**Introduction:**

This document provides the technical specification and the general usage instructions to operate the project being developed for the EMSO team.

1. **Project Overview**

This project is an effort to manage multiple vehicles across the University of Alberta fleet to monitor them and collect data that consists of various parameters of the vehicles and essentially replace Azuga, the current vehicle monitoring system used by the university fleet in the long run.

**1.1 Objective**

The project aims at using Freematics OBD devices to collect the OBD data from vehicles and store them in a database and further perform analysis on them to gain valuable insights on the data.

**1.2 Business Case**

The project aims to reduce the dependency of the university on outside technology like Azuga for vehicle management and monitoring in a bid to become self-sufficient and reduce the overall cost of managing these vehicles.

**1.3 Risks**

Several use cases consisting of the vehicle movement in areas with low connectivity and GPS signal interferences might prove detrimental to the collection of data and the operation of the project as a whole.

**1.4 Out of Scope**

The current Freematics teleserver doesn’t provide an API to access data that was previously not sent to the server due to a network issue. This data that is stored in the SD Card needs to be transferred using an SD Card and is currently not served via any of the APIs that can be used to access the teleserver.

**2. Technical Design Diagram**

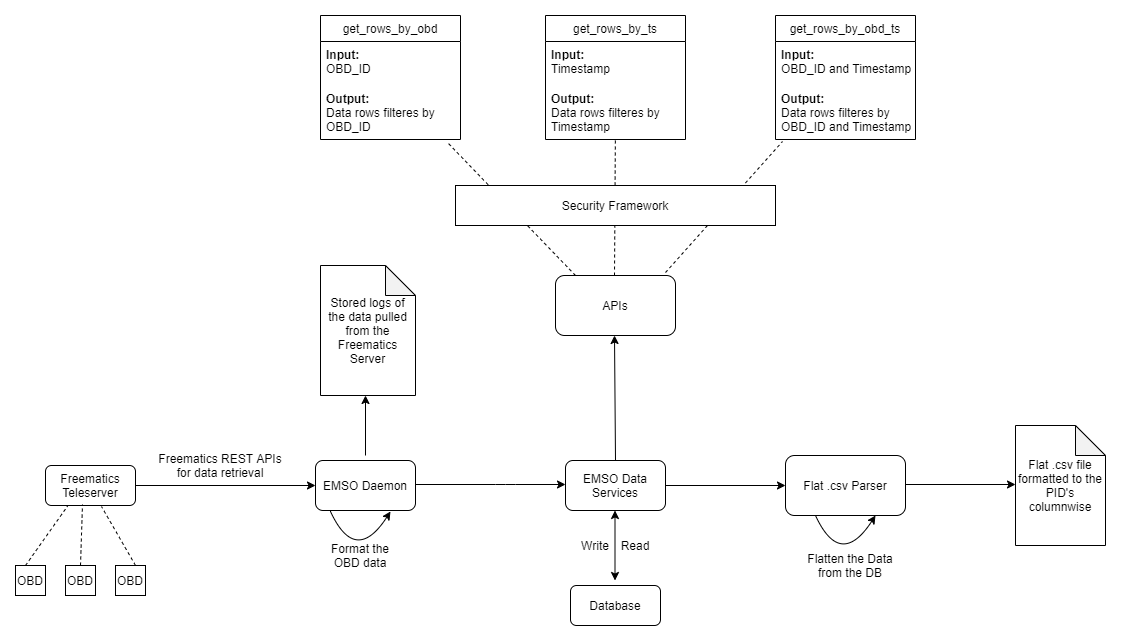


Figure 1: Technical Design Architecture Diagram

Figure 1 provides the high level overview of the system as a whole. The following passages provide a high level overview of the functionality of the architecture diagram and its components:

* **Freematics Teleserver:** The Freematics teleserver is a server management code published by Freematics the organization providing the OBD devices used in all the vehicles to collect their data. This program interacts with the various OBD devices and serves the data collected by them via a set of REST APIs.
* **EMSO Daemon:** This is a program developed in house as part of this project to interact with the teleserver and polling it continuously for data provided by the OBD devices. This piece of code also stores logs of the responses that it receives from the teleserver for future references.
* **EMSO Data Services:** This is part of the EMSO Daemon but is treated as a separate entity as it is functioning modularly irrespective of the data provider i.e this piece of code should stay the same even if Freematics is replace by a different company. The entire functionality of this module lies in interacting with the database to provide the user with the data stored in the database and also in storing data from the teleserver into the database. In addition, this module interfaces with the spring security framework to provide the ability to access the data in the database securely via REST APIs.
* **Flat CSV Parser:** The flat .csv parser is a python script that helps convert the nested data stored in the database into a flat format that is easy to understand and perform analysis on.
* **Security Framework:** The security framework is a layer that is used to validate and authenticate the user calls to the database and serve the data to the caller if they are authenticated only.

**3. Technical Specifications**

The technical specification outlines several details about the various components in the project and their functionality.

**3.1 Freematics Teleserver:**

The freematics teleserver is a piece of code developed by the Freematics organization that works in sync with the Freematics OBD devices. The project aims at leveraging two main APIs provided by the freematics teleserver for the data storage purposes:

1. **Channels API:** The channels API lists the various OBD devices that are currently active with the teleserver.

**API Signature:**

http://<server\_ip>:<port>/api/channels

1. **Pull API:** The pull API gives the data collected by the teleserver from the timestamp argument provided to it.

**API Signature:**

http://<server\_ip>:<port>/api/pull/<OBD\_ID>?ts=<timestamp>

**3.2 EMSO Daemon:**

The EMSO Daemon process is used to interact with the Freematics teleserver and the Security Framework to write and read the data to and from the database to serve it to the users via REST APIs specified in the architecture diagram.

**3.3 Security Framework:**

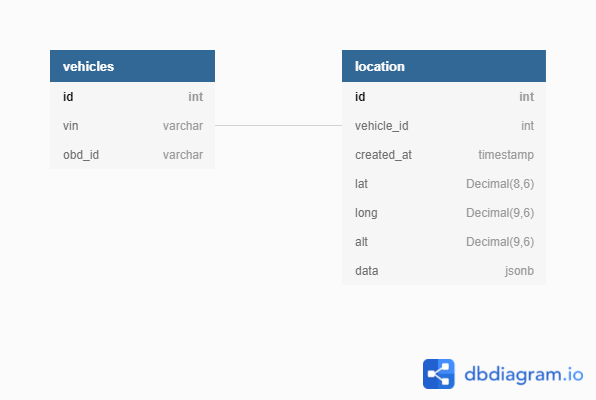
The security framework is built using the Spring Security modules available in Java. The architecture is designed to use JWT token authentication for the REST API calls. This process secures the APIs from being used by unauthenticated users.

**3.4 Databases**

There is one database schema used in this project. The vehicle\_data schema is used for both the vehicle data storage purposes and the authentication purposes for the Spring Security Framework.

1. **Vehicle Tracking Tables:**

The vehicle tracking tables consist of the vehicles and the location tables. These tables are used for the vehicle registration and tracking respectively. The following is the table specification for both the tables:



* Vehicles Table:

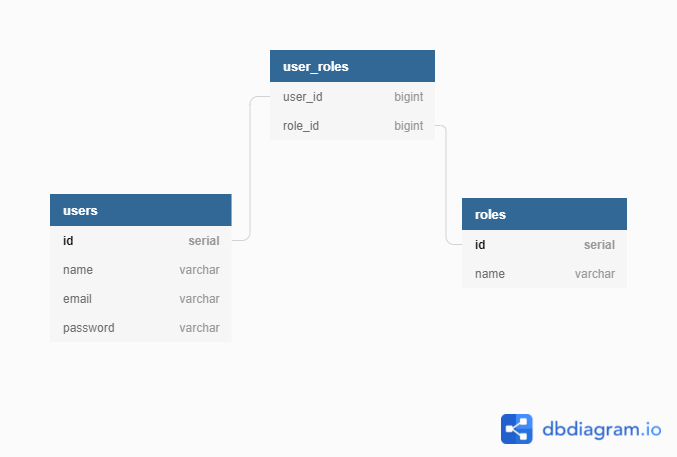
The vehicle table consists of three attributes. The two important ones are the vin (Vehicle Identification Number) and the obd\_id (OBD Device ID). Both the attributes are used to identify a particular vehicle.

* Location Table:

The location tables consists of the vehicle\_id which is a foreign key of the vin from vehicles thus not allowing for unregistered vehicles to be entered into the database. The created\_at attribute corresponds to the timestamp of the data retrieval by the EMSO daemon and the lat, long and alt consists of the location data while the data attribute consists of the various parameters of data collected by the OBD devices from the vehicle.

1. **User Security Tables:**

The user security tables consists of two tables namely the users and the roles table. These two tables consists of the user login information and their respective roles. In addition, a table join called user\_roles consists of the relation between the user and their role. The following is the table specification for both the tables:



* Users table:

The users table consists of the name, email and password of the user that has been registered with the system.

* Roles table:

The roles table consists of the name if the user with a particular role.

* User\_Roles table:

The user\_roles table provides the relation between a user and their role thus allowing us to understand their level of access.

**4. Resource Needs**

The smooth operation of this new system requires the usage of Freematics teleserver, OBD devices and the constant provision of internet and GPS services to the OBD devices installed on the vehicles.

**5. Assumptions**

There are several assumptions that have been made as part of the developmental process of the system at hand. One of the primary assumptions made is that the Freematics code and the teleserver work without any issues and do not fail in capturing the data. Also, assumptions pertaining to consistent internet and location data services has been made to the OBD devices to ensure that proper transfer of the device data is done to the teleserver.

**6. Conclusion**

The project consisting of the backend of the system which integrates with the Freematics teleserver and stores the data in the database and retrieves the same efficiently and in a fast manner has been developed. REST API support with output format of both JSON and CSV has been provided to support further development in the front end easily.

**7. Quick Start Guide:**

The following instructions provide the basic scenarios and activities that are to be performed by the users of this project on a regular basis.

Creating the Vehicle Tables:

The following commands can be used to create the vehicle related tables.

CREATE TABLE vehicles(

id SERIAL NOT NULL,

vin VARCHAR NOT NULL,

obd\_id VARCHAR NOT NULL,

UNIQUE(id)

);

CREATE TABLE location(

id SERIAL NOT NULL,

vehicle\_id INT NOT NULL,

created\_at VARCHAR,

lat Decimal(8,6),

long Decimal(9,6),

alt Decimal(9,6),

data jsonb,

FOREIGN KEY ("vehicle\_id") REFERENCES vehicles("id")

);

**Creating the Security Tables:**

The following commands can be used to create the security related tables.

CREATE TABLE vehicle\_data.users(

id SERIAL NOT NULL,

name VARCHAR NOT NULL,

email VARCHAR NOT NULL,

password VARCHAR NOT NULL,

UNIQUE(id)

);

CREATE TABLE vehicle\_data.roles(

id SERIAL NOT NULL,

name VARCHAR NOT NULL,

UNIQUE(id)

);

INSERT INTO vehicle\_data.roles(name) VALUES('ROLE\_USER');

INSERT INTO vehicle\_data.roles(name) VALUES('ROLE\_ADMIN');

CREATE TABLE vehicle\_data.user\_roles(

USER\_ID BIGINT NOT NULL,

ROLE\_ID BIGINT NOT NULL

);

**Logging into the DB:**

Psql –U postgres

**Connecting to the EMSO DB:**

\c emso

**Listing the registered vehicles:**

Select \* from vehicle\_data.vehicles;

**Registering the vehicles into the DB:**

INSERT INTO vehicle\_data.vehicles (vin, obd\_id) VALUES ('<vehicle vin>', '<vehicle obd>');

**Rename Vin:**

UPDATE vehicle\_data.vehicles SET vin='<old vin>' WHERE vin='<new vin>';

**Delete entry from vehicles:**

DELETE FROM vehicle\_data.vehicles WHERE vin='<to delete vin>';

**Usage of the secure REST APIs:**

**Signup User:**

**Post:** <http://localhost:5000/api/auth/signup>

**Content-Type:** application/json

**Body:** {

"name": <user name>,

"email": <user email>,

"password": <user password>

}

**Signin User:**

**Post:** <http://localhost:5000/api/auth/signin>

**Content-Type:** application/json

**Body:** {

"email": <user email>,

"password": <user password>

}

**Response:**

{

"accessToken": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiIxIiwiaWF0IjoxNjI5NzU2ODU3LCJleHAiOjE2MzAzNjE2NTd9.IilXVF4zQEKXPNLhrgReO\_Y0clzqH-\_tjKl7Irr\_pqOsjeUfLqVn3q-tFD9Sf5KNvDMcj6vr0j0RfWqqUWoklw",

"tokenType": "Bearer"

}

**Get All Data:**

**Get:** [http://localhost:5000/api/data/getAllData?response\_type=<json|csv](http://localhost:5000/api/data/getAllData?response_type=%3cjson|csv)>

**Headers:**

Authorization: Bearer <Access Token>

**Get Data corresponding to OBD ID:**

**Get:** [http://localhost:5000/api/data/getDataByObdId?obd\_id=<obdId>&response\_type=<json|csv](http://localhost:5000/api/data/getDataByObdId?obd_id=%3cobdId%3e&response_type=%3cjson|csv)>

**Headers:**

Authorization: Bearer <Access Token>

**Get Data corresponding to Timestamp Range:**

**Get:** [http://localhost:5000/api/data/getDataByTimestampRange?start\_time=<startTimestamp>&end\_time=<endTimestamp>&response\_type=<json|csv](http://localhost:5000/api/data/getDataByTimestampRange?start_time=%3cstartTimestamp%3e&end_time=%3cendTimestamp%3e&response_type=%3cjson|csv)>

**Headers:**

Authorization: Bearer <Access Token>

**Get Data corresponding to OBD ID and Timestamp Range:**

**Get:** [http://localhost:5000/api/data/getDataByObdIdAndTimestampRange?obd\_id=<obdId>&http://localhost:5000/api/data/getDataByTimestampRange?start\_time=<startTimestamp>&end\_time=<endTimestamp>&response\_type=<json|csv](http://localhost:5000/api/data/getDataByObdIdAndTimestampRange?obd_id=%3cobdId%3e&http://localhost:5000/api/data/getDataByTimestampRange?start_time=%3cstartTimestamp%3e&end_time=%3cendTimestamp%3e&response_type=%3cjson|csv)>

**Headers:**

Authorization: Bearer <Access Token>

**Usage of the Python CSV parser:**

The python parser can be run by just typing the following command in the command line:

**Command:** py FlatResponseBuilder.py

**8. Improvements made from the previous system.**

1. The Previous program used to crash very often leading to data collection loss but the new one doesn't as I have written handlers for every exception that I can think off during development. At least, I haven't seen the daemon crash until now.
2. The DB data retrieval into CSV is way faster as the previous program (used MySQL and noticeably worse schema structure which was not normalized) used to take around 5 mins to retrieve the data (despite the data being much less voluminous than what I'm storing right now) whereas ours (Postgres and I believe better schema structure) takes 2-5 seconds.
3. The previous program as I discovered from the code it was written of was missing the data leading to data loss when there were multiple rows of data in a single call. I fixed that to show them in multiple rows instead stopping the data loss.
4. The previous program was not modular and couldn't be integrated with any other company apart from Freematics and would have required a lot of code changes to do that. Ours needs just a change in the Device Connector module (Freematics to Daemon interface) and the Data format module where we convert the data to the DB schema structure as the format of data contract would be violated when the new company replacing Freematics is used.
5. Also, we have consolidated all the code of the three programs that they were using into a single java jar file. This gives rise to portability into any operating system and environment and also as the backend was written using Spring as the framework on top of Java. It is easy to integrate with the front-end be it React, Vue or Angular and has easy integration with Spring security for token authentication and also has native support to integrate with Hibernate for efficient DB calls. Which is the reason why the DB entries have become much more efficient.
6. In addition, the code structure has been written to allow for flexibility in the number and type of PIDs as I was always in the dark of what Freematics is sending leading to me being more cautious and general in which approach I took in designing the code so as to allow for unknown variables like the new Hexadecimal and weird Null PIDs coming right now.
7. The previous piece of code being used was only looking for a select number of PIDs leading to an issue of data loss whereas the flat parser written by me takes into account every PID provided to me and also stores the unknown PIDs in the nested format to allow for safekeeping of data even though we don't understand it.