Engenharia e Gestão de Serviços

Universidade de Aveiro

Catarina Barroqueiro, Gonçalo Silva, Daniel Silva, Nuno Sousa

Engenharia e Gestão de Serviços

Dept. de Eletrónica, Telecomunicações e Informática

Universidade de Aveiro

(103895),(103244), (102537), (103664)

June 5, 2024

Contents

List of Figures

Glossário

DGEG Direção-Geral de Energia e Geologia.

Chapter 1

Fuellink Project

1.1 Overview

1.1.1 Introduction

The Fuellink Project emerges as a response to the latent need of companies to optimize the management of refueling their automotive fleets. Through the automation of data registration and the offer of analytical tools, Fuellink guarantees greater reliability, control, and efficiency in this process, providing benefits such as:

- **Reduced cost:** Elimination of manual errors and optimization of fuel consumption.
- **Increased productivity:** Streamlining the refueling process and freeing up time for strategic activities.
- **Improved decision-making:** Access to accurate and real-time data to support strategic decisions.
- **Greater control and security:** Monitoring the individual consumption of each vehicle and identifying possible frauds.

1.1.2 Problem

The **manual registration** of refueling data for the automotive fleet is a common practice in many companies. However, this method presents several flaws that can **compromise the reliability of the information** and generate a number of problems, such as:

- **Difficulty reading handwriting:** Manual notes can be illegible, making it difficult to consult and analyze data.

- **Discrepancies between the liters recorded and the real values of the pump:** Typing or annotation errors can generate distortions in consumption data, affecting decision-making.

- **Difficulty in making decisions about stock and the need to buy fuel:** The lack of accurate and reliable data makes it difficult to manage fuel stock and make strategic purchasing decisions.

1.1.3 Solution

The Fuellink Project proposes a complete system for automating the registration of refueling data and analyzing fuel consumption, which aims to solve the problems mentioned above and optimize the management of the automotive fleet. The system offers the following functionalities:

- Automatic registration of the details of each refueling: Eliminates the need for manual registration, ensuring the reliability and accuracy of the information.

- License plate reading: Automatically identifies the vehicle being refueled, associating consumption data with the respective vehicle.

- Employee authentication: Requires the identification of the employee who is refueling, ensuring greater control and security.

- Complete dashboard for the administrator: Offers an overview of all refueling data in real time, allowing the analysis of the individual consumption of each vehicle, identification of consumption patterns, market trends and optimization of fuel stock.

1.2 Development

The development of this project was divided between modules (dedicated to one member each), a Github organization (`https://github.com/Fuel-Link`) was created and a repository was created for each module. A list with the repositories and their addresses are:

1. `https://github.com/Fuel-Link/deployment`: Contains all the deployment files

2. `https://github.com/Fuel-Link/auth`: Contains all the files associated to the Authentication module

3. `https://github.com/Fuel-Link/composer`: Contains all files associated to the composer

4. `https://github.com/Fuel-Link/orchestrator`: Contains all kakfa files and some components of the Composer, which interact with kafka

5. `https://github.com/Fuel-Link/market-analysis`: Contains all the components for the Market Analysis module
6. `https://github.com/Fuel-Link/stock-management`: Contains all the components for the Stock Management module
7. `https://github.com/Fuel-Link/gas-pump`: Contains all the components for the Gas Pump Module
8. `https://github.com/Fuel-Link/license_plate_recognition`: Contains all the components for the License Plate Recognition module, with the exception of the kafka stream, located in the orchestrator

1.3 Modules

1.3.1 Authentication

The authentication module uses a Keycloak deployment to provide an OpenID Connect API that allows other modules to perform authentication and authorization. The Keycloak service runs on a Docker container, using the default official Keycloak image. It uses a PostgreSQL database to store all the configurations and user data. Both the Keycloak and PostgreSQL were initially deployed locally using Docker Compose, with the following file describing the two components.

```yaml
name: auth
services:
  keycloak:
    image: 'keycloak/keycloak:24.0'
    command: start-dev
    depends_on:
      postgres:
        condition: service_healthy
    ports:
      - "8080:8080"
    environment:
      - KEYCLOAK_ADMIN=admin
      - KEYCLOAK_ADMIN_PASSWORD=admin
      - KC_DB=postgres
      - KC_DB_URL_HOST=postgres
      - KC_DB_URL_DATABASE=keycloak
      - KC_DB_SCHEMA=public
      - KC_DB_USERNAME=admin
      - KC_DB_PASSWORD=admin
  postgres:
    image: 'postgres:16-alpine'
    healthcheck:
      test: "exit 0"
    ports:
      - "5432:5432"
    volumes:
      - ./volumes/postgres:/var/lib/postgresql/data
    environment:
```

```
      - POSTGRES_PASSWORD=admin
      - POSTGRES_USER=admin
      - POSTGRES_DB=keycloak
```

Even though the passwords are in plain text in the Docker Compose file, this was not a concern for us when running the services locally. Measures were taken to protect these credentials when deploying to the Kubernetes cluster.

The service needed a lot of configurations that are not presentable in a file, since they were done in Keycloak's web interface. One of these configurations was the inclusion alternative methods, such as a one-time password and a GitHub account. Neither of these is necessary, but is an option available to the users.

To deploy these services in Kubernetes, a new Docker image was used for the Keycloak service. It uses the following Dockerfile. created with the following Dockerfile.

```
FROM keycloak/keycloak:24.0 as builder

# Enable health and metrics support
ENV KC_HEALTH_ENABLED=true
ENV KC_METRICS_ENABLED=true

# Configure a database vendor
ENV KC_DB=postgres

WORKDIR /opt/keycloak
RUN /opt/keycloak/bin/kc.sh build

FROM keycloak/keycloak:24.0
COPY --from=builder /opt/keycloak/ /opt/keycloak/
ENTRYPOINT ["/opt/keycloak/bin/kc.sh"]
```

This Dockerfile uses the official Keycloak image as a base for a building stage. It then enables the health endpoints for easier testing of running status. The vendor of the database used is also defined in this Dockerfile, since it makes the access faster at runtime. Then, we compile the actual Keycloak software. Compiling it in the creation of the Docker image mas it significantly faster to boot up the image in a container. We then use a new Keycloak image as base and copy the compiled output from the building stage, and set the startup script as the entrypoint.

The files used to deploy the services on the Kubernetes cluster are not included in this report, but they can be found on our GitHub repository. To summarize, secrets are used to access and store their credentials, which is more secure. The Keycloak deployment uses the ndots=1 setting to allow it to communicate with the GitHub authentication service, but this prevents us from accessing the admin dashboard from the browser. Essentialy, GitHub authentication and access to the admin panel are mutually exclusive features of our project, and we were not able to understand why.

There is also a service to perform authentication on the pump. This is a simple webpage that sends a post request to a backend express API, which then performs authentication on Keycloak and, if successful, requests the username and ID and sends them to the Kafka broker.

It uses the following code for the backend API.

```
    const config = require('./config');
const key = require("./keycloak.json");

const express = require('express');
const app = express();
const path = require('path');
const bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: false }));

const { Kafka } = require('kafkajs');
const kafka = new Kafka({
  clientId: 'pump',
  brokers: ['kafka:29092'],
});

app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, '/index.html'));
})

app.get('/logo.svg', (req, res) => {
  res.sendFile(path.join(__dirname, '/logo.svg'));
})

app.post('/grantAuthorization', async (req, res) => {

  let myHeaders, urlencoded, response, requestOptions;

  //
  myHeaders = new Headers();
  myHeaders.append("Content-Type", "application/x-www-form-
    urlencoded");

  urlencoded = new URLSearchParams();
  urlencoded.append("grant_type", "password");
  urlencoded.append("client_id", key.resource);
  urlencoded.append("username", req.body.username);
  urlencoded.append("password", req.body.password);
  urlencoded.append("scope", "openid profile email roles");

  requestOptions = {
    method: "POST",
    headers: myHeaders,
    body: urlencoded,
    redirect: "follow"
  };

  try {
    response = await fetch(key['auth-server-url'] + "/realms/Fuel-
    Link/protocol/openid-connect/token", requestOptions)
    result = await response.json();
  }
  catch {
    res.json('login failed');
```

```
    return
  }

  //
  myHeaders = new Headers();
  myHeaders.append("Authorization", "Bearer " + result.access_token);

  urlencoded = new URLSearchParams();
  urlencoded.append("client_id", key.resource);
  urlencoded.append("token", req.body.token);

  requestOptions = {
    method: "POST",
    headers: myHeaders,
    body: urlencoded,
    redirect: "follow"
  };

  try {
    response = await fetch(key['auth-server-url'] + "/realms/Fuel-Link/protocol/openid-connect/userinfo", requestOptions);
    result = await response.json();
  }
  catch {
    res.json('user list failed');
    return
  }

  const producer = kafka.producer();
  await producer.connect();
  await producer.send(
    {
      topic: 'gas-pump_auth',
      messages: [
        {
          key: 'data',
          value: '{"username": "'+result.preferred_username+'", "hash": "'+result.sub+'"}',
        }
      ],
    }
  );
  await producer.disconnect();

  console.log(result.sub)
  console.log(result.preferred_username)
  res.json('success');
});

app.listen(3000, function () {
  console.log('App listening on port 3000');
});
```

There is also a simple service that only performs one function: getting a list of all users, and their associated IDs, roles and some other data. The code for this service can be found on our GitHub in the "auth" repository, in the folder "getusers".

All of the Kubernetes deployment files can be found on the "deployment" repository, in the "auth" folder.

1.3.2 Analysis & Management

The Market Analysis and Stock Management modules implements the necessary mechanisms to increase the efficiency of fuel purchases. These modules were developed by Nuno Sousa.

To deploy these services Kompose-convert was used to generate the deployment scripts from the docker-compose plus some changes, like the correct namespace, and volume mount for the dbs.

1.3.2.1 Market Analysis

The market analysis module aims to accurately predict fuel prices in the near future in order to allow clients to make more efficient purchases. The goals for this module were to:

- Allow multiple users, each with its own data stream,
- Update that data stream from a larger database,
- Accurately predict future prices.

In order to use this service, clients must first register themselves via the /addClient endpoint, providing the necessary information to connect to their influxdb, this will register the client and return an authentication token that must be used whenever they wish to use the service.

```
url = 'http://grupo1-egs-deti.ua.pt/market-analysis/addClient'
payload = {
    'org': 'OrgName',
    'url': 'http://influxdb:8086',
    'bucket': 'bucket',
    'measurement': 'measurement',
    'field': 'field'
}
headers = {
    'Content-Type': 'application/json'
}
response = requests.post(url, data=json.dumps(payload), headers=
    headers)
```

All of this information, except for the organization name, can later be changed via the /updateClient endpoint, providing the org name, auth token and whichever fields need updating.

```
url = 'http://grupo1-egs-deti.ua.pt/market-analysis/updateClient'
payload = {
    'org': 'OrgName',
    'authToken': '.....',
    #any other field can be updated
```

```
}
headers = {
    'Content-Type': 'application/json'
}
response = requests.put(url, data=json.dumps(payload), headers=
    headers)
```

When a client needs to update their database, they could use their own data, or use the /updateData endpoint that uses the DGEG api and will update their influxdb with the latest data.

```
url = 'http://grupo1-egs-deti.ua.pt/market-analysis/updateData'
payload = {
    'org': 'OrgName',
    'authToken': '.....',
    'token': '.....'   #influxdb token not the authtoken
}
headers = {
    'Content-Type': 'application/json'
}
response = requests.put(url, data=json.dumps(payload), headers=
    headers)
```

To predict the fuel prices the client calls the /predict endpoint, this will fetch all the data from their influx, feeds it into Prophet and returns values for the number of days that the client specified in both ways, the past prices and the predicted ones.

```
url = 'http://grupo1-egs-deti.ua.pt/market-analysis/predict'
payload = {
    'org': 'OrgName',
    'authToken': '.....',
    'token': '.....',   #influxdb token not the authtoken
    'days': '7'         #prefered number of days, default 15
}
headers = {
    'Content-Type': 'application/json'
}
response = requests.get(url, headers=headers)
```

1.3.2.2 Stock Management

The stock management module complements the market analysis one by using its predicted values and current fuel consumption to decide the optimal time to buy more fuel.

To track fuel usage when a fuel pump is used, /usePump should be called.

```
url = 'http://grupo1-egs-deti.ua.pt/stock/usePump'
payload = {
    'pump_id': 'Bomba 1',
    'amount': '10',
    'org': 'Org Name',
    'client': 'Pedro'      #pessoa que usou a bomba
}
headers = {
    'Content-Type': 'application/json'
```

```
}
response = requests.post(url, data=json.dumps(payload), headers=
    headers)
```

When more fuel is bought and a pump is restocked /restockFuel should be called.

```
url = 'http://grupo1-egs-deti.ua.pt/stock/restockFuel'
payload = {
    'pump_id': 'Bomba 1',
    'amount': 100,
    'org': 'Org Name'
}
headers = {
    'Content-Type': 'application/json'
}
response = requests.post(url, data=json.dumps(payload), headers=
    headers)
```

To determine whether or not to buy more fuel, /assessFuel is called with the values from the market analysis prediction. This will determine the current fuel amount, median consumption and compare that against the forecast to determine if it is more advantegeous to wait until prices drop, if it is better to buy before prices rise even further or if fuel will run out before better prices.

```
url = 'http://grupo1-egs-deti.ua.pt/stock/assessFuel'
payload = {
    'org': 'Org Name',
    'predictions': [
                 {"ds": "Tue, 21 May 2024 00:00:00 GMT", "yhat":
    1.76397585647508},
                 {"ds": "Wed, 22 May 2024 00:00:00 GMT", "yhat":
    1.76649328681331}
             ]
}
headers = {
    'Content-Type': 'application/json'
}
response = requests.post(url, data=json.dumps(payload), headers=
    headers)
```

This will return the current fuel consumption, total stock, and the suggested decision.

1.3.3 Gas Pump

The Gas Pump module is located in $gas-pump/gas-pump\_device$ folder and is responsible for interacting with the physical Gas Pump allowing it to be locked/unlocked, so that an employee can supply the company vehicle with fuel. After the supply is completed, the pump will signal the supply manager (included in the composer) of the amount supplied and stock left. An image of the final aspect of the pump can also be found below. This module was developed by Gonçalo Silva.

1.3.3.1 PlatformIO project

The Gas pump device project was developed using PlatformIO Visual Studio Code extension. The Arduino framework was chosen due it's extensible library and possibility of use with the ESP32. The project is structured with Header files in the *include* folder and source files in the *src* folder, as to decrease dependencies between packages. In the next sub-sections, we'll be approaching each module and it's purpose.

1.3.3.2 CommsHandler package

This package is responsible for declaring and handling the communication methods of the ESP32. It establishes connectivity to a WiFi AP (with credentials in a local $WiFiCredentials.h$ file) and waits for a NTP (Network Time Protocol) server, using the ArduinoJSON library, to reply with the current time, as to have accurate Timestamps locally. After the NTP response is received, it'll establish connection to the MQTT broker, using WebSockets, subscribe to it's downlink topic and bind the callback function for the subscribe topic/s. Subsequent access to this package operation is done mainly to publish data to the MQTT broker.

```
void CommsHandler::mqtt_message_callback(char* topic, byte* payload, unsigned int length){
    Serial.print("Message arrived in topic [");
    Serial.print(topic);
    Serial.println("]:");
    Serial.println(" - Size: " + String(length));
    Serial.print(" - Message: ");
    for(int i = 0; i < length; i++) {
        Serial.print((char)payload[i]);
    }
    Serial.println();

    // Allocate the JSON document
    JsonDocument doc;

    // Parse JSON object
    DeserializationError error = deserializeJson(doc, payload, length);
    if (error) {
        Serial.print(F("Error: deserializeJson() failed: "));
        Serial.println(error.f_str());
        return;
    }
}
```

Listing 1.1: MQTT subscrived topics callback function

1.3.3.3 DataTypes package

The DataTypes is a simple package to define Enumerators that are shared across multiple packages. The $MESSAGE\_TYPE$ defines the possible message types in communication with the MQTT broker, while the $FUEL_TYPE$

defines the fuel types that the Gas Pump supports. This definitions can be seen in below:

```
enum MESSAGE_TYPE {
    PUMP_INIT, //!< Sent by the pump to Ditto, for initialization
    SUPPLY_AUTHORIZED,  //!< Sent by Ditto to the pump, to
    authorize a supply
    SUPPLY_COMPLETED,   //!< Sent by the pump to Ditto, to confirm
    a supply
    FUEL_REPLENISHMENT, //!< Sent by pump to the Ditto, signalling
    that the fuel has been replenished
    SUPPLY_ERROR,    //!< Sent by the pump to Ditto, to report an
    error
    UNKNOWN //!< Unknown/Error message type
};

enum FUEL_TYPE {
    DIESEL,
    PETROL,
    LPG
};
```

Listing 1.2: DataTypes definition

1.3.3.4 Message package

The Message package handles the creation, Serialization and Handling of messages. It uses the ArduinoJSON library to handle the messages internally as JSON format, but to also serialize them as C-Strings when sending them to the MQTT broker

1.3.3.5 PumpInteraction package

This package is responsible for interacting with the gas pump directly, using GPIO pins. It can track the current fuel stock, the capacity of the pump and replenishment's to the fuel tank, increasing stock levels. The main function used is the $supply\_fuel$, which handles the supply of fuel to the vehicle, after receiving the unlocking order. The function start with turning on the LED pin, to signal the user that he can use the pump. Next, the user has 60 seconds to press the button, triggering the start of the pump, otherwise, the supply operation will be stopped, the pump will be locked and a $SUPPLY\_ERROR$ message will be sent to the broker. While supplying fuel, the $PUMP\_CONTROL\_RELAY\_PIN$ is at HIGH state, with the pump being activated and supplying fuel, otherwise it's at LOW state. This function can be seen below:

```
esp_err_t PumpInteraction::supply_fuel(double &suppliedAmount){
    if(stock < MIN_FUEL_SUPPLY_IN_LITERS)
        return ESP_FAIL;

    ESP_ERROR_CHECK(unlock_pump());

    unsigned long startTime = millis();
```

```
    bool buttonPressed = false;

    Serial.println(" - Waiting for user to press button");

    // Waits x seconds for the user to press the button
    while (millis() - startTime < PUMP_ACTIVATED_WAITING_TIME_IN_SEC * 1000) {
        if (digitalRead(PUMP_BUTTON_PIN) == HIGH) { // check if button is pressed
            buttonPressed = true; // set buttonPressed to true
            break;
        }
    }

    if(!buttonPressed){
        ESP_ERROR_CHECK(lock_pump());
        Serial.println(" - Button not pressed, aborting fuel supply");
        return ESP_FAIL;
    }

    Serial.println(" - Button pressed. Supplying fuel...");

    // Activate pump
    digitalWrite(PUMP_CONTROL_RELAY_PIN, HIGH);

    // Wait for the user to stop using the button
    while(digitalRead(PUMP_BUTTON_PIN) == HIGH){
        suppliedAmount += 0.01;
        Serial.print("\r - Supplied amount: " + String(suppliedAmount) + "L");
    }
    Serial.println();

    // Shutoff pump
    digitalWrite(PUMP_CONTROL_RELAY_PIN, LOW);

    Serial.println(" - Fuel supplied");

    // Decrement stock level
    set_stock(get_stock() - suppliedAmount);

    ESP_ERROR_CHECK(lock_pump());

    return ESP_OK;
}
```

Listing 1.3: function supply_fuel

1.3.4 Plate Reader

The Plate Reader module is located in *license_plate_recognition/plate−reader_device* folder and is responsible for continuously taking images of it's surroundings, so that license plates can be identified. Using an ESP32-CAM, the image is taken and stored in the SD Card and can be accessed by a GET request made to the onboard API server.

1.3.4.1 PlatformIO project

The License Plate device project was developed using PlatformIO Visual Studio Code extension. The Arduino framework was chosen due it's extensible library and possibility of use with the ESP32CAM. The project is structured with Header files in the *include* folder and source files in the *src* folder, as to decrease dependencies between packages. In the next sub-sections, we'll be approaching each module and it's purpose.

1.3.4.2 CommsHandler package

This package is responsible for declaring and handling the communication methods of the ESP32. It establishes connectivity to a WiFi AP (with credentials in a local $WiFiCredentials.h$ file) and waits for a NTP (Network Time Protocol) server, using the ArduinoJSON library, to reply with the current time, as to have accurate Timestamps locally. After the NTP response is received, it'll establish connection to the MQTT broker, using WebSockets, subscribe to it's downlink topic and bind the callback function for the subscribe topic/s. Subsequent access to this package operation is done mainly to publish data to the MQTT broker. The next step is to create and bind the API server on the ESP, so computers in the network can access it. For that, we used the aWOT library, which creates three API endpoints:

1. $/images$: accessed via GET request and returns the latest saved image in the SD card
2. $/swagger.json$: accessed via GET request and returns the Swagger JSON file
3. $/swaggerUI$: accessed via GET request and returns a web-page with the Swagger UI

Due to the simplicity on the uses of the API, all endpoints are processed in one callback function:

```
void CommsHandler::process_api_request(Request &request, Response &
    response){
    // Checks the path
    if(strcmp(request.path(), IMAGE_URL_PATH) == 0) {
        // let the user request the last image, without need for id
        long imageId = lastImageId;

        // Check the query parameters
        char id[50];
        request.query("id", id, 50);
        if(strlen(id) > 0)
            imageId = atol(id);

        String strPath = get_image_path(imageId);
```

```
        char path[strPath.length() + 1];
        strPath.toCharArray(path, strPath.length() + 1);

        // Check if the image exists
        fs::FS &fs = SD_MMC;
        File file = CardHandler::read_data(fs, path);
        if(file.size() == 0){
            response.sendStatus(404); // image not present
            return;
        }

        // Set the headers and send the response
        response.set("Content-Type", "image/jpeg");
        response.set("Connection", "close");

        // Allocate memory for the sending buffer
        if(!psram.allocate(READ_FILE_BUFFER_SIZE)){
            response.sendStatus(500);
            return;
        }

        // Continuously read and send the image in the buffer
        int bytesRead = 0;
        do{
            bytesRead = file.readBytes((char*) psram.get_mem_ptr(), READ_FILE_BUFFER_SIZE);
            response.write(psram.get_mem_ptr(), bytesRead);
        }while (bytesRead > 0);

        // Free the memory
        psram.destroy();
        file.close();

        // Finish response
        response.end();

        // Delete image from SD Card
        if(!CardHandler::delete_file(fs, path))
            Serial.println("Error: Problem occurred deleting photo " + strPath + " from SD card");

        Serial.println("Image " + String(imageId) + " sent to service and deleted from SD card");

    } else if(strcmp(request.path(), JSON_URL_PATH) == 0){
        response.set("Content-Type", "application/json");
        response.set("Connection", "close");

        String wifiStr = WiFi.localIP().toString();
        const char* ipAddress = wifiStr.c_str();

        response.write((uint8_t *) swaggerJSONPart1, strlen(swaggerJSONPart1));
        response.write((uint8_t *) ipAddress, wifiStr.length());
        response.write((uint8_t *) swaggerJSONPart2, strlen(swaggerJSONPart2));

```

```
        response.end();

    } else if(strcmp(request.path(), DOCS_URL_PATH) == 0){
        response.set("Content-Type", "text/html");
        response.set("Connection", "close");

        String wifiStr = WiFi.localIP().toString();
        const char* ipAddress = wifiStr.c_str();

        response.write((uint8_t *) swaggerUIPart1, strlen(
    swaggerUIPart1));
        response.write((uint8_t *) ipAddress, wifiStr.length());
        response.write((uint8_t *) swaggerUIPart2, strlen(
    swaggerUIPart2));

        response.end();

    } else {
        response.sendStatus(400);
        return;
    }
}
```

Listing 1.4: api request processing function

1.3.4.3 PSRAM package

This package is used to take advantage of the external RAM (Pseudo-static RAM) found in the ESP32-CAM, to buffer images read from the SD card and use in the API server, when a request is received to download an image. This approach saves up internal RAM, which guarantees that operation of the API server doesn't affect the overall functioning of the application. This package has functions to reserve, allocate and destroy memory, as well as create Memory streams for previous approaches. This package is also guaranteed to have no memory-leaks, provided that the functions are used in their intended way, thus abstracting to the programmer the operations in memory.

1.3.4.4 cardHandler package

The cardHandler package handles the interaction with the SD card, mainly setting up the directory, storing and reading the images. As mentioned before, the SD card is used as a mean to "cache" the images so that they can be accessed and processed by an external service

1.3.4.5 cameraHandler package

This package interacts with the physical camera of the device (ESP32CAM), allowing it to take pictures (which are saved in the PSRAM of the device) and return an object (pointer), with which we can save in a higher capacity and non-volatile memory (SD card).

1.3.5 Ditto

Eclipse Ditto is a technology in the IoT realm implementing a software pattern called "digital twins". A digital twin is a virtual, cloud based, representation of his real world counterpart (real world "Things", e.g. devices like sensors, smart heating, connected cars, smart grids, EV charging stations, ...). Ditto can potentially mirrors millions and billions of digital twins residing in the digital world with physical "Things". With Ditto a thing can just be used as any other web service via it's digital twin. Meaning that there isn't direct connection between the thing and the service, as Ditto handles this connection. The main advantage of using a platform like Ditto is that there doesn't need to exist direct connection between devices or services, so if one of the entities in our system goes offline, it can always have access to the latest information, without need to talk to the other entities. Our Ditto implemented Ditto architecture can be viewed in Figure 1.1. This module was developed by Gonçalo Silva.

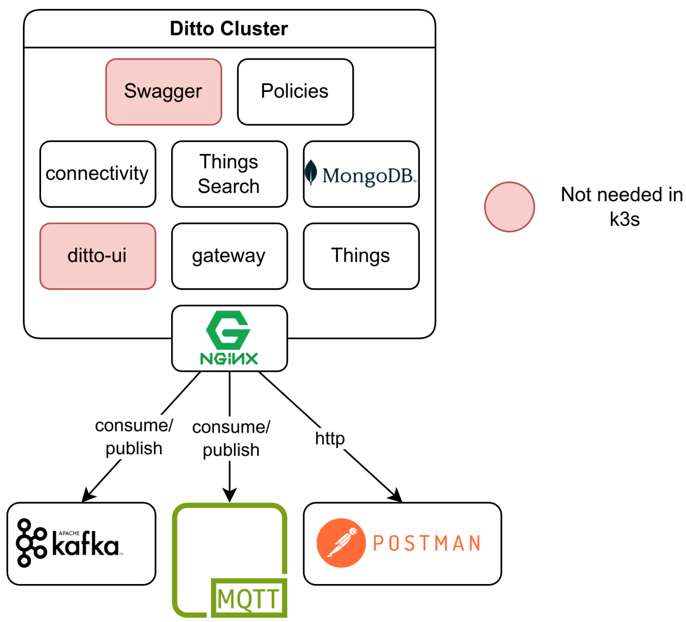

Figure 1.1: Implemented Ditto architecture

1.3.6 Devices

We use Ditto to represent our real-world devices (Gas Pump, Plate Reader), in our digital services. For this, representations of this devices need to first be created in Ditto. They area defined using the JSON-LD file format, with three main fields:

1. Attributes: metadata/information about a thing. Ex: the color or manufacturer of a vehicle
2. Features: represents the different states that a thing can have or information that can be controlled/changed. Ex: lights of a house (status, color)
3. Definition: The file/field which contains all this values

The definitions for both of the things can be found below:

```
{
    "@context": [
    "https://www.w3.org/2022/wot/td/v1.1",
    {
      "om2": "http://www.ontology-of-units-of-measure.org/resource/
    om-2/",
      "schema": "http://schema.org/"
    }
    ],
    "@type": "tm:ThingModel",
    "title": "Gas Pump",
    "description": "A device tasked with locking, unlocking,
    monitoring and supplying fuel at the Gas Pump.",
    "version": {
       "model": "1.0.0"
    },
    "attributes": {
    "deviceID": {
      "title": "Device ID",
      "type": "string",
      "readOnly": true
    },
    "capacity": {
      "title": "Capacity of the fuel tank",
      "type": "number"
    },
    "location": {
      "title": "Location",
      "description": "GPS coordinates of the mounted location (
    latitude, longitude)"
    }
  },
  "features": {
    "authorize_supply": {
      "title": "Authorize Supply",
      "description": "Event directed to the Pump, signalling that
    it's authorized to supply fuel to a vehicle.",
      "properties": {
        "timestamp": {
          "title": "Timestamp",
          "type": "string",
          "format": "date-time",
          "description": "Datetime when authorization was made."
        },
        "authorization": {
          "title": "Authorization",
```

```
          "type": "boolean",
          "description": "Boolean signalling True, to concede authorization or False otherwise."
        },
        "msgType": {
          "title": "Message type",
          "type": "number",
          "description": "Type of the message."
        }
      }
    },
    "supply_completed": {
      "title": "Supply Completed",
      "description": "Event originating from the pump, indicating that the supply of the vehicle was conducted.",
      "properties": {
        "timestamp": {
          "title": "Timestamp",
          "type": "string",
          "format": "date-time",
          "description": "Datetime when supply was completed."
        },
        "amount": {
          "title": "Amount",
          "type": "double",
          "description": "Number of liters supplied."
        },
        "stock": {
          "title": "Stock",
          "type": "number",
          "description": "Current stock of fuel in the pump."
        },
        "msgType": {
          "title": "Message type",
          "type": "number",
          "description": "Type of the message."
        }
      }
    },
    "fuel_replenishment": {
      "title": "Fuel Replenishment",
      "description": "Event originating from the pump, indicating that the pump was replenished with fuel.",
      "properties": {
        "timestamp": {
          "title": "Timestamp",
          "type": "string",
          "format": "date-time",
          "description": "Datetime when pump was replenished."
        },
        "amount": {
          "title": "Amount",
          "type": "double",
          "description": "Number of liters replenished."
        },
        "stock": {
          "title": "Stock",
```

```
          "type": "number",
          "description": "Current stock of fuel in the pump."
        },
        "msgType": {
          "title": "Message type",
          "type": "number",
          "description": "Type of the message."
        }
      }
    },
    "supply_error": {
      "title": "Supply Error",
      "description": "Event originating from the pump, indicating
    that there was an error supplying the vehicle.",
      "properties": {
        "timestamp": {
          "title": "Timestamp",
          "type": "string",
          "format": "date-time",
          "description": "Datetime when pump was replenished."
        },
        "error": {
          "title": "Error message",
          "type": "string",
          "description": "Reason of the Error."
        },
        "msgType": {
          "title": "Message type",
          "type": "number",
          "description": "Type of the message."
        }
      }
    },
    "pump_init": {
      "title": "Pump Initialization",
      "description": "Event originating from the pump, indicating
    that the pump was initialized.",
      "properties": {
        "timestamp": {
          "title": "Timestamp",
          "type": "string",
          "format": "date-time",
          "description": "Datetime when pump was initialized."
        },
        "stock": {
          "title": "Stock",
          "type": "number",
          "description": "Current stock of fuel in the pump."
        },
        "capacity": {
          "title": "Capacity",
          "type": "number",
          "description": "Maximum fuel capacity of the pump."
        },
        "msgType": {
          "title": "Message type",
          "type": "number",
```

```
          "description": "Type of the message."
        }
      }
    }
  }
}
```

Listing 1.5: Definition of the Gas Pump device

```
{
    "@context": [
    "https://www.w3.org/2022/wot/td/v1.1",
    {
      "om2": "http://www.ontology-of-units-of-measure.org/resource/om-2/",
      "schema": "http://schema.org/"
    }
    ],
    "@type": "tm:ThingModel",
    "title": "Plate Reader",
    "description": "An automated license plate reader (ALPR) device",
    "version": {
       "model": "1.0.0"
    },
    "attributes": {
    "deviceID": {
      "title": "Device ID",
      "type": "string",
      "readOnly": true
    },
    "fieldOfView": {
      "title": "Field of View",
      "type": "object",
      "properties": {
      "horizontalAngle": {
        "title": "Horizontal Angle",
        "type": "number",
        "unit": "om2:Degree"
      },
      "verticalAngle": {
        "title": "Vertical Angle",
        "type": "number",
        "unit": "om2:Degree"
      }
      }
    },
    "cameraResolution": {
      "title": "Camera Resolution",
      "type": "number",
      "unit": "schema:Megapixel"
    },
    "imageCaptureRate": {
      "title": "Image Capture Rate",
      "type": "number",
      "unit": "schema:FramePerSecond"
    },
    "location": {
```

```
      "title": "Location",
      "type": "geo:point",
      "description": "GPS coordinates of the mounted location (
    latitude, longitude)"
    }
  },
  "features": {
    "imageCaptured": {
      "title": "Plate Recognized",
      "description": "Event triggered when the device recognizes a
    license plate.",
      "properties": {
        "timestamp": {
          "title": "Timestamp",
          "type": "string",
          "format": "date-time",
          "description": "Datetime when the plate was recognized."
        },
        "imageId": {
          "title": "Image ID",
          "type": "string",
          "description": "Unique identifier of the captured image."
        },
        "url": {
          "title": "Image URL",
          "type": "string",
          "description": "URL of the captured image."
        }
      }
    }
    }
}
```

Listing 1.6: Definition of the Plate Reader device

As present in this device definitions, both devices have multiple attributes, with their *deviceID* being the only one read-only. Both of them have features, which are used to exchange information and propagate to the other services. With the devices defined, the next thing is to create the thing. Multiple things can be created using the same definition, since they are identified, within the system, by their auto-generated UUID and Namespace. Using the above definitions, which are hosted publicly in Github, we can create a new thing with the following command:

```
curl -u ditto:ditto -X POST -H 'Content-Type: application/json' -d
    '{
    "title": "My Plate Reader",
    "description": "ALPR device",
    "definition": "http://raw.githubusercontent.com/Fuel-Link/
    license_plate_recognition/main/ditto/plate-reader.jsonld",
    "attributes": {
        "deviceID": "plate_reader_1234"
    }
}' 'http://localhost:8080/api/2/things'
```

Listing 1.7: Creation of Plate Reader

This request will output in the terminal it's response, which is the generated **ThingID** in the format $Namespace : UUID$. This value will be used in our system to track the device that is sending the information.

1.3.6.1 Connectivity

Ditto can be Queried/Controlled through various technologies, using it's very extensive Connectivity API service, as can be seen in Figure 1.2. In our case, since both devices communicate with an MQTT broker, and the services through Kafka, we created one connection handle for each. With this in mind, Ditto, Kafka and MQTT should have connectivity between each other, allowing them to send/receive information between them. In our case, uplink information from the devices is always propagated to a configured kafka uplink topic, while downlink is only defined for the Gas Pump and only downlink data will be published back to the MQTT Broker.

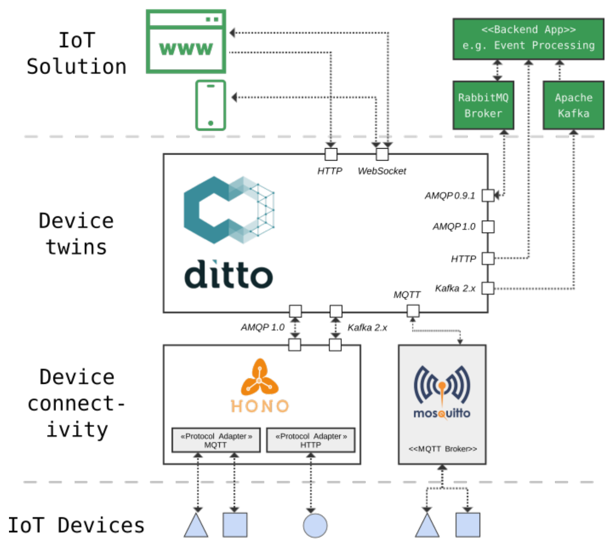

Figure 1.2: Ditto Connectivity

It's worth mentioning that there exists one Ditto service in each device module, because an initial idea would be to have two deployed Ditto instances, which would increase reliability and redundancy in the system, allowing to have better scalability, in separate entities for each device (ex: the number of gas pumps would probably be higher than Plate Reader devices, due to multiple fuel types). So, taking this approach, each module has a separate connection file, for each connection type, with the main change being the topics chosen. Below there are present two out of the four connection requests, made to create the connection in Ditto.

```
curl -u ditto:ditto -X POST -H 'Content-Type: application/json' -d
    '{
```

```
    "targetActorSelection": "/system/sharding/connection",
    "headers": {
      "aggregate": false
    },
    "piggybackCommand": {
        "type": "connectivity.commands:createConnection",
        "connection": {
            "id": "mqtt_localhost_connection",
            "name": "mqtt",
            "connectionType": "mqtt",
            "connectionStatus": "open",
            "failoverEnabled": true,
            "uri": "tcp://192.168.68.111:1883",
            "sources": [
                {
                "addresses": [
                    "plate-reader/+/uplink/#"
                ],
                "qos": 0,
                "authorizationContext": [
                    "nginx:ditto"
                ],
                "filters": []
                }
            ],
            "targets": [
                {
                "address": "plate-reader-evaluation/{{ thing:id }}/
    downlink",
                "topics": [
                    "_/_/things/twin/events",
                    "_/_/things/live/messages"
                ],
                "qos": 0,
                "authorizationContext": [
                    "nginx:ditto"
                ]
                }
            ],
            "mappingContext": {
                "mappingEngine": "JavaScript",
                "options": {
                "incomingScript": "function mapToDittoProtocolMsg(\
    n    headers, \n    textPayload, \n    bytePayload,\n
    contentType\n) {...}\n  "
                }
            }
        }
    }
}' 'http://localhost:8080/devops/piggyback/connectivity?timeout=10'
```

Listing 1.8: Plate Reader Ditto Connection to MQTT broker

```
curl -u ditto:ditto -X POST -H 'Content-Type: application/json' -d
    '{
    "targetActorSelection": "/system/sharding/connection",
    "headers": {
      "aggregate": false
```

```
    },
    "piggybackCommand": {
        "type": "connectivity.commands:createConnection",
        "connection": {
            "id": "mqtt_localhost_connection",
            "name": "mqtt",
            "connectionType": "mqtt",
            "connectionStatus": "open",
            "failoverEnabled": true,
            "uri": "tcp://192.168.167.79:1884",
            "sources": [
                {
                    "addresses": [
                    "gas-pump_downlink"
                    ],
                    "consumerCount": 1,
                    "qos": 1,
                    "authorizationContext": [
                    "nginx:ditto"
                    ],
                    "headerMapping": {
                    "message-id": "{{ header:correlation-id }}",
                    "content-type": "application/vnd.eclipse.ditto+json"
                    },
                    "replyTarget": {
                    "enabled": true,
                    "address": "kafka-errors",
                    "headerMapping": {
                        "message-id": "{{ header:correlation-id }}",
                        "content-type": "application/vnd.eclipse.ditto+json"
                    },
                    "expectedResponseTypes": [
                        "response",
                        "error",
                        "nack"
                    ]
                    },
                    "acknowledgementRequests": {
                    "includes": []
                    },
                    "declaredAcks": []
                }
            ],
            "mappingContext": {
                "mappingEngine": "JavaScript",
                "options": {
                        "incomingScript": "function mapToDittoProtocolMsg(\n    headers, \n    textPayload, \n bytePayload,\n    contentType\n) {...}\n  "
                }
            }
        }
    }
```

```
}' 'http://localhost:8080/devops/piggyback/connectivity?timeout=10'
```

Listing 1.9: Gas Pump connection to Kafka

1.3.6.2 Payload Mappers

Ditto internally only processes data when in the Ditto Format, however this format is based on JSON and very intensive on the fields and data needed to process. That's where the concept of Payload Mapping appears, which is to transform or map messages in a smaller, more efficient format (especially useful when dealing with micro-controllers), into a full Ditto Format message. For this, Ditto counts with various built-in payload mappers, which try to match incoming messages, into them. This payload mappers can also be applied to messages from/to Ditto, which is specially useful, making it so that the Ditto system can be implemented in an existing environment, without having to change any of the devices message formats. Unfortunately, in our case, the default payload mappers weren't working, so we proceed to create JavaScript functions, used by Ditto, for that effect. Both of the devices have incoming payload mappers, but only the Gas Pump has an outgoing mapper. The definition of the incoming payload mapper for the Plate Reader module can be found below:

```
function mapToDittoProtocolMsg(
    headers,
    textPayload,
    bytePayload,
    contentType
) {
    const jsonData = JSON.parse(textPayload || "{}"); // Handle empty payload
    const thingId = jsonData.thingId.split(':');
    const timestamp = jsonData.timestamp;
    const imageId = jsonData.imageId.toString(); // Ensure imageId is a string
    const url = jsonData.url;

    const value = {
        imageCaptured: {
            properties: {
                timestamp: {
                    value: jsonData.timestamp
                },
                imageId: {
                    properties: {
                        value: jsonData.imageId
                    }
                },
                url: {
                    properties: {
                        value: jsonData.url
                    }
                },
            }
        }
```

```
    };
    return Ditto.buildDittoProtocolMsg(
        thingId[0],                     // thing namespace
        thingId[1],                     // thing ID of the device
        'things',                       // (group) we deal with a thing
        'twin',                         // (channel) we want to update
    the twin
        'commands',                     // (criterion) create a command
     to update the twin
        'modify',                       // (action) modify the twin
        '/features',                    // (path) modify all features
    at once
        headers,                        // pass the mqtt headers
        value
    );
}
```

Listing 1.10: Plate Reader Incoming payload mapper

The outgoing payload mapper for the Gas Pump device can be seen below. Note that with this approach, there are payload mappers for when both Kafka and MQTT produce messages, and allowing the creation of messages to be much simpler.

```
function mapToDittoProtocolMsg(
    headers,
    textPayload,
    bytePayload,
    contentType
) {

    const jsonData = JSON.parse(textPayload || "{}"); // Handle
    empty payload
    const thingId = jsonData.thingId.split(':');
    const jsonValue = jsonData.value;

    // Only handle authorized_supply messages
    let channel = "/features/authorize_supply";
    value = {
        properties: {
            timestamp: {
                properties: {
                    value: jsonValue.timestamp
                }
            },
            authorization: {
                properties: {
                    value: jsonValue.authorization
                }
            },
            msgType: {
                properties: {
                    value: jsonValue.msgType
                }
            }
        }
    };
    return Ditto.buildDittoProtocolMsg(
```

```
        thingId[0],                   // thing namespace
        thingId[1],                   // thing ID of the device
        'things',                     // (group) we deal with a thing
        'twin',                       // (channel) we want to update
    the twin
        'commands',                   // (criterion) create a command
     to update the twin
        'modify',                     // (action) modify the twin
        channel,                      // (path) modify only the
    chosen feature
        headers,                      // pass the kafka headers
        value
    );
}
```

Listing 1.11: Gas Pump outgoing payload mapper

Take the example of the *authorize_supply* message, sent by the supply manager (composer) to authorize the supply of fuel in the Gas Pump. A simple message sent to kafka like this:


```
{
    "thingId": "org.eclipse.ditto:9b0ec976-3012-42d8-b9ea-89
    d8b208ca20",
    "topic": "org.eclipse.ditto/9b0ec976-3012-42d8-b9ea-89
    d8b208ca20/things/twin/commands/modify",
    "path": "/features/authorize_supply/properties/",
    "messageId": "{{ uuid() }}",
    "timestamp": "{{ timestamp }}",
    "source": "gas-pump",
    "method": "update",
    "target": "/features/authorize_supply",
    "value": {
        "msgType": 1,
        "thingId": "org.eclipse.ditto:9b0ec976-3012-42d8-b9ea-89
    d8b208ca20",
        "topic": "org.eclipse.ditto/9b0ec976-3012-42d8-b9ea-89
    d8b208ca20/things/twin/commands/modify",
        "path": "/features/authorize_supply/properties/",
        "authorization": 1,
        "timestamp": "2024-05-18T12:03:44+0100"
    }
}
```


Listing 1.12: Authorize Supply message created by the supply manager

Is transformed or mapped by kafka into the following, more complex version:


```
{
   "topic": "org.eclipse.ditto/9b0ec976-3012-42d8-b9ea-89d8b208ca20
    /things/twin/events/modified",
   "headers": {
      "authorization": "Basic ZG10dG86ZG10dG8=",
      "x-real-ip": "172.28.0.1",
      "x-forwarded-user": "ditto",
      "x-ditto-pre-authenticated": "nginx:ditto",
      "postman-token": "b78dbd13-9335-449b-8732-089c7867578d",
      "host": "localhost:8080",
```

```
      "x-forwarded-for": "172.28.0.1",
      "accept": "*/*",
      "user-agent": "PostmanRuntime/7.37.3",
      "ditto-originator": "nginx:ditto",
      "response-required": false,
      "version": 2,
      "requested-acks": [],
      "content-type": "application/json",
      "correlation-id": "2542f36c-65a1-4959-b229-04f6186d6c6d"
   },
   "path": "/features/authorize_supply",
   "value": {
      "properties": {
         "timestamp": {
            "properties": {
               "value": "2024-05-18T12:03:44+0100"
            }
         },
         "authorization": {
            "properties": {
               "value": 1
            }
         },
         "msgType": {
            "properties": {
               "value": 1
            }
         }
      }
   },
   "extra": {
      "thingId": "org.eclipse.ditto:9b0ec976-3012-42d8-b9ea-89d8b208ca20"
   },
   "revision": 52,
   "timestamp": "2024-05-18T15:38:09.409779963Z"
}
```

Listing 1.13: DataTypes definition

And with more modifications and improvements, the published message could also be decreased, by utilizing more information present in it's headers (configured in KafkaJS or other Kafka integration service), as well as a better payload mapper function

1.3.6.3 Deployment

Attempts were made to deploy Ditto into the Kubernetes K3S cluster, using official methods and other approaches both in manual deployments. Helm was also tested, but the same problem occurred. Due to this issue, also observed by the professor, this modules couldn't be deployed, so they needed to be ran locally. However, since some components were still present in the cluster, including a kafka, a bridge/relay server and client needed to be made, as to have connectivity between the local and remote instances. However, this topic will be further discussed in the next subsection.

1.3.7 Kafka

Apache Kafka is an open-source distributed event streaming platform. It's extremely fast, scalable, has storage and supports a cluster environment with multiple brokers, always guaranteeing that messages or Events are processed and distributed in real-time and exactly once. Kafka acts as an Orchestrator, providing easy communication between the Composer and the Ditto environments, as well as with the Plate-Reader device. Our kafka contains three brokers and it's architecture can be observed in Figure 1.3. This module was developed by Gonçalo Silva.

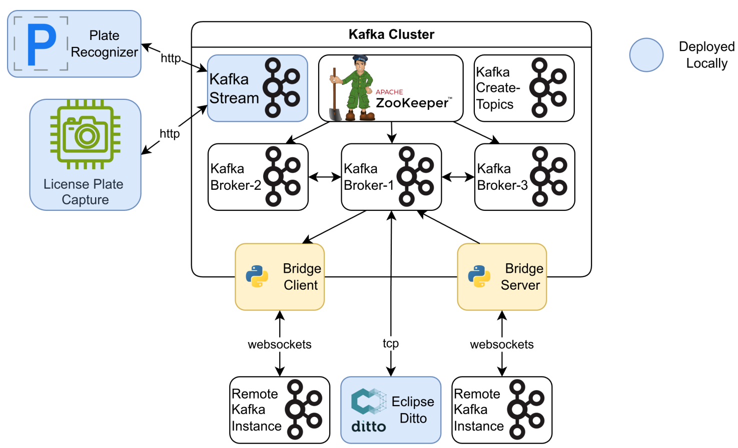

Figure 1.3: Kafka deployment architecture

1.3.7.1 Topic creation

Inside the cluster, some components wait for the topics to be created before initializing. This is a good approach for a production environment, because we guarantee that the system is first configured before the applications can proceed, while also denying bad actors to operate on new topics unrestrictedly. However, since we are operating in a development environment, we let users create topics freely in the Kafka cluster, while also having a dedicated container, which creates the required topics for the applications to work.

1.3.7.2 Bridge server and client

Due to blocking of UA firewalls, Kafka couldn't be completely deployed into Kubernetes, since they block TCP socket connections, and since Kafka has to have connection with the local Ditto instance, we needed to have a solution. So we came up with a bridge/relay server, which consumes data from the local Kafka cluster, connected to Ditto and injects it into the deployed one, and vice-versa. Taking advantage of the already deployed MQTT broker, operating with WebSockets (once again, to bypass UA firewall restriction), we developed two programs. Both programs consume/produce data from their respective kafka instances, but they are interconnected using the MQTT broker. The Bridge Client application is deployed locally, while the Bridge Server is deployed in Kubernetes.

1.3.7.3 Kafka Streams

Kafka processes streams of events with joins, aggregations, filters, transformations, etc., using event-time and exactly-once processing. Meaning that we can create quite powerful applications which reside "inside" of the cluster, taking advantage of internal accesses and are triggered once a certain topic or condition is met. Taking advantage of this functionality, we created a Kafka stream, which consumes data from the Plate Reader device, retrieves the image using the API server in the device and queries an external service for recognition of License Plates from images (platerecognizer.com). If this service returns that a License Plate was found, the result and other necessary data is published into another topic. An example of the response from this service, with a clear license plate image is:

```
"processing_time": 106.46,
"results": [
    {
        "box": {
            "xmin": 22,
            "ymin": 84,
            "xmax": 821,
            "ymax": 386
        },
        "plate": "aa00aa",
        "region": {
            "code": "hk",
            "score": 0.034
        },
        "score": 0.897,
        "candidates": [
            {
                "score": 0.897,
                "plate": "aa00aa"
            }
        ],
        "dscore": 0.806,
        "vehicle": {
            "score": 0.0,
            "type": "Unknown",
            "box": {
                "xmin": 0,
                "ymin": 0,
                "xmax": 0,
                "ymax": 0
            }
        }
    }
],
"filename": "1410_VbZQQ_matricula.jpg",
"version": 1,
"camera_id": null,
"timestamp": "2024-05-28T14:10:26.848840Z"
```


Figure 1.4: Plate Recognizer response to an image

As with the rest of the cluster, this module is also packaged into a container, with us having to package the required libraries, code and compilation into this container. This stream is located in $orchestrator/kafka/docker/kafka-streams/applications/PlateRecognizer.java$.

1.3.8 Composer

With all the services in place and ready for use, Composer took on the role of conductor, orchestrating seamless interaction between them to achieve the desired final product. This Module was developed by Catarina Barroqueiro

1.3.8.1 Backend

To provide the administrator with convenient access to all data, an Angular web application was developed to display the dashboards. This application connects to a NodeJS backend that integrates NestJS and TypeORM. The backend is responsible for data publishing, deletion, and retrieval, interacting directly with a PostgreSQL database.

1.3.8.1.1 DataBase Implementation

To facilitate data interaction via APIs, a robust database was designed, as illustrated in the Figure 1.5 below. This database encompasses a wide range of information, from refueling records to user (employee) data and the company's vehicle fleet.

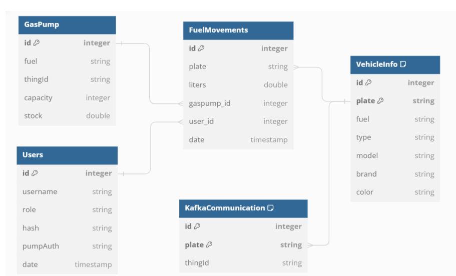

Figure 1.5: Database Diagram

To interact with the database tables, APIs were created for each of them, allowing insertion, deletion, filtered search and querying of stored data. The backend was developed in NodeJS with NestJS and TypeORM integrated. For each table, a module was created with four main files (Figure 1.6):

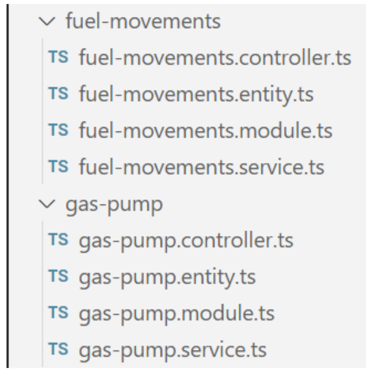

Figure 1.6: Module Initial Structure

- **Entity:** Define the table structure corresponding to the module (Figure 1.7).

```
import { Entity, PrimaryGeneratedColumn, Column, OneToMany, ManyToOne, JoinColumn } from 'typeorm';
import { VehicleInfo } from 'src/vehicle-info/vehicle-info.entity';

@Entity()
export class GasPump {
  @PrimaryGeneratedColumn()
  public gaspump_id!: number;

  @Column({ type: 'varchar', length: 120 })
  public fuel: string;

  @Column()
  public stock: number;

  @Column({unique:true})
  public thingId: string;

  @Column()
  public capacity: number;

}
```

Figure 1.7: GasPump Entity File

- **Service:** Define the functions that allow interaction with the database, called by the controller file (Figure 1.8).

```typescript
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { GasPump } from './gas-pump.entity';
import { Repository } from 'typeorm';

@Injectable()
export class GasPumpService {

    @InjectRepository(GasPump)
    private readonly repository: Repository<GasPump>;

    async findAll() {
        return await this.repository.manager.query('Select * from "gas_pump"');
    }

    async save(gasPump: { fuel: string, stock: number, capacity: number, thingId: string }) {
        return await this.repository.save(gasPump);
    }

    async remove(id: string) {
        return await this.repository.delete(id);
    }
}
```

Figure 1.8: GasPump Service Initial File

- **Controller:** Define the APIs available for the module in question (Figure 1.9).

```
@Controller('gas-pump')
export class GasPumpController {
    res;

    constructor(private readonly gaspumpservice: GasPumpService,private readonly appService: AppService) { }



    @Get('')
    async getGasPump(@Req() request: Request, @Headers() headers: { authorization: string }) {

        let api = {
            op: 'Get Gas Pump',
            date: moment().toString(),
            request: request,
            result: null,
            validation: null
        };

        try {
            api.result = await this.gaspumpservice.findAll();
            if (api.result !== null) {
                this.res = this.appService.handleResponse(true, 'Done! ✔', HttpStatus.OK, api);
            } else {
                this.res = this.appService.handleResponse(false, 'Server error! ❌', HttpStatus.INTERNAL_SERVER_ERROR, api);
            }
        } catch (error) {
            api.validation = null;
            api.result = error;
            this.res = this.appService.handleResponse(false, 'Server error! ❌', HttpStatus.INTERNAL_SERVER_ERROR, api);
        }
        return this.res;

    }
```

Figure 1.9: GasPump Get on GasPump Controller file

- **Module:** Define the dependencies and relationships of each module (imports, exports, suppliers, etc.).

```
import { Module } from '@nestjs/common';
import { GasPumpController } from './gas-pump.controller';
import { GasPumpService } from './gas-pump.service';
import { TypeOrmModule } from '@nestjs/typeorm';
import { GasPump } from './gas-pump.entity';
import { AppService } from 'src/app.service';
@Module({
  imports:[TypeOrmModule.forFeature([GasPump])],
  controllers: [GasPumpController],
  providers: [GasPumpService, AppService],
  exports:[GasPumpService]
})
export class GasPumpModule {}
```

Figure 1.10: GasPump Module File

1.3.8.1.2 Kafka Integration

Both the Plate Recognizer service and the pump login service post their information (recognized plates and users who successfully logged in) in different

Kafka topics. This information is relevant for the correct tracking of all supply movements, at least To obtain them, a Consumer was initialized (Figure 1.11), which, depending on the topic that receives a given message, if it is relevant, will post the relevant data in the respective table. This allows the backend to retrieve the license plate and users data for further processing.

```
private async initializeKafkaConsumer() {
    await kafkaConsumer.connect();
    await kafkaConsumer.subscribe({ topics: ['plateRecognized', 'gas-pump_uplink', 'gas-pump_auth'] });
    await kafkaConsumer.run({
        eachMessage: async ({ topic, partition, message }) => { ...
        },
    });
}
```

Figure 1.11: Kafka Consumer Init

In addition to the Kafka Consumer, a Producer was also created. This Producer posts a message to another Kafka topic, determining whether to unlock the pump based on the success or failure of the Keycloak mobile authentication login. This sending only occurs when the send function (Figure 1.12) is called, invoked whenever the topic receives a bomb unlock request from an authenticated user.

```
async send(authorized: string): Promise<void> {
    await kafkaProducer.connect();
    await kafkaProducer.send({
        topic: 'gas-pump_downlink',
        messages: [{ value: authorized }],
    });
    await kafkaProducer.disconnect();
}
```

Figure 1.12: Send Function

These implementations were only possible due to the initialization of kafka in the NodeJS project through the kafka.config file, where the essential information is provided for the connection to be established (Figure 1.13).

```
import { Kafka, logLevel } from 'kafkajs';
const kafka = new Kafka({
 clientId: 'hello-world',
 brokers:[
    "kafka:29092",
    "kafka:29092"
],
logLevel: logLevel.ERROR,
});
export const kafkaProducer = kafka.producer();
export const kafkaConsumer = kafka.consumer({ groupId: 'hello-world' });
```

Figure 1.13: kafka.config file

1.3.8.2 Web App

To provide the company administrator with a comprehensive and useful view of the database data, a Web App was developed in Angular that integrates with other services. This Web App presents three main screens after successful login:

- **Fleet Management Page:** Displays all vehicles in the fleet (Figure 1.14). By clicking on a vehicle, the user can view its refueling history, photo and characteristics (Figure 1.15).

Figure 1.14: Fleet Management Page

Figure 1.15: Details Page

- **Gas Pump Management Dashboard** Displays essential information, including the current fuel stock at the company's pump, the latest movements near the pump (fleet plates recognized by the Plate Recognizer service) and recent refueling. Both lists can be exported to an Excel file for easier searching. Additionally, this panel displays alerts that indicate replenishment deviations outside of usual consumption patterns (Figure 1.16).

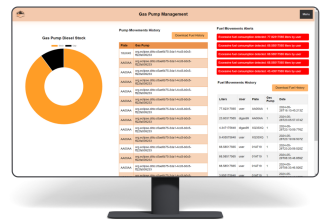

Figure 1.16: Gas Pump Management Dashboard

- **Fuel Forecast Dashboard:** Presents a simple graph with the fuel price forecast and a recommendation for the administrator on whether or not to order more fuel, based on price changes (Figure 1.17).

```
fetchPredictions() {
  const orgName = 'Fuel-Link';
  const authToken = '..............';
  const influxDBToken = '..............';
  const days = 7;

  const params = new HttpParams()
    .set('org', orgName)
    .set('authToken', authToken)
    .set('token', influxDBToken)
    .set('days', days.toString()); // Convert days to string as HttpParams expects strings

  const url = 'http://grupo1-egs-deti.ua.pt/market-analysis/predict';

  this.http.get<any>(url, { params }).subscribe(
    (response) => {
      this.predictions = response.predictions;
      this.decision = response.decision;

      // Split the data into past and future datasets
      const currentDate = new Date();
      this.pastData = this.predictions.filter((d: any) => new Date(d.ds) < currentDate);
      this.futureData = this.predictions.filter((d: any) => new Date(d.ds) >= currentDate);

      this.createLineChart();
    },
    (error) => {
      console.error('Error fetching predictions:', error);
    }
  );
}
```

Figure 1.17: Fuel Forecast Dashboard

1.3.8.2.1 Keycloak Integration

In order to access the web pages specified above, the user will have to log in through the authentication service based on the implementation of Keycloack (Figure 1.18).

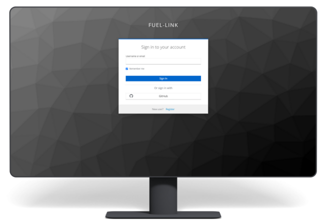

Figure 1.18: Keycloak Login Page

To make the login page available, the authentication service had to be integrated into the Web App by adding the appropriate parameters related to Keycloak within the ApplicationConfig (Figure 1.19).

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routeConfig),
    provideClientHydration(),
    KeycloakService,
    {
      provide: APP_INITIALIZER,
      useFactory: initializeKeycloak,
      multi: true,
      deps: [KeycloakService],
    },
    {
      provide: HTTP_INTERCEPTORS,
      useClass: KeycloakBearerInterceptor,
      multi: true,
    },
    provideHttpClient(
      withInterceptorsFromDi() // tell httpClient to use interceptors from DI
    )
  ],
};
```

Figure 1.19: App Config with Keycloak

A service was then created, with the help of the keycloak-angular library, which provides and initializes all functions useful to the project (Figure 1.20).

```
import { Injectable } from '@angular/core';
import { KeycloakService } from 'keycloak-angular';

@Injectable({ providedIn: 'root' })
export class KeycloakOperationService {
  constructor(private readonly keycloak: KeycloakService) {}

  isLoggedIn(): boolean {
    return this.keycloak.isLoggedIn();
  }
  logout(): void {
    this.keycloak.logout();
  }
  getUserProfile(): any {
    return this.keycloak.loadUserProfile();
  }
}
```

Figure 1.20: Keycloak Service

With everything correctly configured, we were able to implement AuthGuard, which allows users to authenticate correctly. The AuthGuard extends the KeycloakAuthGuard class, leveraging the keycloak-angular library to handle authentication and authorization. Below is a brief explanation of the AuthGuard implementation (Figure 1.21):

```
@Injectable({
  providedIn: 'root',
})
export class AuthGuard extends KeycloakAuthGuard {
  constructor(
    protected override readonly router: Router,
    protected readonly keycloak: KeycloakService
  ) {
    super(router, keycloak);
  }

  public async isAccessAllowed(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ) {
    // Force the user to log in if currently unauthenticated.
    if (!this.authenticated) {
      await this.keycloak.login({
        redirectUri: window.location.origin + state.url,
      });
    }
    // Get the roles required from the route.
    const requiredRoles = route.data['roles'];
    // Allow the user to proceed if no additional roles are required to access the route.
    if (!Array.isArray(requiredRoles) || requiredRoles.length === 0) {
      return true;
    }
    // Allow the user to proceed if all the required roles are present.
    return requiredRoles.every((role) => this.roles.includes(role));
  }
}
```

Figure 1.21: AuthGuard file

The AuthGuard checks if the user is authenticated; if not, it triggers the Keycloak login process. It also verifies if the user has the necessary roles to access specific routes, ensuring secure and role-based access control.

To ensure that access to the app always passes through the login process, the AuthGuard must be added to the route configuration in the routes.ts file. This guarantees that any access to the specified routes requires authentication. The configuration is as follows (Figure 1.22):

```
const routeConfig: Routes = [
    {
        path: '',
        component: HomeComponent,
        title: 'Home Page',
        canActivate:[AuthGuard]
    },
```

Figure 1.22: route.ts file

By adding 'canActivate: [AuthGuard]' to the route configuration, we ensure that the AuthGuard is invoked whenever a user attempts to navigate to the home page or any other protected route. This setup forces users to authenticate through Keycloak before they can access these routes, providing a secure and controlled access mechanism within the application.

1.3.8.2.2 Market Analysis Implementation

For the Market Analysis feature, an instance of Composer is properly instantiated along with an InfluxDB database. This setup allows the system to retrieve both past and future data for analysis. The Market Analysis service calls InfluxDB to obtain the necessary data and makes decisions based on the data received.

The integration process involves the following steps, as implemented in the fetchPredictions method (Figure 1.23):

```
fetchPredictions() {
  const orgName = 'Fuel-Link';
  const authToken = '..............';
  const influxDBToken = '..............';
  const days = 7;

  const params = new HttpParams()
    .set('org', orgName)
    .set('authToken', authToken)
    .set('token', influxDBToken)
    .set('days', days.toString()); // Convert days to string as HttpParams expects strings

  const url = 'http://grupo1-egs-deti.ua.pt/market-analysis/predict';

  this.http.get<any>(url, { params }).subscribe(
    (response) => {
      this.predictions = response.predictions;
      this.decision = response.decision;

      // Split the data into past and future datasets
      const currentDate = new Date();
      this.pastData = this.predictions.filter((d: any) => new Date(d.ds) < currentDate);
      this.futureData = this.predictions.filter((d: any) => new Date(d.ds) >= currentDate);

      this.createLineChart();
    },
    (error) => {
      console.error('Error fetching predictions:', error);
    }
  );
}
```

Figure 1.23: fetchPredictions Function

1.3.8.2.3 Web App Final Architecture

Following the successful implementation of all the previous steps, we have finalized the Composer with the architecture shown below (Figure 1.24).

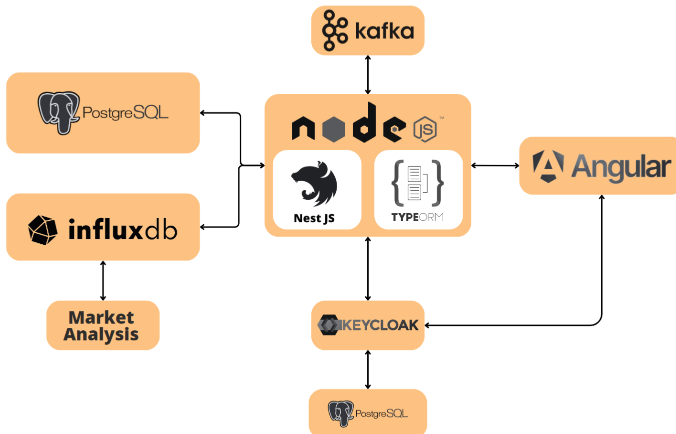

Figure 1.24: Composer Final Architecture

1.4 Deployment

Two services that were deploy on the Kubernetes cluster but never used locally were the Traefik Ingress and the Nginx reverse proxy. The Traefik Ingress redirects all traffic with our DNS name to our Nginx reverse proxy, which in turn redirects traffic to our services based on the URL path. While the Traefik Ingress has a very straight-forward deployment, the Nginx uses a ConfigMap to store the configurations. Some of the paths on the Nginx configuration required some forwarding of settings, and some even needed to allow upgrades to the HTTP protocol in order to use WebSockets.

Both the Keycloak and PostgreSQL services used for authentication are deployed with the usage of secrets so that their login credentials are not viewable by other users with access to the cluster. The PostgreSQL is deployed as a StatefulSet, while Keycloak, authentication at pump and the service to return all users use standard Deployments. All of these have Services that allow them to be accessed by other pods.

Kafka is deployed with StatefulSet brokers, as to have replication of three brokers and MQTT5 module uses configMap to bind the Mosquitto MQTT configuration file to the pod.

The remaining services are also deployed and can be found in the same repository.

1.5 Contributions

This projects contributions are divided in the following way:

1. Gonçalo Silva: Participation of 30%
2. Catarina Barroqueiro: Participation of 27.5%
3. Daniel Silva: Participation of 27.5%
4. Nuno Sousa: Participation of 15%