Radio System for Stock Monitoring in an Industrial Environment

Rui Guilherme Sarreira Horta Monteiro

Orientadores

Professor Doutor Paulo Torres

Trabalho de Projeto apresentado à Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco para cumprimento dos requisitos necessários à obtenção do grau de Licenciatura em Engenharia Eletrotécnica e das Telecomunicações, realizada sob a orientação científica do Professor Doutor Paulo Torres, do Instituto Politécnico de Castelo Branco.

October 2024

Members of the juri

President of the juri

Professor Doutor, Paulo Torres (Advisor)

Prof. Adjunto, IPCB - ESTCB

Members

Professor Doutor, Nuno Octávio Fernandes

Prof. Adjunto, IPCB - ESTCB

Professor Doutor, Hugo Marques

Prof. Adjunto, IPCB - ESTCB

Dedication

To my grandparents.

Acknowledgments

I thank my parents and sisters for all the support that made this project possible.

My advisor, Professor Dr Paulo Torres, for his full availability, motivation, and patience.

David Lucas, from the company RDR, for providing operational information about the company.

And everyone else who, in one way or another, helped make this project possible.

Abstract

In present day companies are becoming increasingly aware on the benefits of digitalization as a mean to simplify and streamline stock management, increase process efficiency, and possibly boost business revenues.

To that end, the present work focuses on developing a radiofrequency-based stock management system for end-of-life vehicle dismantling facilities, so that stock information can be easily managed and accessed allowing fast access to key information such as item location and quantity, aiding with sales and improving the efficiency of the dismantling process, through the possibility of using this information with numerical and machine learning algorithms.

The proposed system consists of an RFID reader module connected to a NodeMCU ESP32 responsible for reading, displaying, and sending the RFID tags information to an API endpoint through an existing Wi-Fi connection using the protocol MQTT. The stock management is made through a webapp so both stock information and associated metrics can be easily visualized and accessible from any device with a supported browser and from anywhere there is an internet connection.

Keywords

Stock management system, inventory tracking, dismantling industry, RFID, ESP32

Resumo

As empresas estão cada vez mais atentas aos benefícios da digitalização, como meio de simplificar e agilizar a gestão de stock, aumentando assim a sua eficiência, os resultados operacionais e as suas receitas.

Este trabalho foca-se no desenvolvimento de um sistema de gestão de stock, baseado em radiofrequência, para o desmantelamento de veículos no fim de vida, permitindo assim que a informação do stock seja facilmente acessível e gerível, permitindo, desta forma, a possibilidade de aplicar métodos numéricos e de aprendizagem de máquina, contribuindo para a melhorar a eficiência do processo de desmantelamento e de venda, ao garantir o rápido acesso a informação-chave, como localização e quantidade.

Este sistema consiste num modulo de leitura RFID, conectado a um NodeMCU ESP32 responsável por ler e enviar a informação das etiquetas RFID, para uma API, através de uma ligação de Wi-Fi, usando o protocolo MQTT. Sendo que, a gestão de stock é feita através de uma aplicação web, que possibilita a fácil visualização e acesso à informação relativa ao stock e métricas associadas, a partir de qualquer equipamento, com ligação à internet e um browser.

Palavras chave

Sistema de gestão de stock, monitorização de inventário, indústria de desmantelamento, RFID, ESP32

Table of contents

Table of figures

Symbol index

bps – bits per second

dBm – decibel-milliwatt

GHz – giga Hertz

kHz- kilo hertz

m- meters

mA – milli-ampere

MB – Mega byte

Mbps – mega bit per second

MHz - mega hertz

μs – micro seconds

V - Volt

List of tables

List of abbreviations

RFID – Radio Frequency Identification

UHF – Ultra High Frequency

LF – Low Frequency

HF – High Frequency

ISO – International Standards Organization

SQL – Structured Query Language

RDBMS – Relational Database Management System

noSQL – not only SQL

JSON – Javascript Object Notation

BSON – Binary Javascript Object Notation

DB – Database

MQTT – Message Queuing Telemetry Transport

REST – Representational State Transfer

HTTP – Hyper Text Transfer Protocol

CRUD – Create, Read, Update, Delete

JWT – JSON Web Token

EPC – Electronic Product Code

Web app – Web application

PC – Protocol control bits

Id – Identifier

RSSI – Received Signal Strength Indicator

SOAP – Simple Object Access Protocol

RPC – Remote Procedure Call

RF – Radio Frequency

PIE – Pulse Interval Encoding

ASK – Amplitude Shift Keying

FSK – Frequency Shift Keying

DSB-ASK – Double Side Band Amplitude Shift Keying

SSB-ASK – Single Side Band Amplitude Shift Keying

PR-ASK – Phase Reversal Amplitude Shift Keying

TID – transponder ID

CRC – Cyclic Redundancy Check,

XPC – Extended Protocol Control

TLS – Transport Layer Security

SSL – Secure Sockets Layer

QoS – Quality of Service

URL – Uniform Resource Locator

UART – Universal Asyncronous Transmitter Receiver

1. Introduction and Context

In the year 2000, the European Union enacted a directive (Directive 2000/53/EC) [1]regarding end-of-life vehicles to be enforced by member states starting on 21 April 2002, setting targets for the reuse, recycling, and recovery of vehicle components. This directive mandates vehicle and equipment manufacturers to factor in the dismantling, reusing, and recovery when designing and producing their products such as to have a minimum of 85% by weight per vehicle to be reusable and/or recyclable, and a minimum of 95% by weight per vehicle to be reusable and/or recoverable. Meanwhile in Portugal, according to VALORCAR [2] data, during the year 2023, there were 101.315 end-of-life vehicles decommissioned at wreckers or junkyards. This results in thousands, if not millions of vehicle parts being removed from end-of-life vehicles which will need to be identified, inventoried and cataloged in order to be able to be sold for re-use. Thus, it is desirable to have a system providing the ability to identify and monitor the location of those parts, and so optimize the dismantling and sales processes.

1.1 Objectives

The present work is applied to auto dismantling industry and has the objective of developing a radio system for item tagging and reading of the tags at a distance, in a warehouse, based on existing solutions such as RFID and Wi-Fi. Each tag should correspond to an item or a family of items, i.e. car parts. To that end there is a need to develop a database to store the multiple item types and information to be implemented and visualized in an easy-to-use web application, to help mainly the inventory and sales operations, and make the whole process more efficient and accurate.

1.2 Report Structure

The report is divided into four chapters.

In the first chapter an introduction and framing as well as the required objectives for the proposed work is described.

At the second chapter current technologies and similar system implementations are discussed.

The third chapter describes the proposed system implementation as well as the necessary hardware and software components, the protocols, the necessary steps and how the system works.

In the fourth chapter a conclusion and future work suggestion is presented.

2. State of the art

2.1 RFID

2.1.1 Brief RFID history

RFID stands for radio frequency identification and as the name would suggest is a technology developed to identify objects at a distance using radio waves. Its origin is hard to pinpoint, however according to some authors [3] [4] it may have appeared the first time, albeit in a very rudimentary implementation from a modern standpoint, during World War II, as way to distinguish friend from foe aircrafts and ships from radar signals. At these early days, transistors had just been invented and RFID technology was very expensive, bulky and only used by the military, however with advances in technology such as miniaturization of transistors, development of microchips and smaller antennas and the development of RFID standards, led to its adoption in logistics, starting once again within the military before being adopted by large retailers such as Walmart. Fast forwarding to today, fostered by its advantages and decreasing cost thanks to economies of scale as well as standardization of RFID communication protocols, this technology has become ubiquitous and can easily be found in almost everywhere from supply chain to animal identification to credit and debit card payments.

2.1.2 RFID technology differences

RFID systems can be implemented in different ways. The main way they differ resides on the operating frequency of the carrier wave. As such, the choice of the operating frequency is one of the most important choices in the design of the RFID system and is heavily dependent on the application of the system, since the operating frequency dictates the maximum reading distance between the tag and the interrogator as well as the reliability and speed of the tag and interrogator operations which are influenced by the quality of the signal, being this signal quality in turn influenced by noise and radio waves interactions, such as reflections and absorptions, between the different materials in the environment in which it will operate. The choice of the operating frequency will also have an impact on the tag and system implementation cost and upkeep. Below follows a comparison (table 1) featuring the main differences and characteristics between RFID technologies based on its operating frequency.

As the system requires a reading distance greater than few centimeters and low tag costs, the RFID stock management and monitoring system will be based on the UHF (860-960Mhz) carrier wave technology as it presents the best compromise between tag cost, reading distance and susceptibility to opaque materials and interference.

Table 1 - Differences and characteristics between RFID frequencies based on its operating frequency, adapted from [5] [4]

	Low Frequency (LF)	High Frequency (HF)	Ultra High Frequency (UHF)	Microwave
Frequency (in Hz)	9-135 kHz	13.553-15.567 kHz	430-440 MHz; 860-930 MHz	2.4-2.4835 GHz; 5.8 GHz
Standards	ISO/IEC 18000 Part 2	ISO/IEC 18000 Part 3, ISO 15693, ISO 14443 Parts A and B	ISO/IEC 18000 Part 7; ISO/IEC 18000-6; EPCglobal Gen-1 and Gen-2 Standards	ISO/IEC 18000-4
Tag Expense	High	High, Medium	Medium	High
Reader Cost	Low	Medium	High, Medium	High
Max. Work Ramge (in meters)	~0,30m	~1m	~30m (active)	More than 100m (>300m active)
Date Rate	Low	Medium	High	High
Read Rates	Slow		Fast	Very Fast
Interference	Low	Low	Medium	High
Opaque Materials	Not Susceptible	Somewhat Susceptible	from Somewhat to Very Susceptible	Very Susceptible
Advantages	Low Environment Absorption	Available Worldwide	Perfect for medium range application	Wide access range
Common Applications	Animal ID, Security, Engine Immobilizers, Aeronautical and Marine Communication	Security, Item Tracking, Ticketing, Smart Cards, Personal Identification	Container, Truck, Tracking, Baby Monitors	Access Control, Industry, Cordless Telephones

2.1.3 Typical warehouse RFID implementation in logistic operations

Modern high efficiency and high-volume logistics operations use RFID technology throughout the supply chain. Identification can occur at item or pallet level or a combination of both. In the case of item level identification, a tag is attached to an item or to the package containing said item. In the case of pallet level identification, a tag is attached to a pallet containing a bundle of packaged items. These tags contain a unique identifier number which will be associated with the item information in a database. There are some tags with programmable memory space which support some additional information even though they have a relatively small memory, so information stored within the tag itself is rather limited. The tracking of the items is usually made by placing readers(interrogators) at key locations. For example, at a logistic center warehouse, they can be placed at the entrance where the items will be delivered by the trucks and at the warehouse exits. If the warehouse has a system of conveyor belts, they will usually be placed at key locations, such as choke points, points where the conveyor belts split and so on as to guarantee the correct routing and avoid misplacements. The readers can also be placed at the storage locations to ensure their arrival and presence as well as in forklifts to ensure the right items and quantities are being picked up [6] (figure 1).

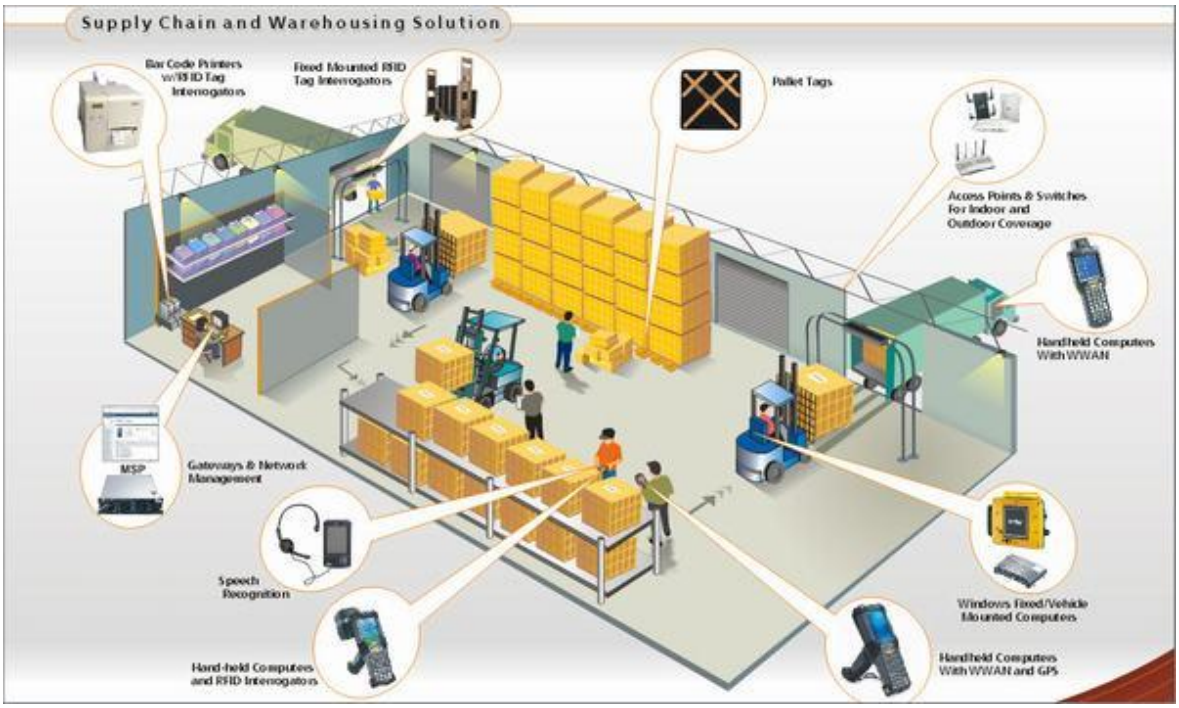

Figure 1 – RFID supply chain implementation [6]

2.2 Database management system

Traditionally structured query language (SQL) or Relational Database Management Systems (RDBMS) are preferred for stock management systems, over noSQL, since they guarantee ACID transactions [7]. ACID is an acronym, referring the key properties of a transaction, these being Atomicity, Consistency, Isolation, and Durability [8]. Atomicity means all the commands or operations in a transaction are treated as a single unit and will either all succeed or fail, hence atomic, from the Greek word Atomos meaning indivisible. Consistency means all the changes made as a result from a transaction must be consistent with the database constrains. Meaning, if a transaction fails to comply with any of the imposed constrains the whole transaction will fail and revert to its previous state. Isolation guarantees concurrent transactions won't interfere with each other since they run in isolation or separate from each other. Durability guarantees that after the transaction completes the changes will be written to the database permanently. However modern non-relational databases such as MongoDB, can also guarantee ACID transactions.

While RDBMS follow a rigid tabular structure, noSQL databases such as MongoDB stores data in the form of javascript object notation (JSON)-like documents (it is stored as Binary Javascript Object Notation or BSON) with customizable schemas, allowing more flexibility. Another difference between noSQL and RDBMS comes from the scaling differences. Traditionally RDBMS scale vertically, meaning that as data increases the usual option to deal with the increase in data is to increase the system resources, and even though modern RDBMS can scale horizontally, the process is complex and leads to some degradation of performance. noSQL on the other hand, can easily scale horizontally, meaning that it can deal with increases in data by adding additional nodes and distribute the storage and workload across multiple servers, usually implying fewer costs and the ability to scale dynamically, adapting to demand in real time.

Because of the higher flexibility, ease of scaling and potential lower costs, the noSQL database MongoDB was chosen to be used in proposed radio frequency-based stock management and monitoring system, in order to store the devices, users, credentials, vehicle, parts and analytics information. The database can be configured either as standalone, replica set, or sharded cluster. In the standalone configuration, the database consists of only one server storing the entire DB. The replica set configuration, consists of multiple servers connected to each other with each replica as the name indicates holding a replica, meaning a copy of the entire DB and should ideally live in different servers, ensuring high reliability, availability and redundancy. The sharded cluster configuration segments the DB in multiple shards, meaning that it splits the data in multiple segments which can be stored in different servers across different regions. This configuration ensures horizontal scaling and the possibility for lower latency and faster DB access speeds while also benefit from redundancy and high availability and reliability. For production deployments, it is advised to configure MongoDB either as a replica set of at least 3 servers or as a sharded cluster with the latter being the optimal configuration.

3. Development

3.1 Workflow walkthrough

The company receives a vehicle to be dismantled and adds it to the database, manually through the web application. It will then be stored at the company End of Life vehicle park until it will be dismantled. When it comes time to dismantle it, it will first pass through a process to depollute the vehicle, meaning that, the hazardous material such as the oils, batteries and so on will be removed prior to the dismantling process. It will then be dismantled. The company has then a workflow choice of dismantling all the parts to be removed from the vehicle first and then add them to the database, or it can do it part by part. Either way, to be able to make an item level identification, as the part information is inserted into the database an RFID tag needs to be attached to said part and associated with its information in the database. This process is made through the web application using the reader to read the tag unique identifier. Regarding storage location the worker will also have a choice. If the storage location is known upfront it can be inserted at the time the part is added to the database. If not, the location can be updated at the time of storage. After the dismantling process the parts will then be placed in storage until the time comes for the item to be sold. Inventory existence and correct placement can be made by a worker using the handheld RFID reader to scan the warehouse location selected in the webapp and verifying whether the items registered in the database for the location were detected (figure 2).

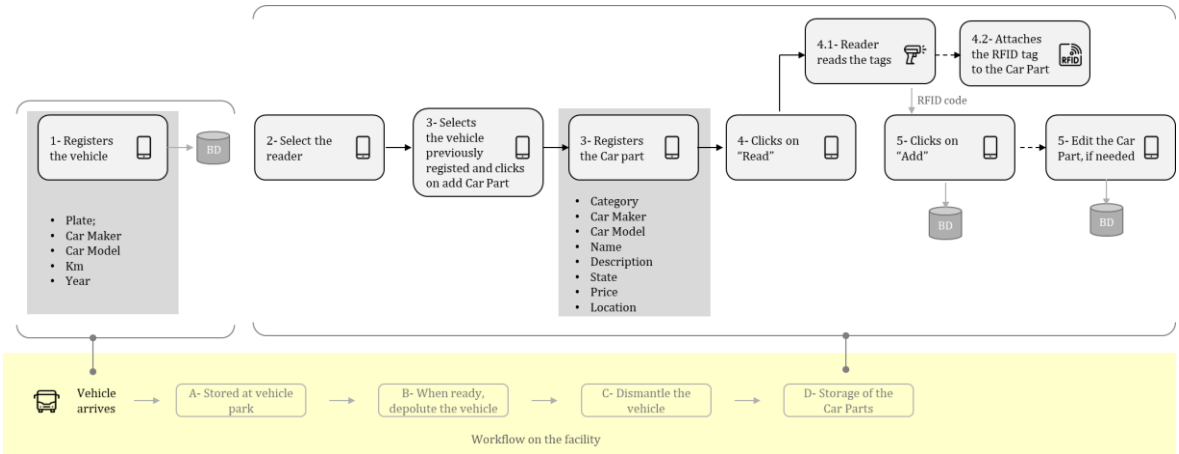

Figure 2 – Workflow block diagram

3.2 Proposed System infrastructure implementation

The proposed stock monitoring radio system infrastructure implementation (figure 3) consists of an edge device (RFID handheld reader), a MQTT broker (EMQX), a database (MONGODB), a RESTful API and a web application. The RFID reader uses the MQTT protocol to communicate through an existing WI-FI network with the web application and API. The calls to the API are made by the MQTT broker with the use of a data processing functionality they called rule engine. With the help of this functionality the messages to be sent to the API are filtered out and routed to the according API endpoint using the Hypertext transfer protocol (HTTP). All the write requests to the database are made using the REST API, however the EMQX MQTT broker like the API can read directly from the database. This is because the broker needs to be able to access the authorization credentials stored within the database. The REST API will handle all the CRUD (Create, Read, Update, Delete) operations on the database as well as the dynamic one-time only credentials to be used by the device, the devices setup passwords, the password hashing operations and the generation of the JSON Web Token (JWT). The database will store the users information such as the webapp login credentials, the broker credentials for the users and devices as well as the allowed subscription and publishing topics, the stock information such as the vehicles and vehicle parts, the stock analytics, the edge devices (RFID reader) information. All the requests and responses made to and from the API will require authentication and be made using the HTTP protocol. The web application communicates with the API using the HTTP protocol and with the RFID readers using MQTT protocol over WebSocket. The RFID reader communicates with the API through HTTP, in order to request the MQTT credentials to connect to the broker and will use the MQTT protocol to send the read RFID EPC number to the web application and to send battery and WI-FI received signal strength index (RSSI) to the broker which will in turn process and forward the signal message to the API to be saved in the database. The EMQX broker and the MongoDB database run inside a docker container each and are linked together with the use of Docker Compose tool. This will ensure isolation, increased security, system standardization, portability and easier scalability.

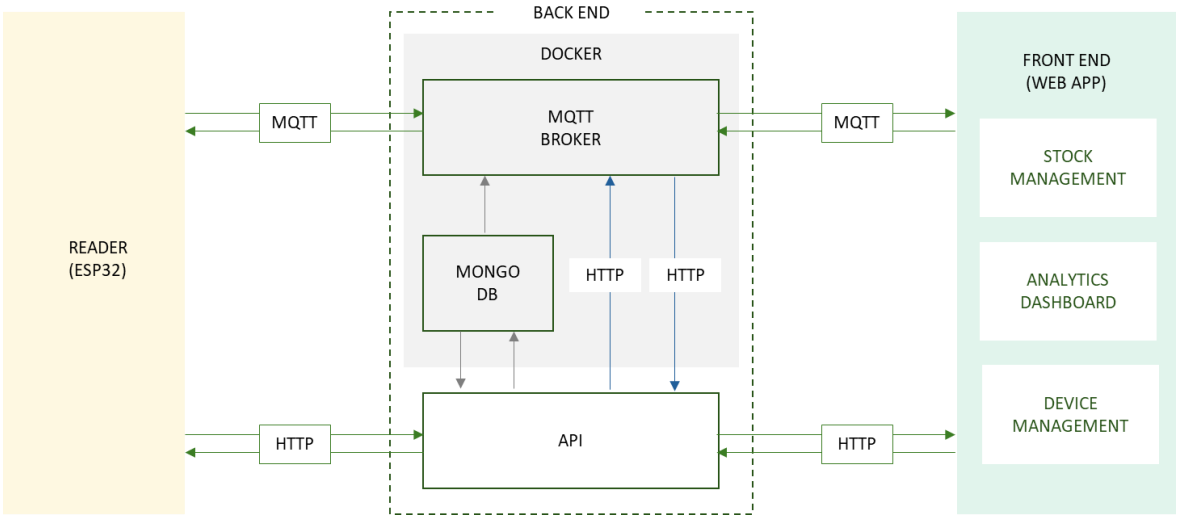

Figure 3 - System infrastructure diagram

The systems physical link can be better understood through the following diagram.

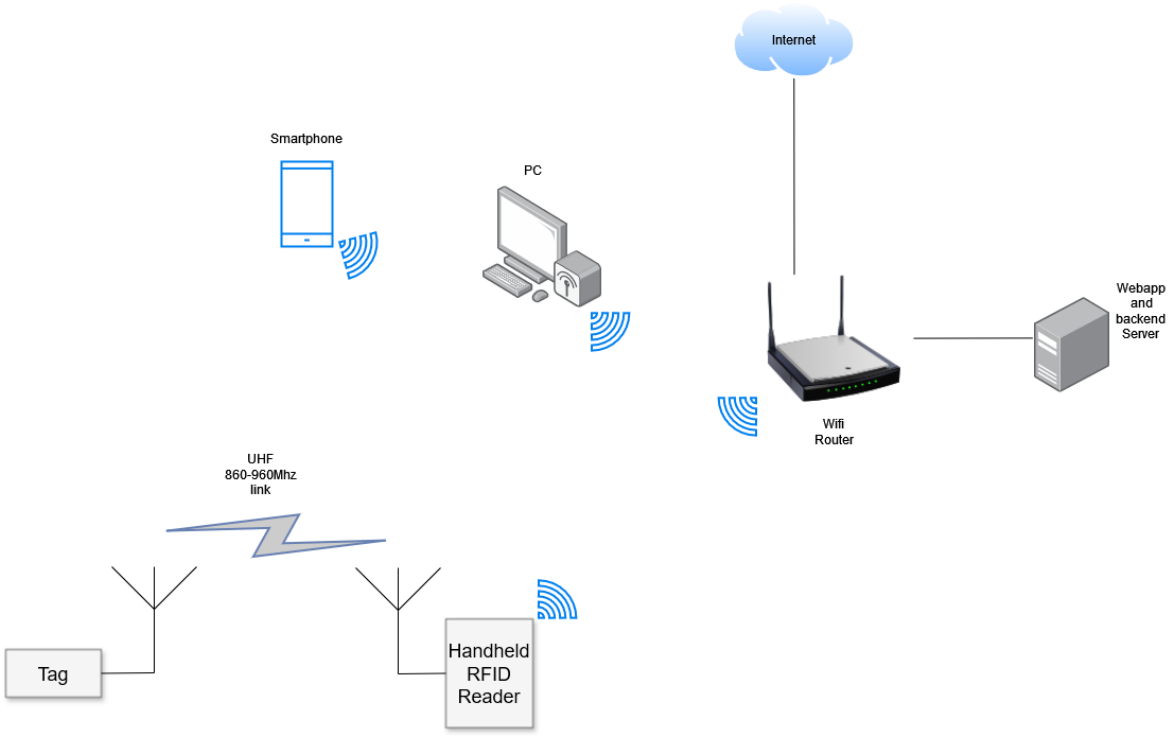

Figure 4 - Physical link diagram

3.3 Database implementation

Since the present work is a proof of concept, the DB was configured as a replica set with 3 servers as it uses fewer resources, is of lower complexity and if needed can easily be converted into a sharded cluster.

The database was named “LEETRM” and it follows Mongo DB structure. It is divided into 9 collections, with each collection containing a document for each item and each

document on the collection contains the values from all the fields defined. For example, the vehicle collection contains a document for each vehicle present in the database where the fields for each document hold the information such as the brand name. The following diagram (figure 5) pictures the DB structure and the links between collections.

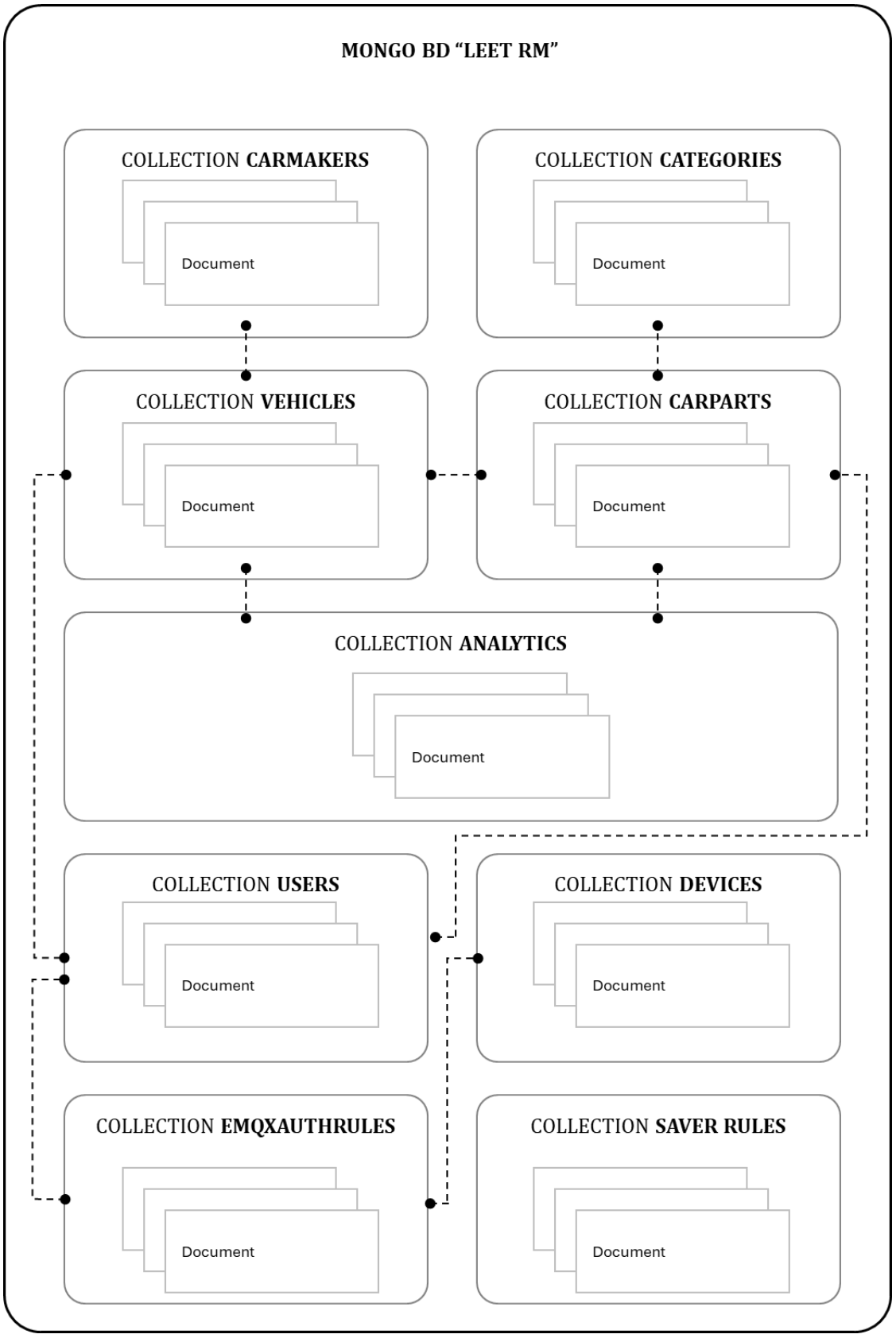

Figure 5 – DB structure and the link between collections

The “carmaker” collection (table 2) which holds the manufacturers name of ideally all existent vehicles and the respective model names. It contains the 2 following fields: the “name” field of type String containing the name of the manufacturers and the “models” field of type Array of Strings containing the models name from that manufacturer.

Table 2 – Example collection CARMAKERS document

COLLECTION **CARMAKERS** DOCUMENT		
Fields	**Field Type**	**Values (Example of a document)**
		{ "_id": {"$oid": "63eea353e789221a0b33a490"}, "name": "HONDA", "models": [
Document ID	Object ID	
Manufacturer name	String	
Model names	Array	
	String	"Accord",
	String	"Civic",
	String	...
	String	"Z"
		] }

The “carparts” collection (table 3) has 15 fields: the field “userId” of type String containing the value of the “_id” field of the user document from the “users” collection who registered the car part on the webapp; the field “vehicleId” of type String storing the value of the “_id” field of the vehicle document from the “vehicles” collection from which the car part was removed; the field “type” of type String which holds the name of the category group stored in the “categories” collection to which the part belongs; The field “whlocation” of type String holding the name of the location in which the part is being stored; the field “createdTime” of type Double to register the unix timestamp when the car part was registered on the webapp; a field called “delete” which will be either “true” or “false”, in order to preserve all the data that was once registered, but deleted by the user on the front end. The field “name” of type String holds the name of the car part, the filed “description” of type String stores a brief description for the car part, the “price” field of type Int32 which holds the price of the part, the “RFID” field of type String which holds the EPC read by the device and the “state” field of type String storing the quality of the car part and should be either “satisfatorio”, “bom” or “excelente”.

Table 3 - Example collection CARPARTS document

COLLECTION **CARPARTS** DOCUMENT		
Fields	**Field Type**	**Values (Example of a document)**
Document ID	Object ID	{ "_id": {"$oid": "66927bc7de915d90d05f2259"},
Document visibility	Boolean	"deleted": false
User ID	String	"userId": "63606653ae9bf054108523c3",
Vehicle ID	String	"vehicleId": {"$oid": "65d75e02fcd03834ac87c6ba"},
Car part category	String	"type": "motor",
Car part name	String	"name": "Motor 1.6",
Car part's description	String	"description": "Motor 1.6 Gasolina",
Car part's status	String	"state": "Bom",
Car part's price	Int32	"price": 1200,
RFID	String	"rfid": "392433",
Manufacturers' name	String	"carMaker": "Honda",
Models' name	String	"carModel": "Civic",
Storage Location	String	"whlocation": "a1p1",
Document creation time	Double	"createdTime": 1720875975923,
Document version	Int32	"__v": 0, }

The Categories collection (table 4) holds a reference list to classify the vehicle parts on 9 groups, such as “Suspensão”; “Direção”; “Motor”; “Carroçaria”; “Sistema Elétrico”; “Sistema de Travagem”; “Filtros/Óleo” e “Pneus”.

Table 4 - Example collection CATEGORIES document

COLLECTION **CATEGORIES** DOCUMENT		
Fields	**Field Type**	**Values (Example of a document)**
Document ID	Object ID	{ "_id": {"$oid": "64d4fbf74eb77a373c5cd5fb"},
Category's name	String	"name": "motor",
Document version	Int32	"__v": 0 }

The Devices collection (table 5) has 11 fields: the “chargeLeft” which holds a value of type Int32 for the charge level of the device, a field “rssi” with a value of Int32 type and holding the Wi-Fi signal strength, a “name” field of String type with the device name, “dId” of type String with the device Id(serial nr equivalent), a “userId” field of type ObjectId which holds the Id of user which inserted the device to the database through the webapp, the “createdTime” field of type Double holding the timestamp at the time the device was added to the DB, the “password” field of type String holding the password created by the API at the time the device was added to the database and which will serve as the base password for the device to be able to request the one time only password to the API in order to connect to the EMQX broker, and at last the

"lastUpdatedTime" of type Double holding the timestamp at the time of which the device information was last updated.

Table 5 - Example collection DEVICES document

COLLECTION **DEVICES** DOCUMENT		
Fields	**Field Type**	**Values (Example of a document)**
		{
Document ID	Object ID	"_id": {"$oid": "63606e583a849035f81fb676" },
Device in use	Boolean	"selected": false,
Device's battery	Int32	"chargeLeft": 20,
Wi-fi signal strength	Int32	"rssi": -59,
Device's name	String	"name": "A1",
Device ID	String	"dId": "1",
Created by userID	String	"userId": "63606653ae9bf054108523c3",
Doc creation time	Double	"createdTime": 1667264088078,
Webhook Password	String	"password": "eZS4Izvhpl",
Document version	Int32	"__v": 0,
Doc updated time	Double	"lastUpdatedTime": 1714665614044
		}

The emqxauthrules collection (table 6) contains 10 fields: The "publish" field of type Array is an array of strings with each string containing which topics the associated user, superuser or device will be able to publish to. The "subscribe" field also of type Array of Strings with each string containing which topics the associated user, superuser or device will be able to subscribe to. The "userId" containing the id of the user, the "username" field containing the one-time username created by the API to connect to the EMQX MQTT broker, the "password" field containing the one-time password created by the API to connect to the EMQX MQTT broker, the "type" filed which stores the type of credentials(user, superuser or device), the "time" field stores the unix timestamp for when the document was first created and the "updatedTime" containing the unix timestamp of the last time the document was updated.

Table 6 - Example collection EMQXAUTHRULES document

COLLECTION **EMQXAUTHRULES** DOCUMENT		
Fields	**Field Type**	**Values (Example of a document)**
Document ID	Object ID	{ "_id": {"$oid": "6360665dae9bf054108523ca"},
Allowed publish topics	Array	"publish": ["63606653ae9bf054108523c3/#"],
Allowed subscribe topics	Array	"subscribe": ["63606653ae9bf054108523c3/#"],
Created by userID	String	"userId": "63606653ae9bf054108523c3",
One time username	String	"username": "GyzbBzeVxS",
One time password	String	"password": "912VF5XGwn",
Type of credential	String	"type": "user",
Doc creation time	Double	"time": 1667262045086,
Doc updated time	Double	"updatedTime": 1720302079116,
Document version	Int32	"__v": 0 }

The users collection (table 7) contains 5 fields all of type String, the “name” field which stores the user name, the “email” which holds the email address of the user, and the “password” field which stores an hashed version of the password.

Table 7 - Example collection USER document

COLLECTION **USER** DOCUMENT		
Fields	**Field Type**	**Values (Example of a document)**
Document ID	Object ID	{ "_id": {"$oid": "63606653ae9bf054108523c3"},
WA User name	String	"name": "a",
WA User email	String	"email": "a",
WA User password	String	"password": "$2b$10$xyil3m5ByqrBsOxV.dV9wO/M2EZHhIu6JQFFToFJH3C9rMxWygRda",
Doc version	Int32	"__v": 0 }

The vehicle collection (table 8) contains 11 fields. The field “userId” of type ObjectId holding the reference Id of the use which inserted the vehicle information to the DB. The field “plate” of type String holds the number plate of the vehicle. The field "maker” of type String holds the brand name of the vehicle. The field “carModel” of type String which stores the model name of the vehicle. The field “km” of type Int32 holds the number of kilometers shown by the vehicle odometer. The field "year” of type String stores the date at which the vehicle was registered. The field “createdTime” of type Double holds the time at which the vehicle was inserted into the DB, in unix timestamp format. The field “deleted” of type Boolean which is either true or false indicating if it

has been deleted so that it won't be returned in queries. The field "depolluted" of type Boolean indicating whether the vehicle has been depolluted.

Table 8 - Example collection VECHICLES document

COLLECTION **VEHICLES** DOCUMENT		
Fields	**Field Type**	**Values (Example of a document)**
Document ID	Object ID	{ "_id": {"$oid": "65d75e02fcd03834ac87c6ba"},
Vehicle's dismantle status	Boolean	"abate": true,
Document visibility	Boolean	"deleted": false
Document	String	"userId": "63606653ae9bf054108523c3",
Vehicle's plate	String	"plate": "00-00-AB",
Manufacturers' name	String	"maker": "Honda",
Models' name	String	"carModel": "Civic",
Vehicle's Kilometers	Int32	"km": 123566,
Vehicle's year	String	"year": "2002",
Document creation time	Double	"createdTime": 1708613122460,
Document version	Int32	"__v": 0, }

The saverrules collection (table 9) contains 4 fields the "emqxRuleId" of type string and "status" of type Boolean fields. The former contains the id of the EMQX rule created by the API in the EMQX broker to forward the devices information to its endpoint through HTTP, while the former contains the status of said rule which can be either true or false

Table 9 - Example collection SAVERRULES document

COLLECTION **SAVERRULES** DOCUMENT		
Fields	**Field Type**	**Values (Example of a document)**
Document ID	Object ID	{ "_id": {"$oid": "65451f788216ff31989a1a9a"},
Ruleid EMQX webhook	String	"emqxRuleId": "rule:e59e284f",
Rule enabled status	Boolean	"status": true,
Doc version	Int32	"__v": 0 }

All the documents contain the fields "_id" and "__v". These are automatically added by MongoDB and the first refers to the ID of the document and the last refers to the document version.

3.4 API

The API (Application Programmable Interface) is a piece software which serves as an interface, enabling communication between different computer applications or components. There are 4 different categories of APIs based on the protocol or architecture. These are, SOAP which stands for Simple Object Access Protocol, RPC meaning Remote Procedure Call, WebSocket and REST which is short for Representational State Transfer [9] [10].

A REST API was chosen to be used in the proposed system implementation. The API was written in JavaScript using the nodeJS framework Express, and makes use of the libraries, jsonwebtoken, bcrypt, mongoose, morgan, cors, colors, axios. According to the official nodeJS website: “Node.js® is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts” [11]. And in the context of this work, it was used in conjunction with the framework Express to build the REST API. Express is a nodeJS framework which simplifies and expands nodeJS functionalities making it easier to develop web applications, APIs and middlewares.

The API will be responsible for handling the database read and write requests from the web application, the write requests from the forwarded device information sent by EMQX broker, generation of the JWT token, update analytics, and create the EMQX webhook resources and rules, as well as any data processing needed to store or retrieve the database information like hashing the users passwords, generate one time credentials for MQTT connection.

3.5 Technologies and protocols used

3.5.1 EPCglobal class-1 Generation-2

The EPCglobal class-1 Generation-2 is an air interface protocol developed by EPCglobal in 2004 and defines the physical and logical requirements and procedures for communication between RFID interrogators and tags working in the 860 to 960Mhz frequency range in compliance with the relevant international standards organization (ISO) standards, such as ISO 18000-63 and regional and local regulations regarding allowed frequency spectrum usage.

At the physical link an interrogator communicates with a Tag by means of a modulated RF carrier wave. The Tag communicates with the interrogator by backscatter modulating an unmodulated carrier wave sent by the interrogator which after sending the carrier wave will be listening for a reply. The modulation used by the interrogator to talk to the Tag can be either double sideband amplitude shift keying (DSB-ASK), single sideband amplitude shift keying (SSB-ASK) or phase reversal amplitude shift keying (PR-ASK) using pulse interval encoding (PIE) format. The modulation which the Tag can use to reply to the interrogator can be either amplitude shift keying (ASK) or frequency shift keying (FSK) with either FM0 or Miller-modulated subcarrier encoding. The communication link is half-duplex, and the is made on the principle of the interrogator talks first and the tag responds. The tables 6-1 and 6-2 of the EPC class 1 gen 2 protocol, give more detailed information on the communication from Interrogator to Tag and vice versa and can be found at the official GS1 Global standards website[1].

At the logical level it defines the tag memory configuration, consisting of 4 sections of different memory banks being the reserved, electronic product code (EPC), transponder ID (TID) and user memory. The reserved memory bank contains the kill and or access passwords. The EPC memory bank contains the storedCRC, the storedPC, the EPC and an optional extended protocol control (XPC). The storedCRC is a cyclic redundancy error code calculated by the tag either when an interrogator writes or overwrites the EPC bits in the EPC memory bank or when the tag powers up. The storedPC or the protocol control provides metadata information about the EPC data and encoding such as the EPC length the version of the EPC protocol and other information needed for correct interpretation by the reader of the data stored in the tag memory, the EPC or electronic product code which uniquely identifies the tag, and XPC is an optional field indicating the support and options of the extended product code. It also defines the command set that readers can use to communicate with the tags and tag responses as well as the anti-collision mechanism. A detailed information about the command set can be found on the table 6-29 of the EPC class 1 gen 2 protocol[1].

[1] https://ref.gs1.org/standards/gen2/

3.5.2 MQTT

MQTT stands for Message Queuing Telemetry Transport and is a lightweight messaging protocol, based on a publisher subscriber logic designed to be used in low-bandwidth, high latency, unreliable networks by resource-constrained devices such as small sensors and mobile devices [12]. It is bi-directional due to its publish and subscribe logic, secure thanks to support for TLS/SSL encryption as well as authentication and authorization mechanisms, reliable as it implements 3 different levels of QoS levels as well as session awareness and persistent connections and has broad language support.

The routing of the MQTT messages is based on topics. The topic structure follows a similar hierarchy to URL paths being subdivided using a slash (/). MQTT supports the use of two wildcards: + and #. The former represents a single level wild card, matching any substring in between two slashes or, in other words, any substring in the level where the wild card is. The latter cannot be used in between two slashes as it indicates multiple level wild card and as such can only be used as a termination substring, since it implies a wild card matching all substrings of all subsequent levels. MQTT works by having an MQTT broker which can be thought of as a network router to which the clients can connect. Besides the connection and disconnection of client devices and the routing of messages the broker also handles subscription and unsubscription requests. When a client connects to the broker it can request to subscribe to a topic and/or publish to a topic. Communication between devices happens only when one publishes to a topic which has been subscribed by another device. A topic can be subscribed by multiple devices and multiple devices can publish to the same topic. Also, a device can publish and subscribe to multiple topics [13].

In the present work the topic structure follows the logic:

userId/dId/+/(actdata/sinfo/notif). userID refers to user ID who made the request, dId to the device ID to which the request are being sent or from the replies are being sent,+ is an unimplemented variable, actdata indicates it's the message contains an actuation request to de device such as read an RFID tag, sinfo is the route to which the device sends its info such as battery information and Wi-Fi signal strength(RSSI), and notif is a route to which the device sends information to be viewed as a notification at the front end such as an RFID tag read unsolicited from the front end such as a push of a physical button on the device.

3.5.3 UART

UART stands for Universal Asynchronous Receiver Transmitter and is a communication protocol [14] to interchange serial data from one device to another. It is serial because data is transmitted sequentially, one bit at a time. As the name indicates, it is asynchronous, meaning that the devices don't need to share the same clock signal, and use predefined baud rates instead, to define the bit timing. It uses two wires for communication and a shared ground, with one being the transmit and the

other one being the receive. Regarding communication direction, it can be either be simplex, half-duplex or full-duplex [7].

3.6 Hardware

3.6.1 RFID handheld reader

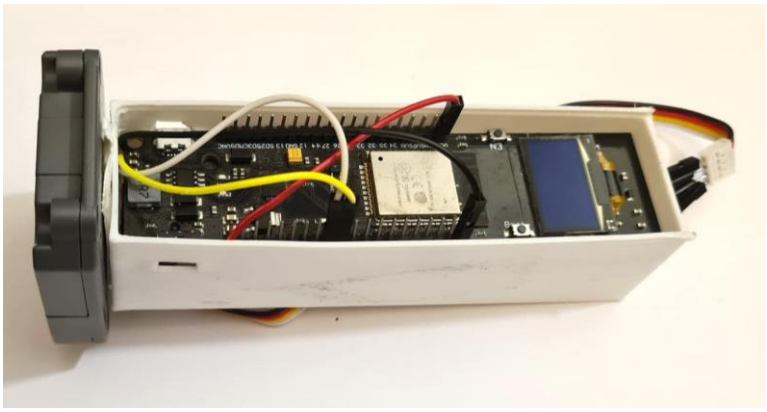

Figure 6 - Handheld RFID seen from the side

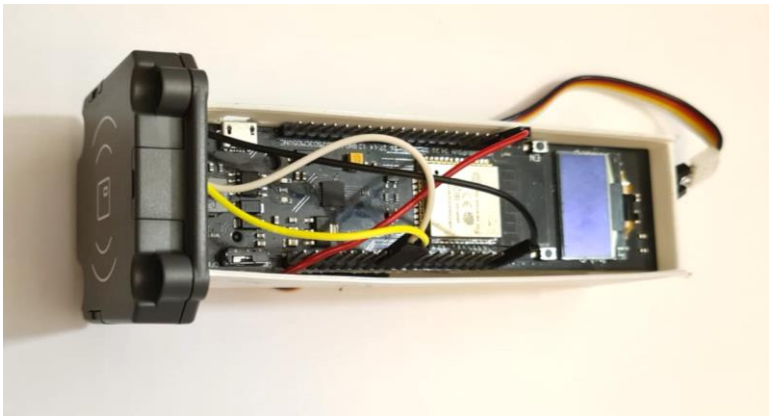

Figure 7 - Handheld RFID reader seen from the top

The following image (figure 8) shows the connections between the TTGO ESP32 module and the UHF reader module:

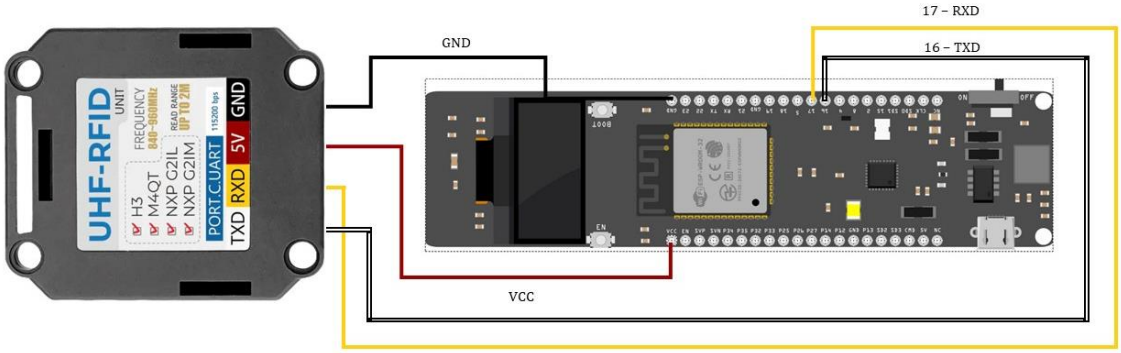

Figure 8 - Handheld RFID reader connections schematic

The RFID reader device consists of a TTGO ESP32 module with OLED display and 18650 battery holder development board with 18650 battery and an M5Stack UHF RFID reader module (JRD-4035).

3.6.2 TTGO ESP32 module with OLED display and 18650 battery holder

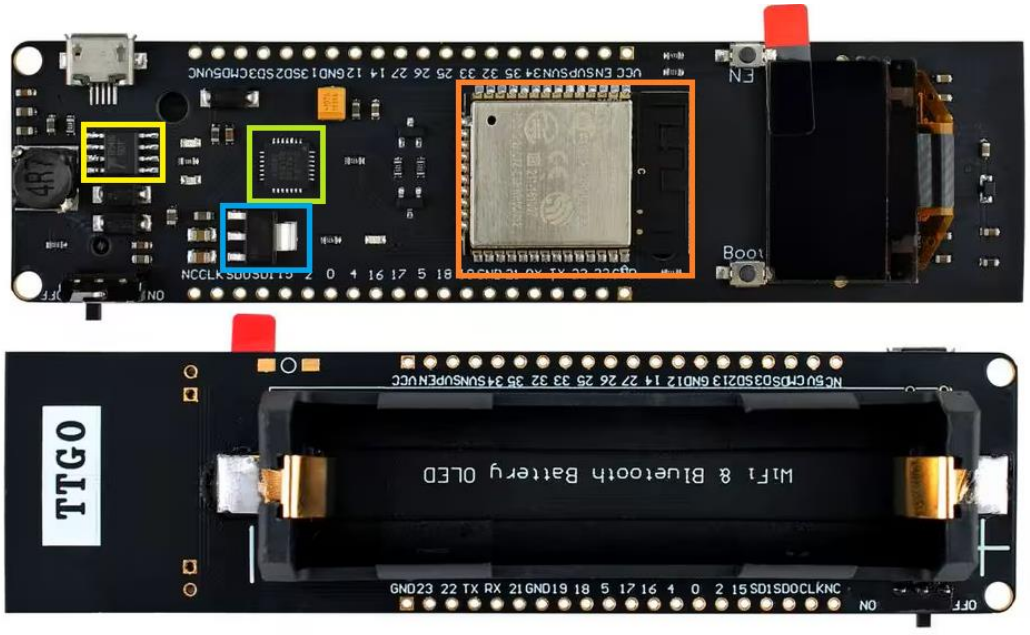

Figure 9 –TTGO dev board with components highlighted. Orange – ESP32, Blue AMS1117 3.3v, Green – CP210X, Yellow – TP5410

3.6.3 ESP32

The ESP32 is a SoC developed by the company Expressif and is based on the Xtensa LX6 dual core microprocessor with configurable frequency from 80 to 240MHz, high performance and low energy consumption accompanied by a rich set of peripherals

and connectivity such as Wi-Fi 802.11 b/g/n, Bluetooth, Bluetooth LE, high-speed SPI, UART, I2C and many more. It is also able to run freeRTOS as the operating system. The table below summarizes its characteristics (table 10):

Table 10 - Esp32 characteristics, adapted from [14] [15]

Number of cores	2 (dual core)	
Clock frequency	Up to 240 MHz	
Pins	30, 36, or 38 (depending on the model)	
Certification	RF certification	See certificates for ESP32-WROOM-32
	Wi-Fi certification	Wi-Fi Alliance
	Bluetooth certification	BQB
	Green certification	RoHS/REACH
Test	Reliability	HTOL/HTSL/uHAST/TCT/ESD
Wi-Fi	Protocols	802.11 b/g/n (802.11n up to 150 Mbps)
		A-MPDU and A-MSDU aggregation and 0.4 µs guard interval support
	Center frequency range of operating channel	2412 ~ 2484 MHz
Bluetooth	Protocols	Bluetooth v4.2 BR/EDR and Bluetooth LE specification
	Radio	NZIF receiver with –97 dBm sensitivity
		Class-1, class-2 and class-3 transmitter
		AFH
	Audio	CVSD and SBC
Hardware	Module interfaces	SD card, UART, SPI, SDIO, I2C, LED PWM, Motor PWM, I2S, IR, pulse counter, GPIO, capacitive touch sensor, ADC, DAC, Two-Wire Automotive Interface (TWAI®), compatible with ISO11898-1 (CAN Specification 2.0)
	Integrated crystal	40 MHz crystal
	Integrated SPI flash	4 MB
	Operating voltage/Power supply	3.0 V ~ 3.6 V
	Operating current	Average: 80 mA
	Minimum current delivered by power supply	500 mA

3.6.4 Ams1117 3.3v

The AMS1117 3.3v is a low dropout linear voltage regulator made by Advanced Monolithic Systems. It has a fixed output voltage of 3.3v. It offers a relatively high output current of 1A and an absolute maximum of 1.3V dropout voltage, decreasing at lower currents.

3.6.5 CP210X

The CP2102 is a USB to UART bridge controller with a high level of integration made by Silicon Labs. It is used to convert serial data from USB protocol to UART.

3.6.6 TP5410

The TP5410 is a single-cell lithium-ion battery charger and constant 5V boost controller made by NanJing Top Power ASIC Corp. It can charge 18650 batteries up their full voltage of 4.2V with a maximum programmable current of 1A. It features an anti-backflow circuit, charging temperature protection, input power current limiting loop, and over-current shutdown functions. It also supplies an output voltage of 5V and a maximum output current of 1A.

3.6.7 M5Stack UHF RFID reader module (JRD-4035)

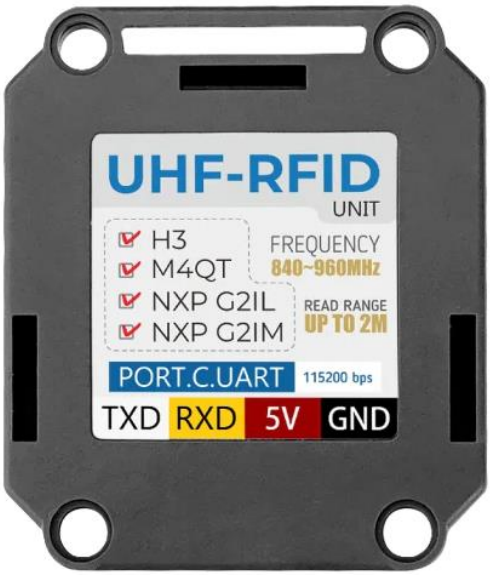

Figure 10 - UHF reader module

The JRD-4035 (figure 10) module solution is an Ultra High Frequency reader module compatible with the EPCglobal UHF Class 1 GEN 2 and ISO 18000-6c protocol standards. It's almost a complete solution which features a built-in ceramic antenna and a MagicRF M100 SoC. According to the datasheet, it features an optimized RF design to achieve low power consumption and high performance with a transmission

power of 100mw. The company claims it can cover a distance of 1.5m even though, in real tests, at least with the tags available it falls short of that claim and is heavily dependent of the operating region frequency. The communication with the module is made with the use of a serial interface using the UART protocol. The following table summarizes the module characteristics (table 11):

Table 11 - UHF reader module characteristics Source: m5-docs (m5stack.com)

Specifications	**Parameters**
Detection distance	up to 2M
Working spectrum range	840-960MHz
Air interface protocol	EPCglobal UHF Class 1 Gen 2 / ISO 18000-6C
Work area support	US, Canada and other regions following U.S.FCC. Europe and other regions following ETSI EN 302 208, Mainland China, Japan, Korea, Malaysia, Taiwan
Working spectrum range	840-960MHz
Output power range	18-26 dBm
Tag cache area	200 tags
Communication protocol	UART (Baud rate: 115200bps)

The UHF reader module supports the following operations (table 12):

Table 12 - UHF reader module operations

Operations	
Get Information	Set up working channel
Single polling instruction	Get the working channel
Multiple polling instructions	Set to automatic frequency hopping mode
Stop multiple polling instructions	Insert the working channel
Set the SELECT parameter instruction	Acquire transmitting power
Get the SELECT parameter	Set the transmitting power
Set the SELECT mode	Set up transmitting continuous carrier
Set communication baud rate	Module hibernation
Set work area	Read label data storage area
Acquire work locations	Write the label data store

Of these operations only the operations for getting module information, setting up the work area and the single polling of instructions were used. A detailed table with the operations and UART commands from the manufacturer can be found in the annex section.

3.7 Software used

3.7.1 EMQX

EMQX is an open-source, highly scalable, high performance and low latency MQTT broker, written in Erlang. It supports many different protocols such as MQTT, HTTP, QUIC, WebSocket and more [15]. It is also secure, offering the possibility of using various backends like a database to manage authentication and authorization credentials and of configuring secure bi-directional MQTT communications over TLS/SSL. It is customizable offering support for already developed or custom Plugins. Has a web-based dashboard and REST API for managing and monitoring the broker and an SQL based rule engine enabling the processing of messages, such as filtering and routing, in real time [16]. The EMQX application is shown in the image below (figure 11).

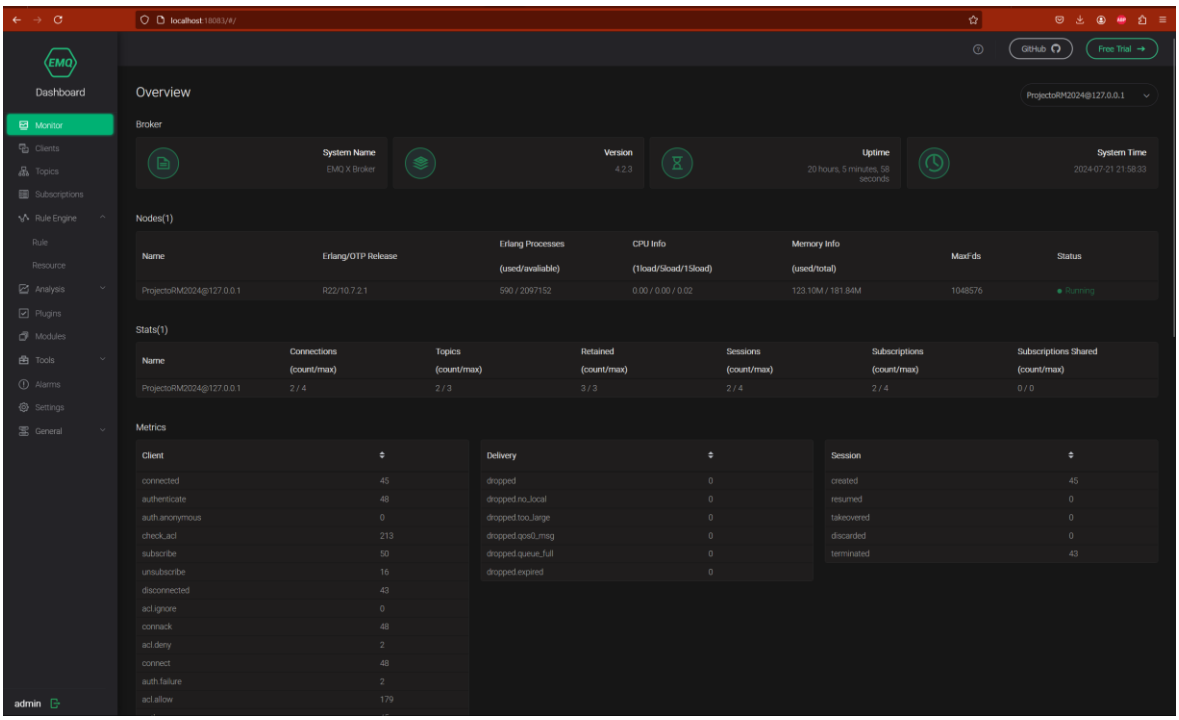

Figure 11 - EMQX Broker

3.7.2 Docker

According to wikipedia “Docker is a set of platform as a service services that use OS-level virtualization to deliver software in packages called containers” [17]. It is developed by Docker, Inc. and is based on the open-source Docker Engine. Containers, or container images are software packages containing everything needed to run an application being it code, system tools and libraries, runtime and the appropriate

settings. Using Docker containers offer several benefits including being highly portable since by including system tools, libraries and runtime they become standardized and will work independently of system OS or even OS versions and settings. Compared to virtual machines, containers are lightweight since they share the machine OS system kernel removing the need of having OS virtualization. They also make applications more secure by providing OS or even container to container isolation [18]. The figure below (figure 12) shows the docker desktop application.

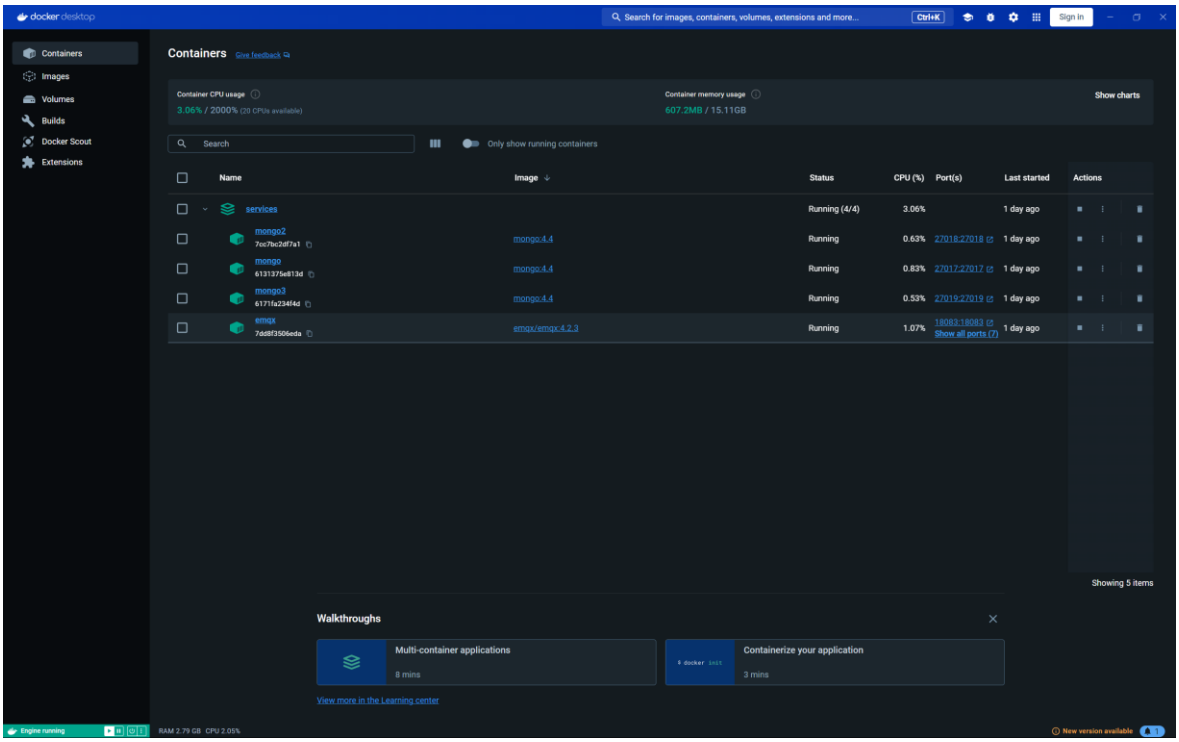

Figure 12 - Docker Desktop

3.7.3 Visual Studio Code

Visual Studio Code is a lightweight and powerful source code editor, developed by Microsoft [19]. It supports several different programming languages including C, C++, java, JavaScript, HTML, CSS, Python, Rust, etc. It features IntelliSense autocompletion for JavaScript, syntax highlighting, bracket matching, code folding and configurable snippets out of the box and its features can be extended with custom extensions or with extensions obtained from the VS Code Marketplace. The extensions, Vue – official v2.0.26, Vue VSCode Snippets v3.1.1, npm IntelliSense v1.4.5 and PlatformIO IDE v3.3.3 were used. Vue – official extension offers syntax highlighting, typescript support and IntelliSense for Vue template expressions and component props. Vue VSCode snippets offers snippets allowing for faster coding, removing the need to write repetitive pieces of code. Npm IntelliSense makes importing of npm modules and packages faster through the implementation of an autocomplete feature. PlatformIO is a fully featured

embedded software development environment. It supports many different embedded system platforms such as the Espressif 32 and 8266, Atmel AVR and SAM, Microchip PIC32, ST STM32 and many others. It also supports many platform specific frameworks such as Arduino, ESP-IDF, FreeRTOS SDK, STM32Cube and many others. It is cross-platform, supports debugging, unit testing, static code analysis, remote development, C/C++ IntelliSense, library manager, serial port monitor and a built-in Terminal. The image below (figure 13) shows VScode running the nuxtJS webapp.

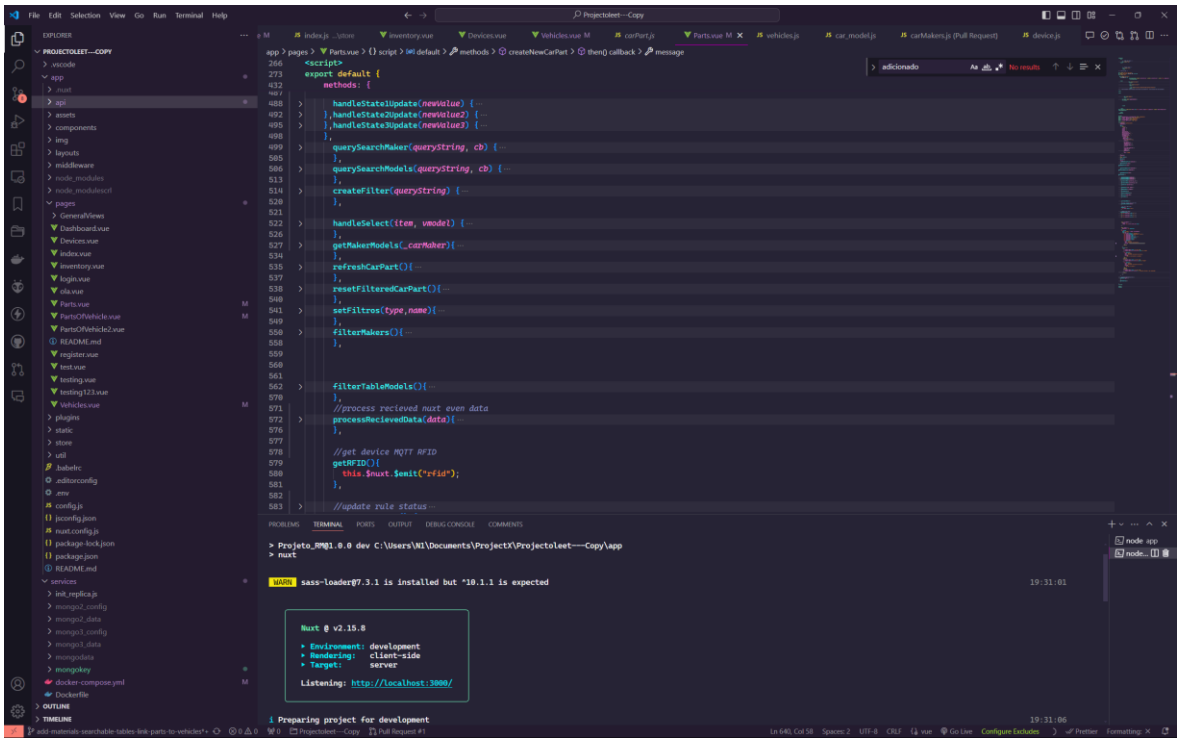

Figure 73 - Visual Studio Code running web app

The image below (figure 14) shows the extension PlatformIO for visual studio code.

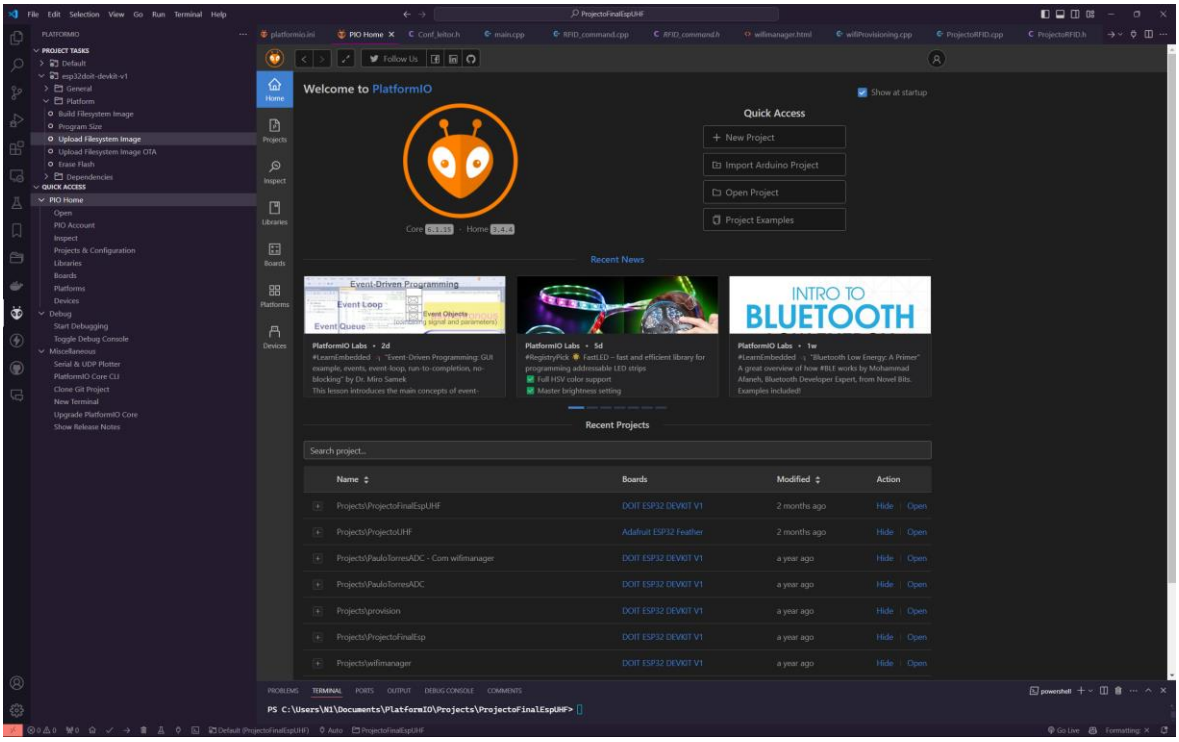

Figure 14 – PlatformIO

3.7.4 MongoDB Compass

MongoDB Compass (figure 15) is a free graphical user interface developed by MongoDB Inc, as a way to easily visualize and manage MongoDB databases. It can be used to query data, visualize and analyze schemas and data distribution, evaluate query performance, create and manage indexes and manage and create aggregation pipelines as well as using the MongoDB shell [20]. The image below shows the MongoDB Compass application connected to the LEETRM database:

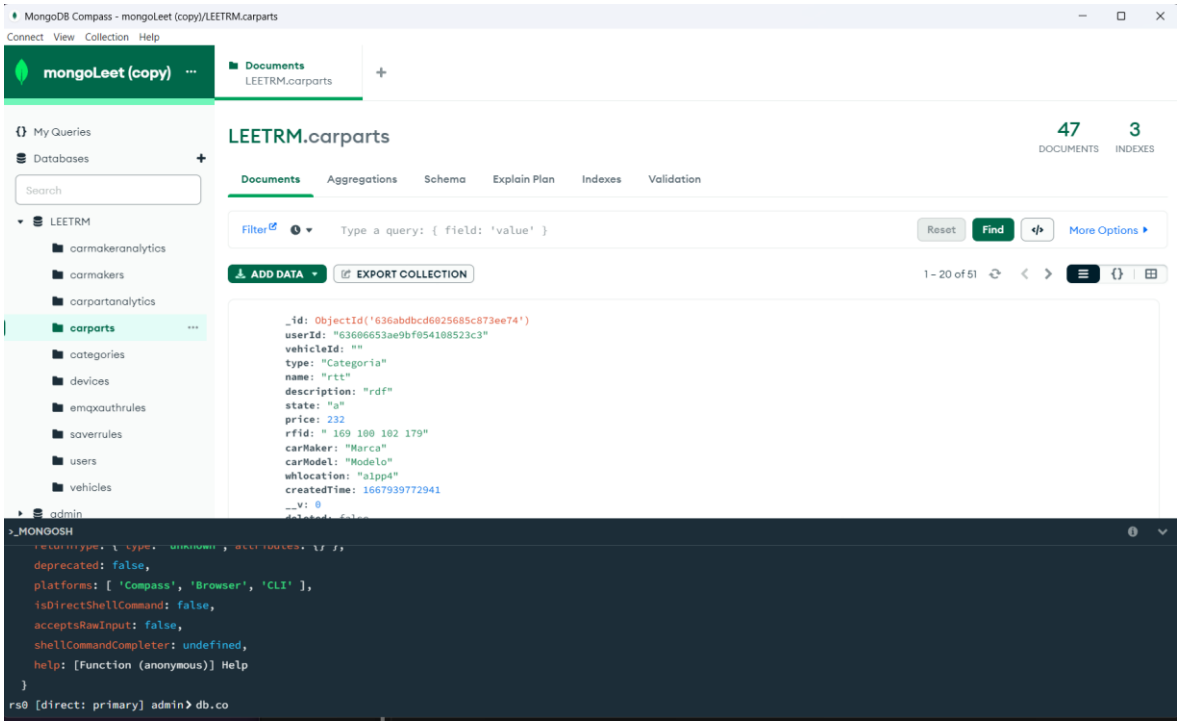

Figure 15 - MongoDB compass aplication

3.7.5 NPM

Npm or node package manager is a javascript program used to install and manage library dependencies and packages. For this work the npm version 6.14.16 was used.

3.7.6 NVM

Nvm is a software to easily switch between different nodeJS and npm versions. The nodeJS version used in this project was v12.22.12 and the npm version was v6.14.16.

3.7.7 Programming Languages and Frameworks used

The web app was developed using the NuxtJS framework. The NuxtJS framework is based on VueJS which is itself a JavaScript framework and makes it easier and faster to develop web applications by automating routing based on the directory structure, auto

importing of components, composables and utils, support for image, font and script optimizations and support for CSS modules Sass, PostCSS and more [21] [22]. It has built-in error handling and logging, data fetching, state management, internal messaging system and many more features. The web app frontend look and feel was based on the free version of the NuxtJS template developed by Creative Tim called Black Dashboard [23].

The firmware for the hardware, namely the ESP32 code, was developed using Visual Studio Code with the PlatformIO extension and using the Arduino C framework and the libraries:

- bblanchon/ArduinoJson@^6.18.5,
- knolleary/PubSubClient@^2.8,
- me-no-dev/AsyncTCP@^1.1.1,
- ayushsharma82/AsyncElegantOTA@^2.2.7 ,
- ottowinter/ESPAsyncWebServer-esphome@^3.0.0

ArduinoJson is a library which makes the use of JSON documents easier to work with using the Arduino framework. The PubSubClient is an MQTT client library, and it will be used to handle the RFID handheld reader MQTT connection, the AsyncTCP library implements fully asynchronous TCP and is the base library for ESPASyncWebServer, the ESPAsyncWebServer library makes it simple to implement a web server and it will be used to serve the web page for the user to enter the Wi-Fi configuration and the APIs MQTT credentials endpoint configuration. The AsyncElegantOTA library simplifies the implementation of Over The Air (OTA) firmware updates.

3.8 Web application

3.8.1 Steps to start the Web application

Before the web application can be started for the first time, the backend must be preconfigured and its docker images pulled. To that end the file docker-compose.yml must be edited. Inside this document one must configure the desired EMQX dashboard and app passwords, the database connection from EMQX to MongoDB so the broker can obtain the access control list, and the links from the internal docker ports to the host machine exposed ports. After the docker-compose.yml is properly configured the command “docker-compose up” should be run from a terminal inside the services folder working directory. For the front end to be started for the first time all needed dependencies must be installed. To do so, a terminal is run from inside the folder called app and the command: “npm install” is run. After all dependencies are installed the command “npm run devn” and “npm run dev” should be run in separate terminals. The former command starts the API while the former command starts the web application. These terminals should be left running and unclosed permantly or until the API and webapp are to be shutdown.

3.8.2 Web application workflow

When the webapp is opened for the first time the user is greeted with a login page (figure 16).

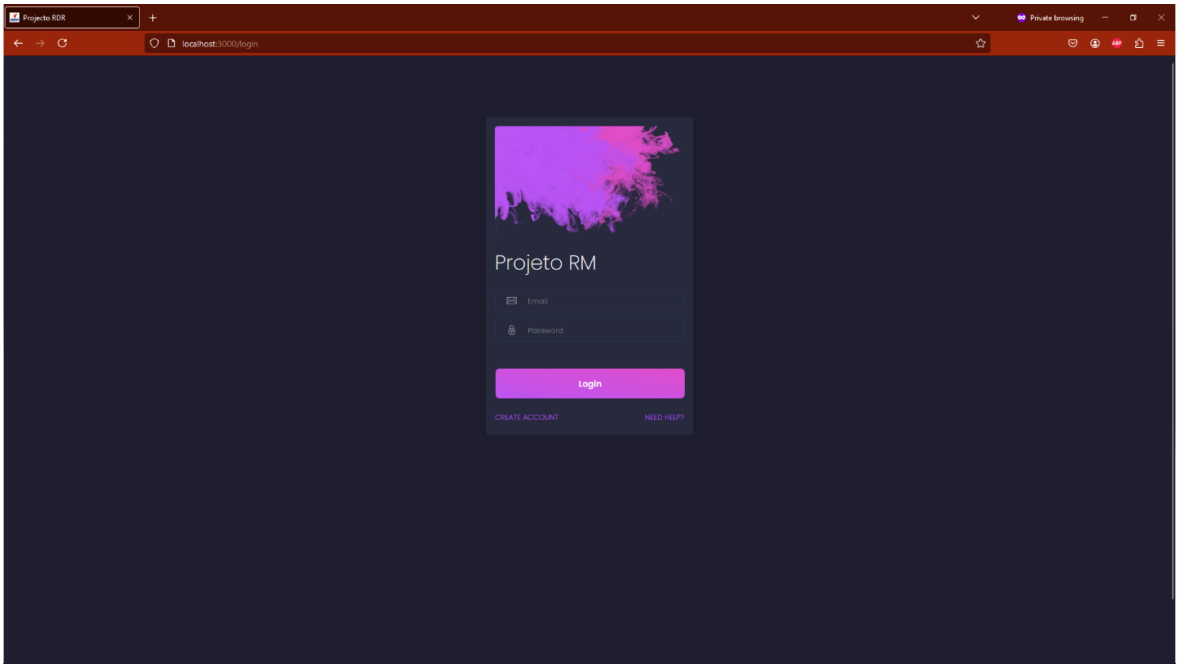

Figure 16 - Web app login page

If it is not yet registered it can click the registration link and create a user profile (figure 17).

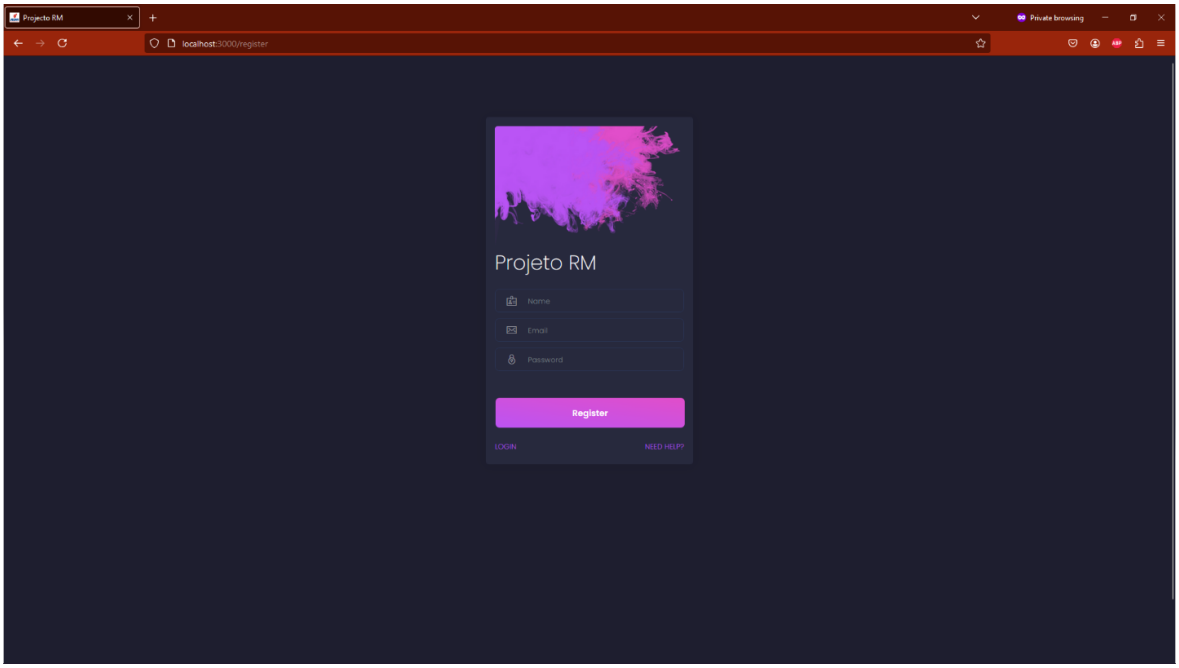

Figure 17 - Web app register page

After registering the user will need to login, and after a successful login the user will be greeted with the dashboard page (figure 18) which displays relevant information with regards to stock distribution, quantity, and value:

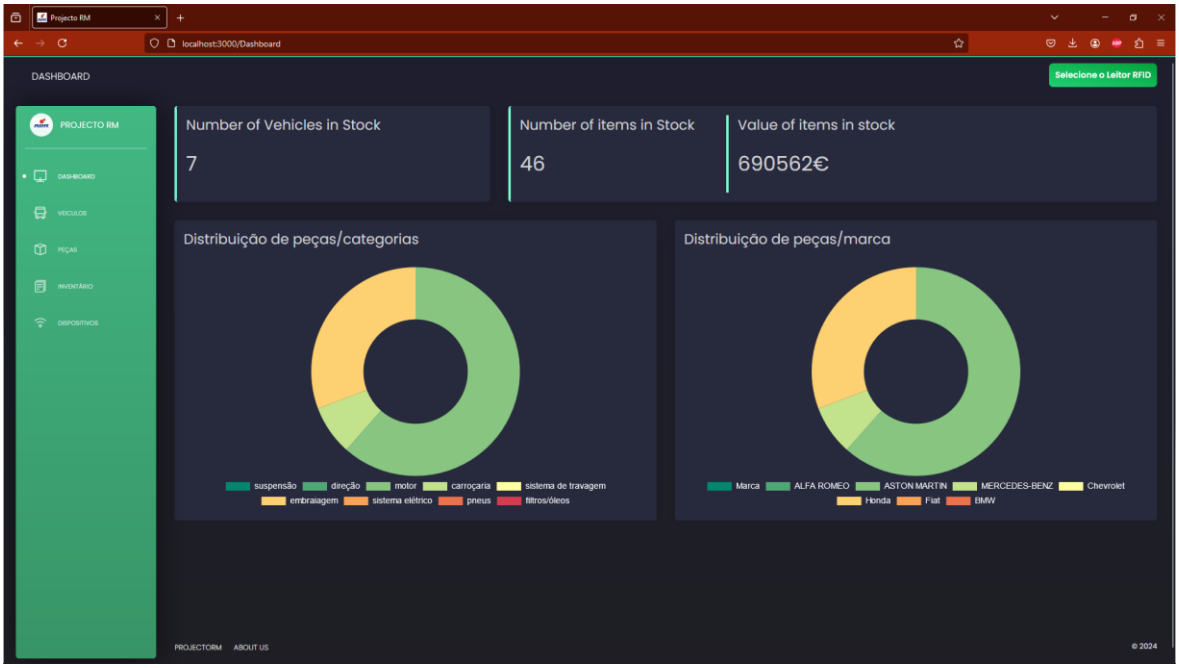

Figure 18 - Web app dashboard page

The following figure (figure 19) represents the site map.

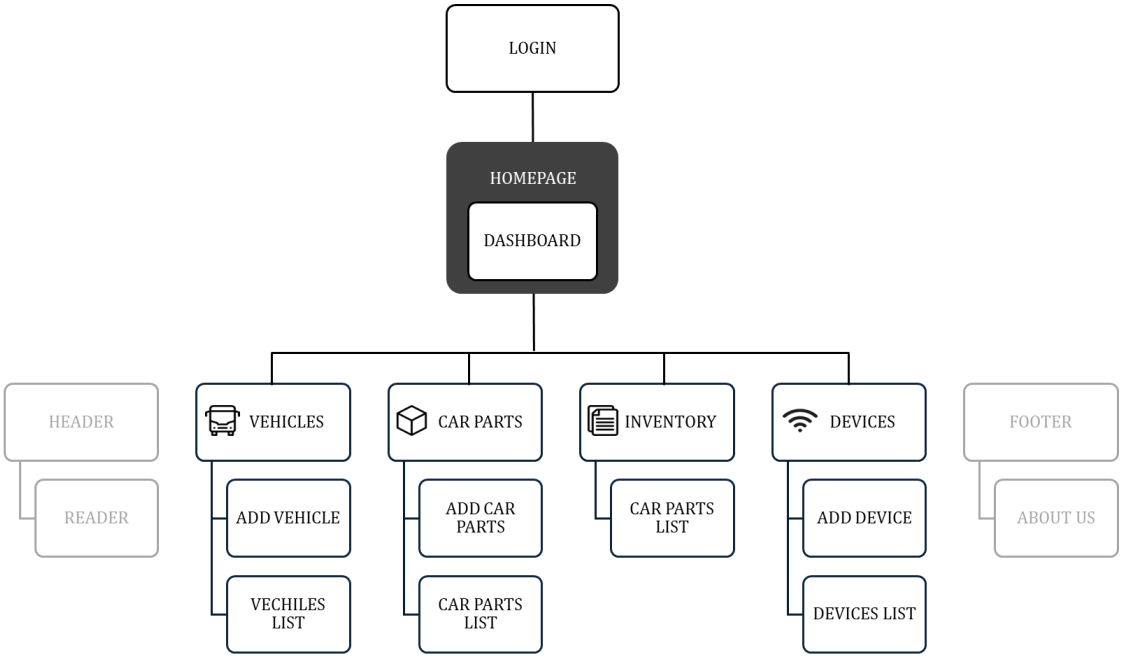

Figure 19 - Web app site map

3.8.3 RFID handeld reader management

To manage the devices the user must click the “dispositivos” sidebar link where it will be routed to the device management page (figure 20):

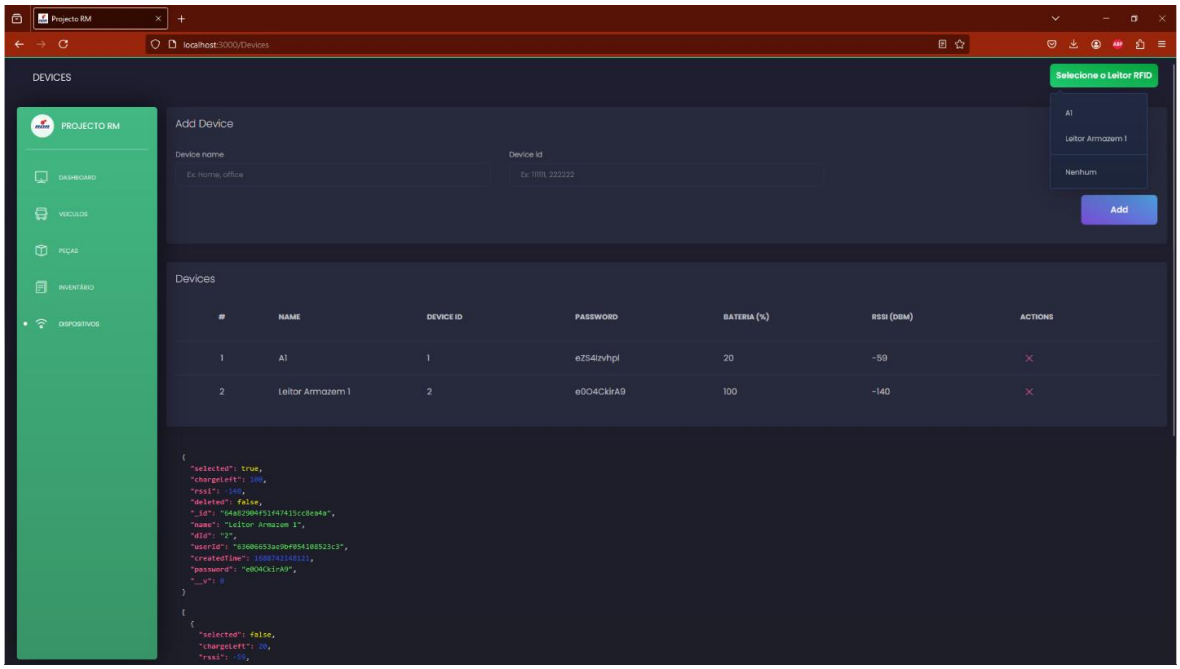

Figure 20 - Web app devices management page

To be able to use a device for the first time the device must first be registered in the database. To do so the user must enter the device’s name the user wishes to give, and the device dId, which is the unique identification number of the device attributed

through the firmware. After being processed by the API and stored in the database, the table with the devices list is updated and shows the device information, including the necessary password to be used in the device configuration for it to be able to retrieve the one-time credentials to connect to the MQTT broker. If a user wishes to remove a device from the database, it can do so by clicking the X button found in said device's row at the "actions" column.

After the device being registered in the database it will be possible to select it from any page through the green drop-down menu called "Selecione o leitor RFID" on the top right corner of the web app.

Also to be able to send and receive RFID information to and from the reader, the desired device must first be selected using the method described above.

3.8.4 Stock inventorying workflow

With the user logged in and following the workflow walkthrough described in the beginning of this chapter, to register a new vehicle the user selects the "veículos" sidebar link. It will then be presented with a page (figure 21) where it will be able to add the vehicle information to the database as well as visualize the vehicles already present in the DB.

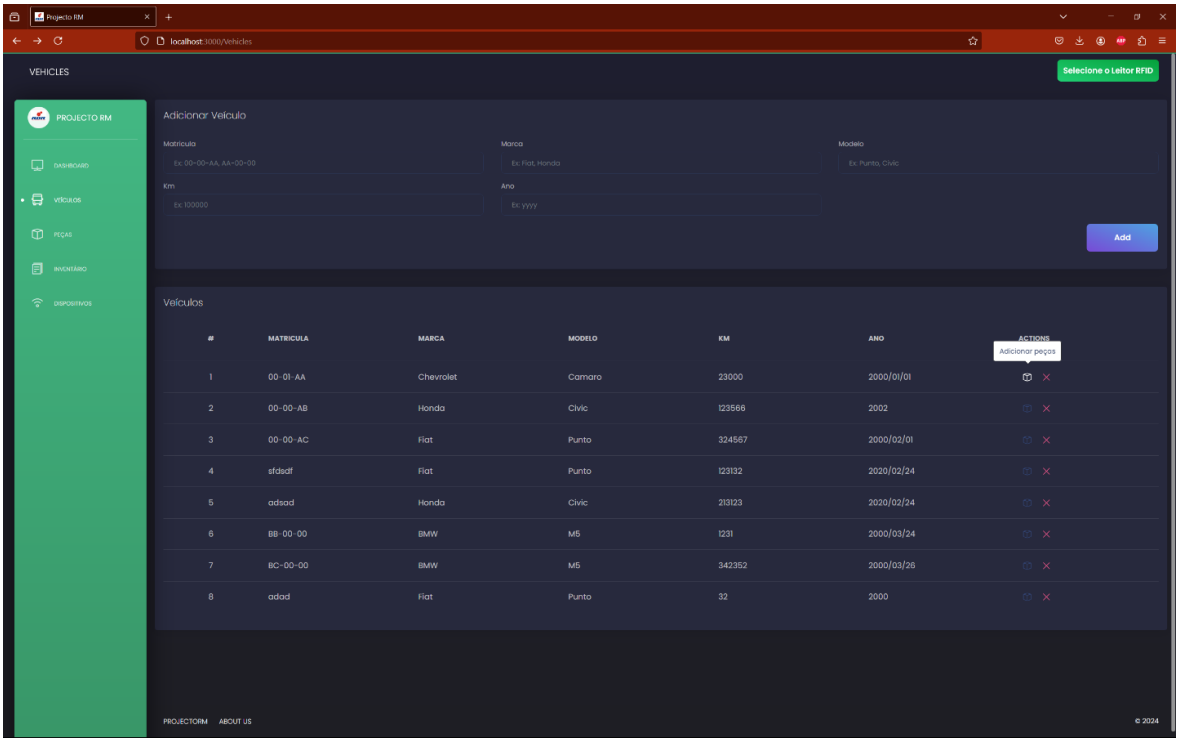

Figure 21 - Web app devices management page

If the user wants to start adding item information regarding items taken from a dismantled vehicle present in the database, the user can travel to the page for doing so by clicking the "adicionar peças" icon located on the actions tab of the wanted vehicle

displayed on the vehicles table. After clicking on said icon, the user will be routed to the page(figure22) where it will be able to add the vehicle part information for the selected vehicle. This page has a section with a table with all the parts associated with that vehicle and a section where the user can add the new part information and request the handheld RFID read to read the tag to be associated with said part.

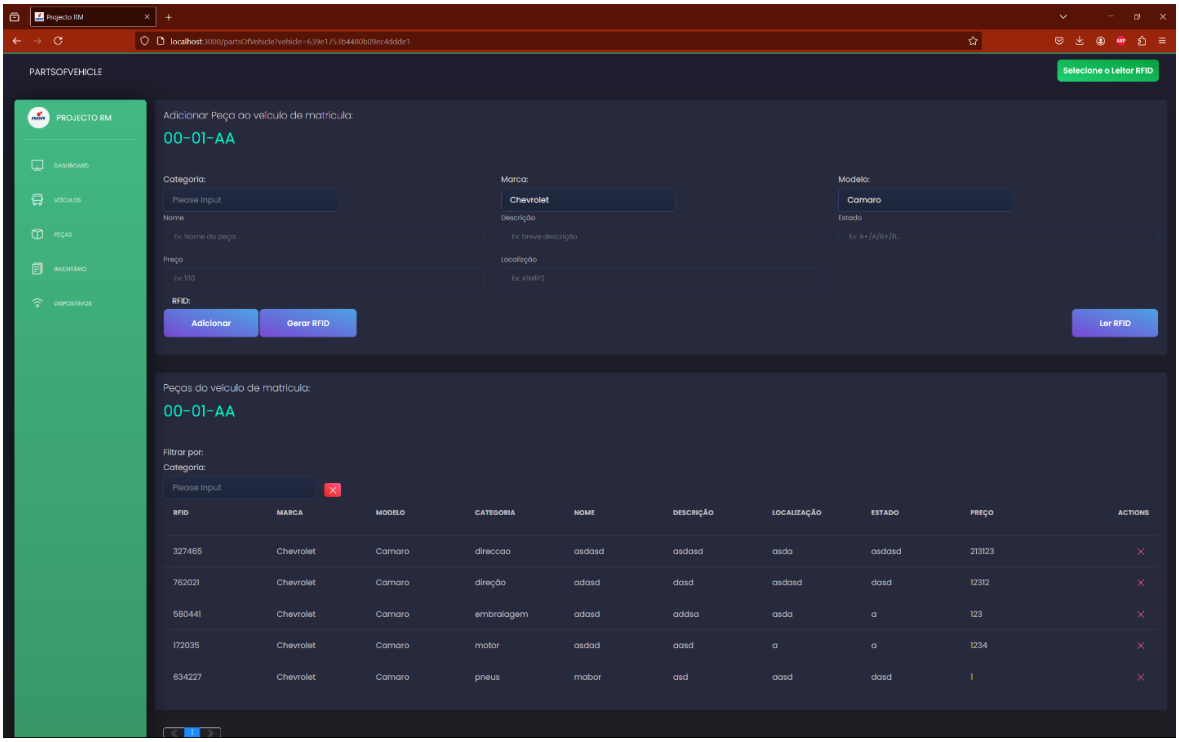

Figure 22 - Web app vehicles page

If the user wants to add an item without being assigned to a specific vehicle it can do so by navigating to the tab named “peças”.

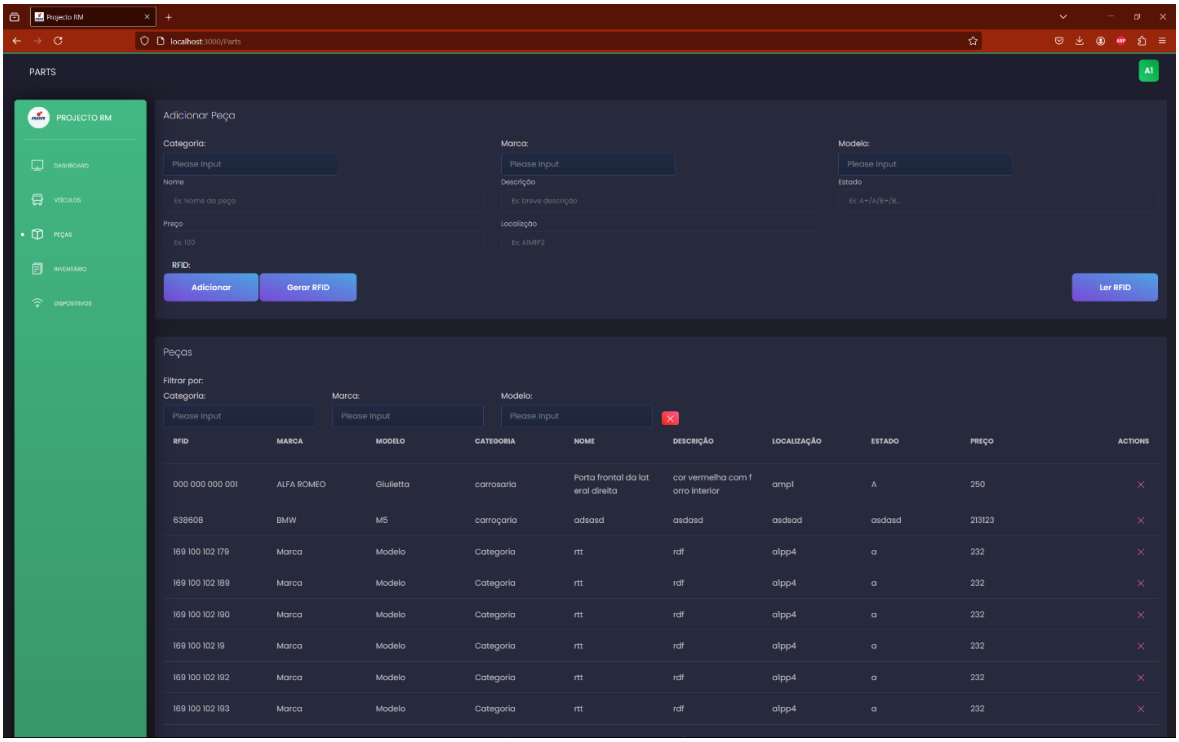

Figure 23 - Web app Car parts page

This page (figure 23) is identical to the page described before with the difference being the ability to select “marca” and “modelo” fields since its is not associated with any particular vehicle. In this page the user can also filter the vehicle parts present in the database by “categoria”, “marca” and “modelo” for the same reason and the parts table will have all the parts present in the database.

3.8.5 Inventory verification

To check the presence of the parts attributed to a location according to database records, the page “Inventário” (figure 24) should be used. In this page the user can select a location and a table with the parts associated with it will be shown with a greyed-out check mark. The following image shows the inventory checking page without being read, where all parts are seen with gray checkmarks:

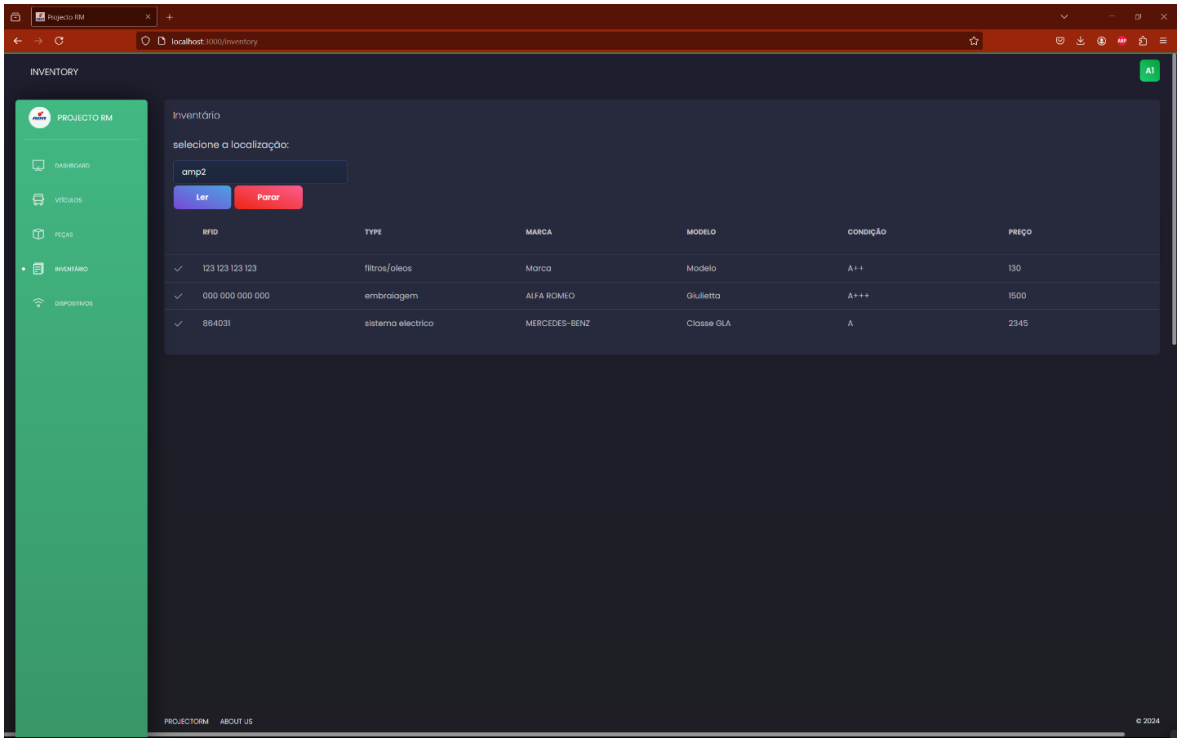

Figure 24 - Web app Inventory Verification page

The user standing near the location to be checked and having previously selected the desired RFID handheld reader on the webapp, can then click the button “Ler” and start scanning the location with the reader. As the reader reads the tag IDs and send them to the webapp the table updates the corresponding parts check marks to a green color. The user can request the device to stop scanning (reading) by clicking the “Parar” button. The following image (figure 25) illustrates the successful reading of some of the tags corresponding to some parts:

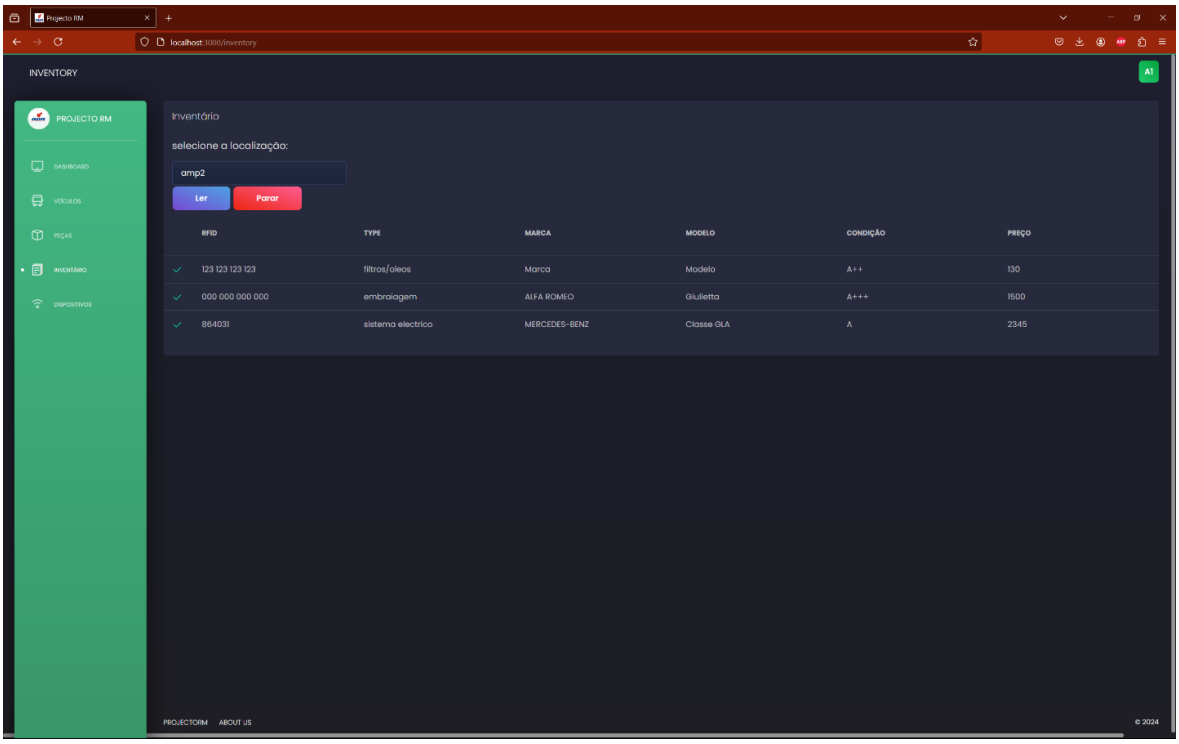

Figure 25 - Web app Inventory Successful Verification page

3.9 RFID handeld reader application: Configuration and network registration

When the device starts, it will try to connect to the last connected Wi-Fi AP. If it is unsuccessful, it will then restart in Wi-Fi AP mode with the SSID name of “ESP-WIFI-MANAGER” and self-host a configuration webpage (figure 27) at IP address “192.168.4.1”. The user using a device(smartphone, computer,…) with a browser connects to the AP created by the RFID reader and accesses the website by entering the IP address in the browser and through this configuration webpage it can enter the SSID and password of the desired Wi-Fi AP for the Wi-Fi connection, the MQTT server IP Address, the API endpoint and the password needed for the device for the single use credentials given by the API. The RFID handheld reader then restarts and tries to connect to the configured Wi-Fi network and MQTT broker. After successfully connecting with the Wi-Fi network and with the MQTT broker it will then be ready to accept commands from the webapp and will start sending its information such as Wi-Fi RSSI to be stored on the database. The following flow chart figure (26) illustrates the firmware logic.

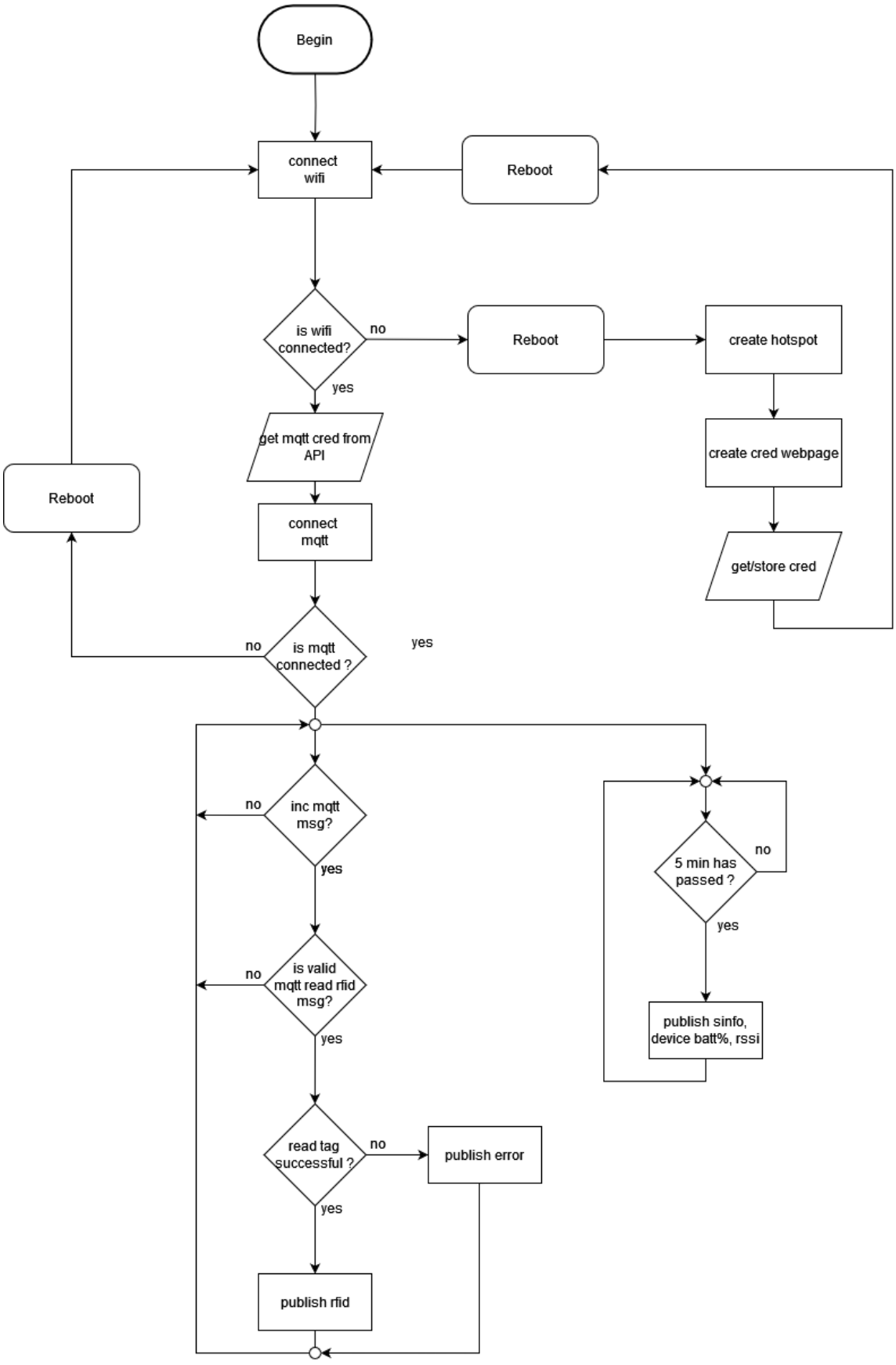

Figure 26 - ESP32 flow chart

The following figure (figure 24) shows the device configuration webpage:

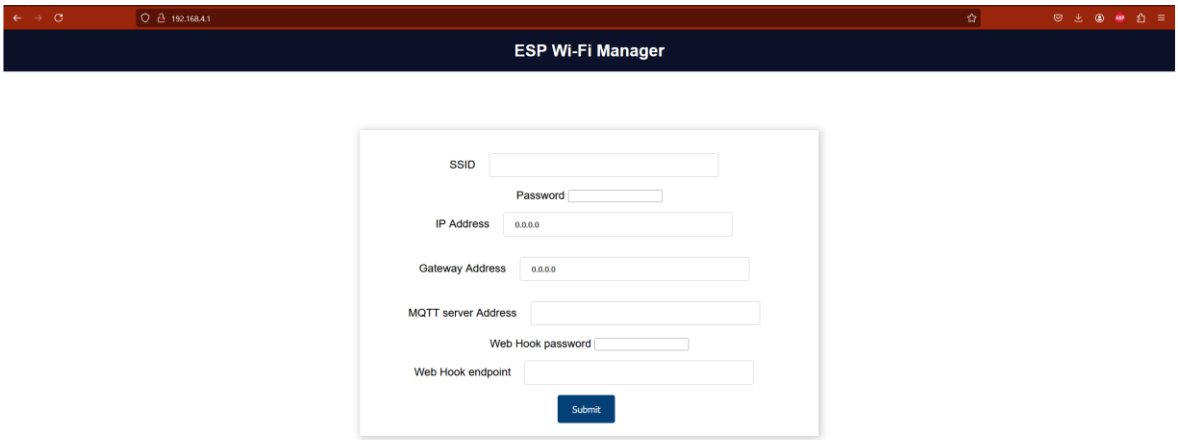

Figure 27 - ESP32 Credentials manager

4. Conclusion and future work

The present work was made as proof of concept, and as such some aspects were not taken into consideration. From a security standpoint, in case of a production environment, both the web application website and MQTT should use a secure version implementing the SSL/TLS protocol which should render the one-time credentials for the MQTT connections unnecessary since these were merely implemented to mitigate a possible man-in-the-middle attack. The versions of software and frameworks used should also be kept up to date. Regarding the database configuration, this should be configured as a sharded cluster with each replica hosted on different servers to be able to leverage the benefits of horizontal scaling, high reliability and so on. Regarding the device itself, it could feature a physical button to start a reading, and display some information in the OLED screen such battery information, and some information about the eventual item.

The developed system does allow an end-of-life vehicle company to keep a record of the vehicle and vehicle parts information and for this information to be accessed remotely. The tracking of inventoried parts still needs to be made manually and even though inventorying and stock placement confirmation can be made at a distance, it falls short of desired distance in order to be made remotely. The UHF RFID module used, even though it was meant to be used in relative proximity, for the allowed frequency in Europe, has an even further reduction in reading range probably because the small antenna it comes with has a narrow bandwidth and is tuned for another one of the other possible configurable frequencies.

As possible future work, it would be interesting to complement the system with a reader capable of reading the vehicle part attached tags from further away. To accomplish this, one might consider either using a higher gain antenna tunned for the frequency used in Europe or an RFID reader module with a higher gain or a combination of both. It would also be interesting to use a different wireless communication link such as LoRa or Wi-Fi low power protocol specific for IoT devices such as Wi-Fi 802.11af/ah/ba.

5. Bibliography

[1] [Online]. Available: http://data.europa.eu/eli/dir/2000/53/oj. [Accessed 21 4 2024].

[2] [Online]. Available: https://www.valorcar.pt/en/vfv/antigos-indicadores. [Accessed 21 4 2024].

[3] A. Lozano-Nieto, RFID Design Fundamentals and Applications, Boca Raton, FL: CRC Press, 2011.

[4] D. E. Brown, RFID implementation, New York: McGraw-Hill, 2007.

[5] [Online]. Available: https://www.researchgate.net/publication/290628144_Smart_tagging_technologies_in_pervasive_learning_environments2024. [Accessed 21 4 2024].

[6] [Online]. Available: https://www.rfidtagworld.com/Application/Smart-Warehousing/Smart-Warehousing_893.html. [Accessed 23 4 2024].

[7] [Online]. Available: https://www.geeksforgeeks.org/difference-between-relational-database-and-nosql/. [Accessed 23 4 2024].

[8] [Online]. Available: https://www.mongodb.com/resources/basics/databases/acid-transactions. [Accessed 23 4 2024].

[9] [Online]. Available: https://www.ibm.com/topics/api. [Accessed 1 5 2024].

[10] [Online]. Available: https://aws.amazon.com/what-is/api/. [Accessed 1 5 2024].

[11] [Online]. Available: https://nodejs.org/en. [Accessed 1 5 2024].

[12] [Online]. Available: https://mqtt.org/ . [Accessed 1 5 2024].

[13] [Online]. Available: https://www.emqx.com/en/blog/the-easiest-guide-to-getting-started-with-mqtt. [Accessed 1 5 2024].

[14] "Analog devices," [Online]. Available: https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html. [Accessed 20 09 2024].

[15] [Online]. Available: https://github.com/emqx/emqx?tab=readme-ov-file. [Accessed 5 5 2024].

[16] [Online]. Available: https://www.emqx.com/en. [Accessed 5 5 2024].

[17] [Online]. Available: https://en.wikipedia.org/wiki/Docker_(software). [Accessed 7 5 2024].

[18] [Online]. Available: https://www.docker.com/resources/what-container/. [Accessed 7 5 2024].

[19] [Online]. Available: https://en.wikipedia.org/wiki/Visual_Studio_Code. [Accessed 7 5 2024].

[20] [Online]. Available: https://www.mongodb.com/products/tools/compass. [Accessed 7 5 2024].

[21] [Online]. Available: https://github.com/nuxt/nuxt. [Accessed 13 5 2024].

[22] [Online]. Available: https://nuxt.com/. [Accessed 13 5 2024].

[23] [Online]. Available: https://www.creative-tim.com/product/nuxt-black-dashboard. [Accessed 13 5 2024].

ANEXO A

API endpoints documentation

/carmaker

Description

This is an api to fetch, add and delete the carmaker collections in the database

Base URL

The base URL for all API requests is:

`http://localhost:3001/api/`

Authentication

All requests are only processed if valid token is provided. Token is provided through headers using key-pair "token:$validatedUserToken"

Endpoints

GET CARMAKERS

`GET /carmakers`

Returns a list of all vehicle brands in the database.

Parameters

Response

Returns a JSON object with the following properties:

- `status`: Whether request was successful or not.
- `data`: An array of car makers objects, each with the following properties:
  - `id`: The unique identifier of the car maker.
  - `name`: The name of the vehicle brand.

Example

/carmaker 1

Request:

```
GET /carmakers
```

Response:

```
{
  "status": "success",
  "data": [
    {
      "_id": "63eea353e789221a0b33a468",
      "name": "ABARTH"
    },
    {
      "_id": "63eea353e789221a0b33a469",
      "name": "AIWAYS"
    },
    {
      "_id": "63eea353e789221a0b33a46a",
      "name": "ALFA ROMEO"
    },
    {
      "_id": "63eea353e789221a0b33a46b",
      "name": "ALPINA"
    },
    {
      "_id": "63eea353e789221a0b33a46c",
      "name": "ALPINE"
    },
  ...
  ]
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.

- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

GET MAKERMODELS

`GET /makermodels`

Returns a list of all models for a specified manufacturer in the database

Parameters

- `name`: The name of the vehicle manufacturer for which will be returned the maker models

Response

Returns a JSON object with the following properties:

- `status`: Success, error,
- `data`: An array of the specified car maker models.

Example

Request:

```
GET /makermodels?name=AIWAYS
```

Response:

```
{
  "status": "success",
  "data": [
    "Família",
    "U5"
```

/carmaker 3

```
  ]
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

ADD CAR MAKER

`POST /carmakers`

Adds a car maker information to the database.

Parameters

Response

Returns a JSON object with the following properties:

- `status`: Success, error.

Example

Request:

```
{
  "newCarMaker":{
    "name":"testing2",
    "models":[
      "testing1",
      "testing2"]
```

```
  }
}
```

Response:

```
{
  "status": "success",
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

DELETE CAR MAKER

`DELETE /carmakers`

Deletes a manufacturer and its associated models (carmaker) information from the database.

Parameters

`id`: id of the carmaker to be removed from DB.

Response

Returns a JSON object with the following properties:

- `status`: Success, error.
- `data`: An object, with the following properties:
  - `n`: Number of deleted entries.

- `ok`: A boolean `acknowledged` as `true` if the operation ran with write concern or `false` if write concern was disabled
- `deletedCount`: number of deleted copies.

Example

Request:

```
DELETE /carmakers?id=64d4cf164eb77a373c5cd5e4
```

Response:

```
{
  "status": "success",
  "data": {
    "n": 1,
    "ok": 1,
    "deletedCount": 1
  }
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

/carpart

Description

This is an api to fetch, add, delete, and update the car parts collection db.

Base URL

The base URL for all API requests is:

`http://localhost.com:3001/api/`

Authentication

All requests are only processed if valid token is provided. Token is provided through headers using key-pair “token:$validatedUserToken”

Endpoints

GET CAR PARTS

`GET /carpart`

Returns a list of all car parts in the database if requested without parameters, and the matching car parts for the provided parameters.

Parameters

- `id` (optional): The unique identifier of the book.
- `userId` (optional): The ID of the user who added the part to the database.
- `vehicleId` (optional): The vehicleID from which the part was taken.
- `type` (optional): The category/type of the part.
- `name` (optional):The name of the part.
- `description` (optional): Part description.
- `state` (optional) : The quality state of the part
- `price` (optional): The part price.

- `rfid` (optional): The RFID tag number associated with the part.
- `brand` (optional): The vehicle manufacturer name of the vehicle from which the part was taken.
- `model` (optional): The vehicle model name from which the part was taken.
- `whlocation` (optional): The warehouse location of the part.
- `createdTime` (optional): The time which the car part was first created

Response

Returns a JSON object with the following properties:

- `status`: Success, error.
- `data`: An array of car part objects, each with the following properties:
  - `id`: The unique identifier of the car part document.
  - `userId`: The ID of the user who added the part to the database.
  - `vehicleId` (optional): The vehicleID from which the part was taken.
  - `type`: The category/type of the part.
  - `name` The name of the part.
  - `description`: Part description.
  - `state` : The quality state of the part
  - `price`: The part price.
  - `rfid`: The RFID tag number associated with the part.
  - `carMaker`: The vehicle manufacturer name of the vehicle from which the part was taken.
  - `carModel`: The vehicle model name from which the part was taken.
  - `whlocation`: The warehouse location of the part.
  - `createdTime`: The time which the car part was first created
  - `__v`: The version of the DB document.

Example

Request:

```
GET /carpart
```

Response:


```
{
  "status": "success",
  "data": [
    {
      "_id": "636abdbcd6025685c873ee74",
      "userId": "63606653ae9bf054108523c3",
      "vehicleId": "",
      "type": "Categoria",
      "name": "rtt",
      "description": "rdf",
      "state": "a",
      "price": 232,
      "rfid": " 169 100 102 179",
      "carMaker": "Marca",
      "carModel": "Modelo",
      "whlocation": "a1pp4",
      "createdTime": 1667939772941,
      "__v": 0
    }
  ]
}
```


Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

GET LOCATIONS

/carpart 3

`GET /location`

Returns a list of all locations in the database.

Parameters

None.

Response

Returns a JSON object with the following properties:

- `status`: The total number of books in the library.
- `data`: An array of with the locations present in the database.

Example

Request:

```
GET /location
```

Response:

```
{
  "status": "success",
  "data": [
    "A1pp4",
    "a",
    "a1p1",
    "a1pp3",
    "a1pp4"
  ]
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.

- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

ADD CAR PARTS

`POST /carpart`

Adds a car part to the database.

Parameters

- `vehicleId` (optional): The vehicleID from which the part was taken.
- `type`: The category/type of the part.
- `name` The name of the part.
- `description`: Part description.
- `state` : The quality state of the part
- `price`: The part price.
- `rfid`: The RFID tag number associated with the part.
- `carMaker`: The vehicle manufacturer name of the vehicle from which the part was taken.
- `carModel`: The vehicle model name from which the part was taken.
- `whlocation`: The warehouse location of the part.

Response

Returns a JSON object with the following properties:

- `status`: Success, error.

Example

Request:

```
{
  "newCarPart":
    {
      "vehicleId": "",
      "type": "Motor",
      "name": "Motor 1.6 gasoleo",
      "description": "Motor 1.6 gasoleo com 128965km",
      "state": "a",
      "price": 2320,
      "rfid": " 169 100 102 189",
      "carMaker": "Honda",
      "carModel": "Civic",
      "whlocation": "a1pp4"
    }
}
```

Response:

```
{
  "status": "success"
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

DELETES CAR PARTS

`DELETE /carpart`

/carpart 6

Deletes(sets deleted flag to true) a car part from the database.

Parameters

- `rfid` : The RFID of the car part to be removed from the database

Response

Returns a JSON object with the following properties:

- `status`: Success, error.
- `data`: An object, with the following properties:
  - `n`: Number of entries.
  - `nModified`: Number of deleted copies.
  - `ok`: A boolean `acknowledged` as `true` if the operation ran with write concern or `false` if write concern was disabled

Example

Request:

```
DELETE /carpart?rfid= 169 100 102 189
```

Response:

```
{
  "status": "success",
  "data": {
    "n": 1,
    "nModified":1
    "ok": 1,
  }
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.

- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

/category

Description

This is an api to fetch and add car part categories information

Base URL

The base URL for all API requests is:

`http://localhost:3001/api/`

Authentication

All requests are only processed if valid token is provided. Token is provided through headers using key-pair “token:$validatedUserToken”

Endpoints

GET CATEGORIES

`GET /category`

Returns a list of all car parts categories.

Parameters

None.

Response

Returns a JSON object with the following properties:

- `status`: Success, error.
- `data`: An array of car part objects, each with the following properties:
  - `_id`: The unique identifier of the category.
  - `name` The name of the category.
  - `__v`: The version of the DB document.

Example

Request:

```
GET /category
```

Response:

```
{
  "status": "succes",
  "data": [
    {
      "_id": "64d4fbf74eb77a373c5cd5fb",
      "name": "suspencao",
      "__v": 0
    },
    {
      "_id": "64d4fc414eb77a373c5cd5fe",
      "name": "direccao",
      "__v": 0
    },
    {
      "_id": "64d4fc594eb77a373c5cd601",
      "name": "Motor",
      "__v": 0
    },
...
  ]
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.

- `500 Internal Server Error`: An unexpected error occurred on the server.

ADD CAR PARTS CATEGORY

`POST /category`

Adds a car part category to the database.

Parameters

- `name` The name of the category to be added to the database.

None, request from body.

Response

Returns a JSON object with the following properties:

- `status`: Success, error.

Example

Request:

```
{
  "newCategory":{
  "name":"suspencao"
  }
}
```

Response:

```
{
  "status": "success"
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

GET CAR PARTS CATEGORY COUNT

`GET /categorycount`

Gets the quantity of a car parts of the provided category from the database.

Parameters

- `type`: The name of the car parts category for which its count will be returned

Response

Returns a JSON object with the following properties:

- `status`: Success, error.

Example

Request:

```
GET /categorycount?type=motor
```

Response:

```
{
  "status": "success",
  "data": 8
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

/category 5

/device

Description

This is an api to get, add, delete and alter devices state

Base URL

The base URL for all API requests is:

`localhost:3001/api/`

Authentication

All requests are only processed if valid token is provided. Token is provided through headers using key-pair “token:$validatedUserToken”

Endpoints

GET

`GET /device`

Returns a list of all devices in the database.

Parameters

- None.

Response

Returns a JSON object with the following properties:

- `status`: request status(”success”, “fail”)
- `data`: An array of device objects, each with the following properties:
  - `chargeLeft`: Last reported device charge left
  - `_id`: The unique identifier of the device assigned by the database.
  - `rssi`: The last reported wifi signal strength of the device
  - `name`: The devices name.

- `did`: Device id.
- `userId`: The id of the user who created the device.
- `createdTime`: Date of device creation.
- `password`: Password assigned to the device on creation as to obtain device connection credentials
- `lastUpdatedTime`: Last time device was edited.

Example

Request:

```
GET /device
```

Response:

```
{
  "status": "success",
  "data": [
    {
      "selected": false,
      "chargeLeft": 0,
      "rssi": -54,
      "_id": "63606e583a849035f81fb676",
      "name": "A1",
      "dId": "1",
      "userId": "63606653ae9bf054108523c3",
      "createdTime": 1667264088078,
      "password": "eZS4Izvhpl",
      "__v": 0,
      "lastUpdatedTime": 1682092864254
    },
    {
      "selected": false,
      "chargeLeft": 100,
      "rssi": -140,
      "_id": "64a82904f51f47415cc8ea4a",
      "name": "Leitor Armazem 1",
      "dId": "2",
```

```
      "userId": "63606653ae9bf054108523c3",
      "createdTime": 1688742148121,
      "password": "e0O4CkirA9",
      "__v": 0
    }
  ]
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

POST

`POST /device`

Creates a device in the database.

Parameters

- `name` : The name of the device being created.
- `dId`: The ID/SerialN of the device to be created and added to DB.

Response

Returns a JSON object with the following properties:

- `status`: request status(”success”, “error”)

Example

- `400 Bad Request`: The request was malformed or missing required parameters.
- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

GET CAR PARTS CATEGORY COUNT

`GET /categorycount`

Gets the quantity of a car parts of the provided category from the database.

Parameters

- `type`: The name of the car parts category for which its count will be returned

Response

Returns a JSON object with the following properties:

- `status`: Success, error.

Example

Request:

```
GET /categorycount?type=motor
```

Response:

```
{
  "status": "success",
  "data": 8
}
```

Errors

This API uses the following error codes:

Returns a JSON object with the following properties:

- `status`: request status(”success”, “error”)
- `data`: An object, with the following properties:
  - `n`: Number of deleted entries.
  - `ok`: A boolean `acknowledged` as `true` if the operation ran with write concern or `false` if write concern was disabled
  - `deletedCount`: Number of deleted copies.

Example

Request:

```
DELETE /carpart?dId=123
```

Response successful:

```
{
  "status": "success",
  "data": {
    "n": 0,
    "ok": 1,
    "deletedCount": 0
  }
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

PUT

`PUT /device`

Updates a device selected state in the database.

Parameters

- `dId`: The ID/SerialN of the device to be selected and updated to DB.

Response

Returns a JSON object with the following properties:

- `status`: request status(”success”, “error”)

Example

Request:

```
{
"dId":"123"
}
```

Response successful:

```
{
  "status": "success"
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

/webhook

Description

This is an api for the emqx broker and device credentials

Base URL

The base URL for all API requests is:

`http://localhost:3001/api/`

Authentication

All requests are only processed if valid token is provided. Token is provided through headers using key-pair “token:$webhookToken”

Endpoints

POST saver-webhook

`POST /saver-webhook`

Updates device information coming from the device through EMQX to the API

Parameters

- `rssi` : The Wi-Fi RSSI from device.
- `bat` : The battery level information of the device in %.

Response

None.

Example

Request:

```
{
"payload":{
    "rssi":"-80",
    "bat": "75",
```

```
    }
}
```

Response:

```
{

}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

POST getdevicecredentials

`POST /getdevicecredentials`

Updates device information coming from the device through EMQX to the API

Parameters

- `dId` : The device dId.
- `password`: The device password for obtaining one time credentials.

Response

- `username` : The device username to connect to EMQX broker.
- `password`: The device password for connecting to EMQX broker.

Example

Request:

```
{
    "dId":"-80",
    "password": "75",
}
```

Response:

```
{
  "username":"raAfRfajYr",
    "password": "ZflpqoeRWv",
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

/users

Description

This is an api to register users, login, and obtain mqtt credentials

Base URL

The base URL for all API requests is:

`http://localhost:3001/api/`

Endpoints

Register

`POST /register`

Adds an account to the database

Parameters

- `name`: Name of the account being created
- `email` :Email associated with the account used for login
- `password`: Account password

Response

Returns a JSON object with the following properties:

- `status`: Success,error

Example

Request:

```
{
  "nami":"test",
  "email":"t@t.com",
  "password":"a"
}
```

Response:

```
{
  "status": "success"
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

Login

`POST /login`

Login request, returns JWT token and the userdata associated with the logged account.

Parameters

- `email` :Email associated with the account used for login
- `password`: Account password

Response

Returns a JSON object with the following properties:

- `status`: Success,error
- `token`: JWT webtoken
- `userData`: an object with the following proprierties
  - `_id`: unique ID from db
  - `name:` Account name

- `email:` Logged in email information
- `__v` : db document version

Example

Request:

```
{
  "email":"t@t.com",
  "password":"a"
}
```

Response:

```
{
  "status": "success",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyRG
F0YSI6eyJfaWQiOiI2NGQyNWQwZTAzYjYxMTQ3NWM1YjIzMGMiLCJuYW1lI
joidGVzdCIsImVtYWlsIjoidEB0LmNvbSIsIl9fdiI6MH0sImlhdCI6MTY5
MTUwODEyNywiZXhwIjoxNjk0MTAwMTI3fQ.LQBNgMDT9KY0E1e4Er5dPQJk
HiaAAlLl58trg_-qq_A",
  "userData": {
    "_id": "64d25d0e03b611475c5b230c",
    "name": "test",
    "email": "t@t.com",
    "__v": 0
  }
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

GET MQTT CREDENTIALS

`POST /getmqttcredentials`

MQTT connection credentials request, returns One time username and one time password to connect to EMQX broker

Response

Returns a JSON object with the following properties:

- `status`: Success,error
- `username`: one time Username for mqtt connection
- `password`: one time Password for mqtt connection

Example

Request:

```
{

}
```

Response:

```
{
  "status": "success",
  "username": "hXMVvdfJzI",
  "password": "kMrsIwsoSn"
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `401 Unauthorized`: The API key provided was invalid or missing.
- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.

GET MQTT CREDENTIALS FOR RECONNECTION

`POST /getmqttcredentialsforreconnection`

MQTT connection credentials request, returns One time username and one time password to connect to EMQX broker, used to reconnect after loosing conection

Response

Returns a JSON object with the following properties:

- `status`: Success,error
- `username`: one time Username for mqtt connection
- `password`: one time Password for mqtt connection

Example

Request:

```
{

}
```

Response:

```
{
  "status": "success",
  "username": "hXMVvdfJzI",
  "password": "kMrsIwsoSn"
}
```

Errors

This API uses the following error codes:

- `400 Bad Request`: The request was malformed or missing required parameters.
- `401 Unauthorized`: The API key provided was invalid or missing.

- `404 Not Found`: The requested resource was not found.
- `500 Internal Server Error`: An unexpected error occurred on the server.