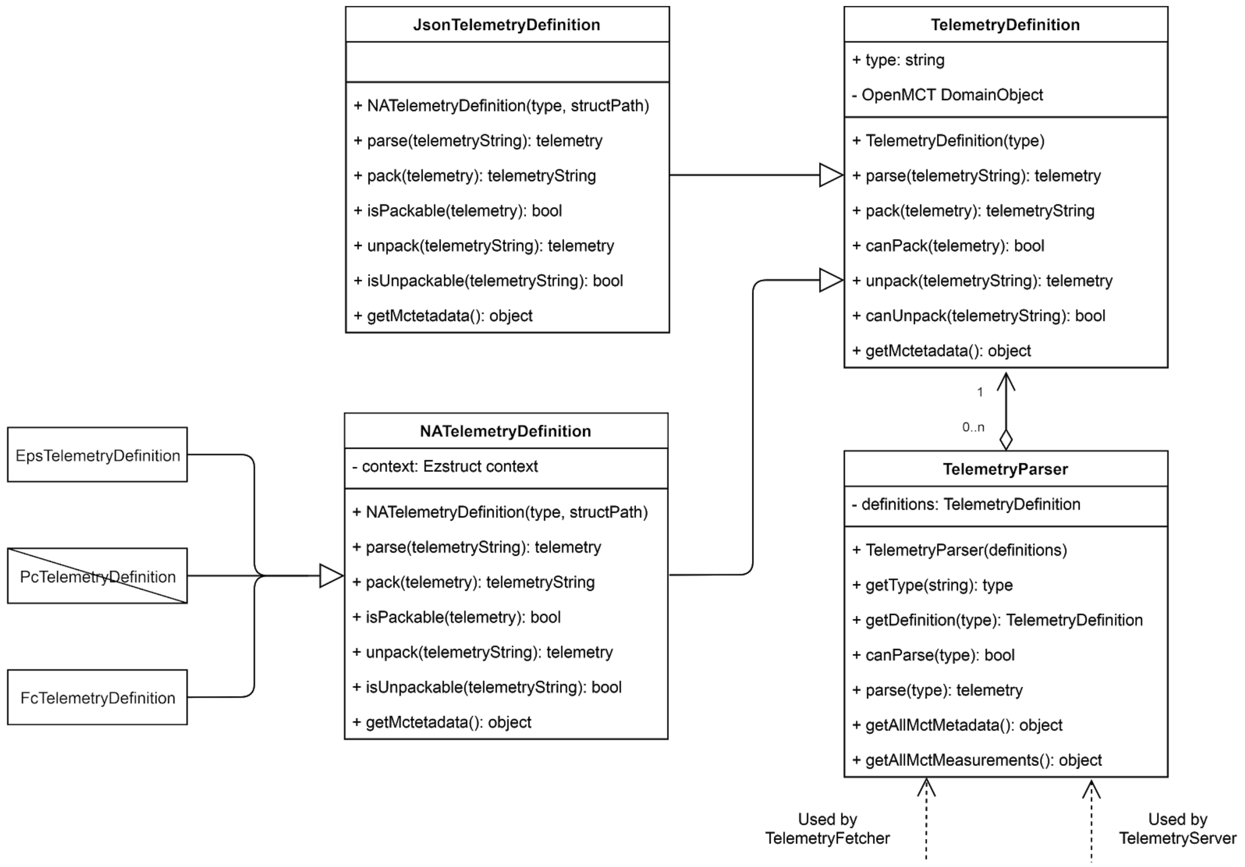

JsonTelemetryDefinition
+ NATelemetryDefinition(type, structPath)
+ parse(telemetryString): telemetry
+ pack(telemetry): telemetryString
+ isPackable(telemetry): bool
+ unpack(telemetryString): telemetry
+ isUnpackable(telemetryString): bool
+ getMctetadata(): object
TelemetryDefinition
+ type: string
- OpenMCT DomainObject
+ TelemetryDefinition(type)
+ parse(telemetryString): telemetry
+ pack(telemetry): telemetryString
+ canPack(telemetry): bool
+ unpack(telemetryString): telemetry
+ canUnpack(telemetryString): bool
+ getMctetadata(): object
1
0..n
EpsTelemetryDefinition
PcTelemetryDefinition
FcTelemetryDefinition
NATelemetryDefinition
- context: Ezstruct context
+ NATelemetryDefinition(type, structPath)
+ parse(telemetryString): telemetry
+ pack(telemetry): telemetryString
+ isPackable(telemetry): bool
+ unpack(telemetryString): telemetry
+ isUnpackable(telemetryString): bool
+ getMctetadata(): object
TelemetryParser
- definitions: TelemetryDefinition
+ TelemetryParser(definitions)
+ getType(string): type
+ getDefinition(type): TelemetryDefinition
+ canParse(type): bool
+ parse(type): telemetry
+ getAllMctMetadata(): object
+ getAllMctMeasurements(): object
Used by TelemetryFetcher
Used by TelemetryServer

Figure 2.3.5 Class Diagram for telemetry parsing section

The next major part is the `DbManager`, which handles storing data (usually from a `TelemetryFetcher`) and reading data (usually from a `TelemetryServer`). The interface here is fairly simple, as the current implementation just has a single table that can be indexed by timestamp and type (which maps to a `TelemetryDefinition` that can be used to unpack the data stored for that type).

The class diagram for this subsystem can be found in Figure 2.3.6.

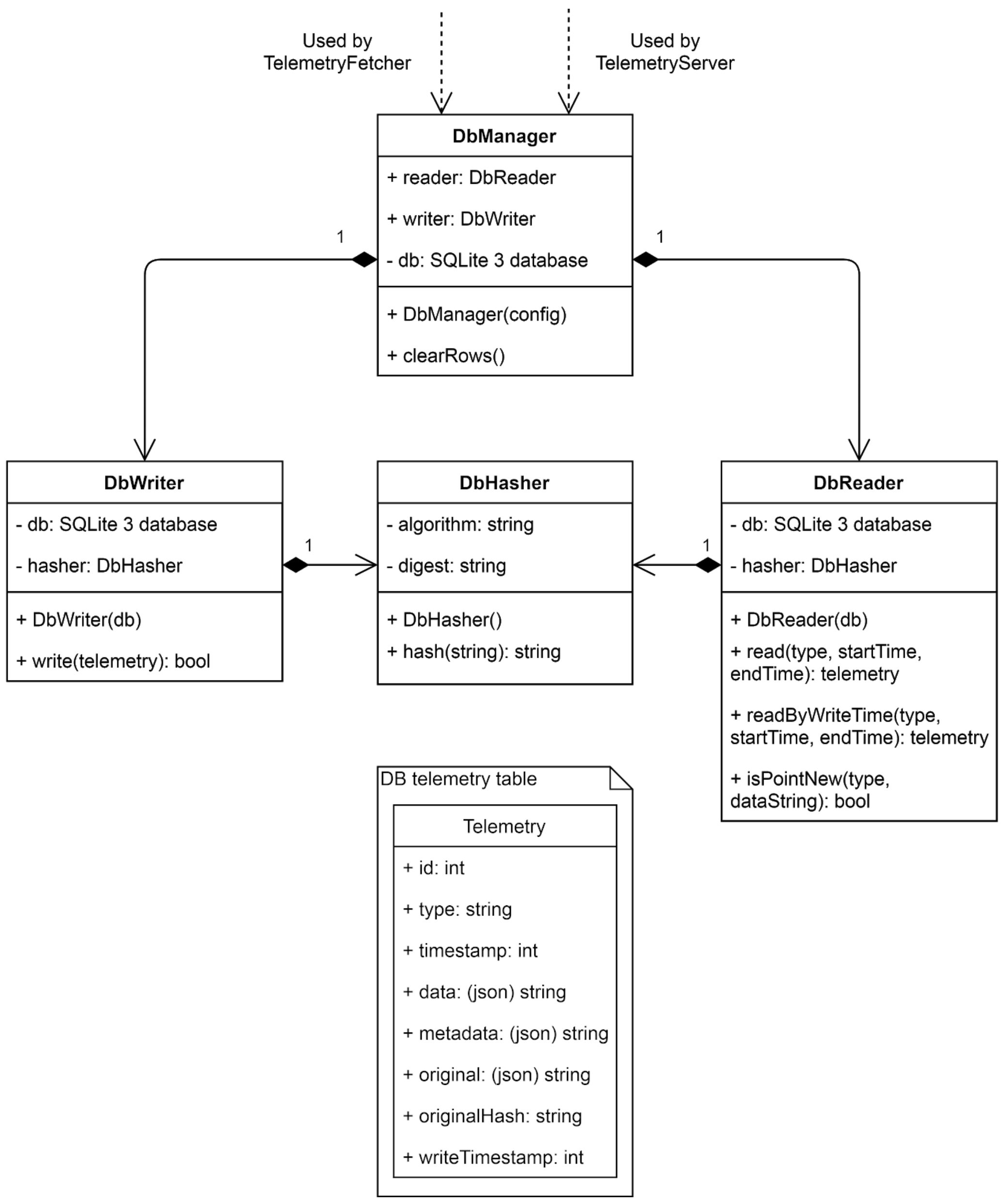


Figure 2.3.6 Class Diagram for database management section

To host the database it was decided to use SQLite; the fairly simple structure of the database used works well with it, and the large benefits it provides in portability (for “offline” work or backups) due to the database being stored in a single file made it a good candidate compared to other more complex database solutions.

2.3.4 Telemetry serving subsystem

This subsystem hosts and manages one or more HTTP/WebSocket servers for getting data for a specified `TelemetryDefinition` from a `DbManager`. The HTTP servers (implemented by

`HistoryServer`) are used for allowing a service to request historical data for a period, while the WebSocket servers (implemented by `RealtimeServer`) allow a service to subscribe to real-time data updates without having to continuously poll a HTTP server.

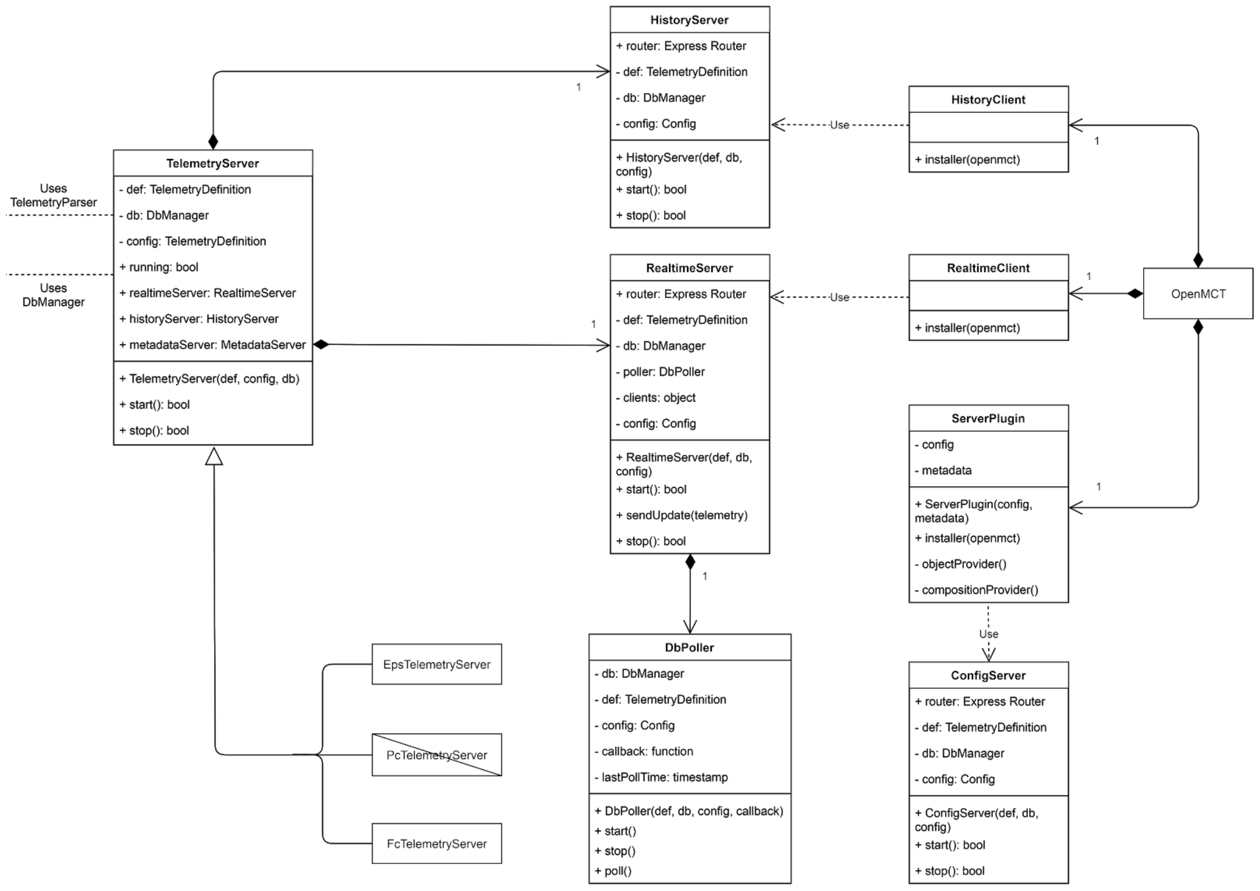


Figure 2.3.7 Class Diagram for telemetry server and Open MCT connection

2.3.5 Open MCT client plugins

These allow Open MCT to get and subscribe to telemetry data using the interface described above in section 2.3.4, and allows mapping this data to various user-defined visualizations.

2.4 Code standards

It was decided to keep the code style and syntax similar to what's already encouraged and used by the Open MCT project for both the backend and new frontend code required. A somewhat simplified version of Open MCT's ESLint configuration file is used to enforce this code style across this project; the simplifications are mostly the removal of configuration specific to external dependencies not used in this project.

2.5 Third-party modules

The main external modules to note besides Open MCT are EzStruct, better-sqlite3, Express and Jest with SuperTest.

3 System usage

THIS SECTION IS IN ACTIVE DEVELOPMENT, AND WILL EXPERIENCE MAJOR CHANGES AS THE SYSTEM DESIGN DESCRIBED ABOVE IS STILL IN THE PROCESS OF BEING FINALIZED AND IMPLEMENTED.

3.1 Setup and configuration

3.1.1 Server setup

The production server setup is still a work in progress.

A test/development server can be started by cloning the repository, navigating to it with a terminal and running npm install followed by npm run start to start the server. npm run test can be used instead to get a test coverage report.

3.1.2 Configuration options

Configuration options for the system are by default loaded from config.json inside the /state directory. See the included file for a sample on how to set up or change the system configuration - it uses all currently available configuration parameters.

3.1.3 Data definitions

NanoAvionics data definitions are added by copying the C struct definition for it to the structs folder, and referencing it in the configuration options. This has already been done for the EPS and FC general telemetry definitions.

3.1.4 Data sources

The data source for a telemetry definition is set in the definition for it inside config.json. Currently only one data source is supported per telemetry definition.

3.2 Operation

3.2.1 Using Open MCT

This section is still in development.

Navigating pre-configured views, creating new views, links to relevant Open MCT documentation

3.2.2 Monitoring system status

Not yet implemented in v1.0 of MCT Depot.

Status indicators and system health - last time data received from local telemetry server, last time data received from external telemetry server, last time data received from spacecraft

3.3 Expansion options

3.3.1 Adding new data sources

Documentation on how to set up new fetcher for data source, setting up definition for parsing and storing the data, setting up Open MCT plugin to read data

3.3.2 Telemetry API

The server provides telemetry data and metadata over HTTP and WebSockets.

Currently the easiest way to familiarise yourself with how these work is by taking a look at the requests sent by Open MCT to the server or by checking out the RealtimeServer, HistoryServer and ConfigServer test files.

4 Current status

See also the issue tracker for this project at https://github.com/AudunVN/hypso-openmct/issues for up-to-date and more detailed descriptions of the status and remaining work required for each subsystem.

Table 4: Module Status Summary

Module	**Status**
Shared modules	Initial version finished
Telemetry fetching	Initial version finished, needs additional testing against NanoAvionics' live PostgREST telemetry server
Data management and storage	Initial version finished
Telemetry serving	Initial version finished
Open MCT client	Initial version finished
Documentation	Work in progress
Server setup	Work in progress

4.1 Plan

The documentation, test coverage and server setup currently have no specific time frame, but will likely be done at a later time in 2020 to be ready for launch.

5 List of abbreviations

Table 5: List of Abbreviations

Abbreviation	Description
DB	Database
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation: Common data-interchange format for web services
NA	NanoAvionics
NASA	National Aeronautics and Space Administration
Open MCT	Open Mission Control Technologies: https://nasa.github.io/openmct/about-open-mct/