# Implementation of industrial standards on a SaaS for an MVP for logistics

**Metougui Taha**

**Supervised by:**

**Pr. Zineb BESRI**

**Mr. Othman DARRAZ**

# Abstract

# Acronyms

| | |
|---|---|
| DB | Database |
| KPI | Key performance indicator |
| MVP | Minimum viable product |
| PoC | Proof of concept |
| SaaS | Software as a service |

# Contents

# Chapter 1

# General Introduction

Chapter 2

# Context and Problematic

## 2.1 Introduction

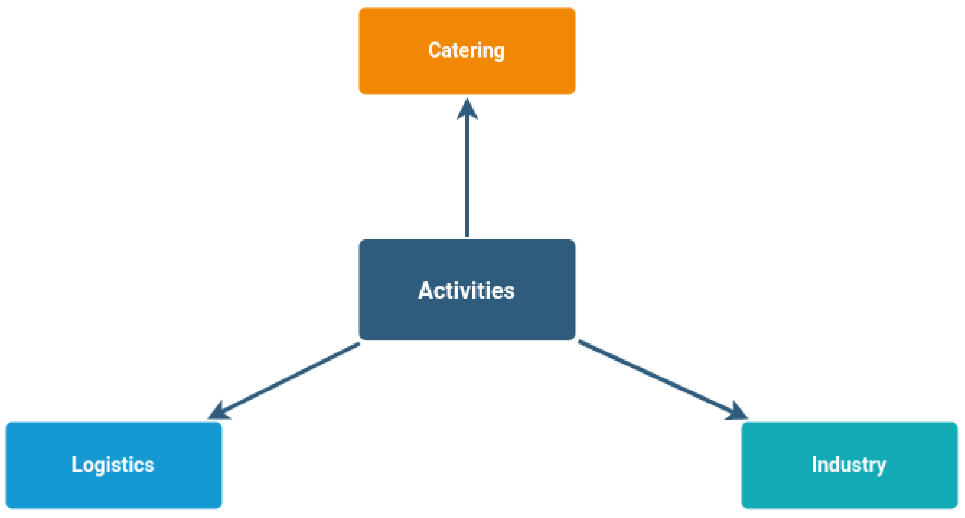
In the future, most if not all the services that rely on humans to organize and manage them will become automated, and autonomous, and thus, the human resources will be no longer needed. So, the need for a service or software that can automate the management of logistics is definitely a growing one.

So in the course of my engineering studies in ENSA of Tetouan, I have chosen to partake in the development of a service that can automate and ease the management of logistics factories workflow, within the company ONAR MMCall with a team of competent developpers, taking in charge everything that's related to backend and infrastructure of the SaaS.

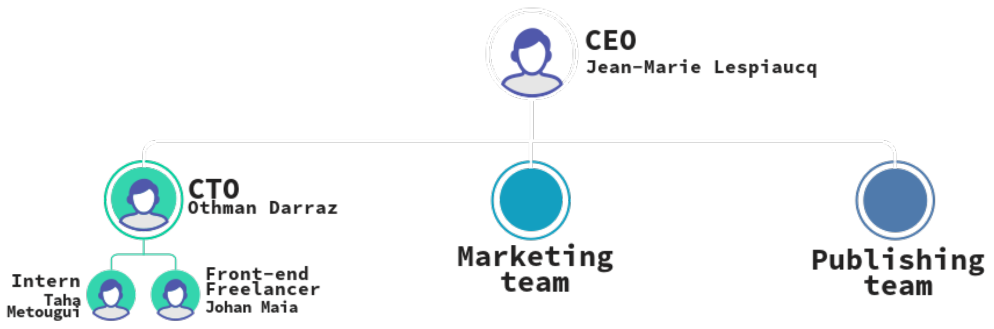
## 2.2 Host organism

The company ONAR MMSoft is a company that located at Jouy-en-Josas, France. It's part of a bigger company, ONAR MMCall, which specializes in the distribution of pagersm, in as a complement to this offering software that can blend with the pagers.

As of today, the company has worked in several fields of activities:



Mainly handling the problems of management of part of the workflows in all these fields. And lately turning all their focus to logistics as their field of interest.



The team, I've integrated, is composed of a team of 2 people, the CTO Othman Darraz who's in charge of managing the project and a freelancer Johan who's in charge of everything that's front related, and me joining them as the third member of the team taking care of backend mainly refactoring, optimization and security.

## 2.3 Work methodology

## 2.4 Problematic

So with the worflow management problem in hand, and the company's expertise in building system oriented to thise kind of problems, it has built a solution which is an MVP called "Easy Truck in" that takes in charge the management of the trucks workflow and the logistics of the deliveries. Giving multiple features and options but the solution is just a MVP and not a full-fledged product. So the main goal of this project is to adapt the MVP to industrial norms giving it a more

# Chapter 3

# Assessment of current state

## 3.1 Analysis

Having to take care of managing multiple trucks, truck drivers, their turns, their parking lots and their parking spaces can be a difficult task to achieve, especially when you have to deal with time management, availability, delays, and so on. However, going by old systems, either an Excel spreadsheet or just a ledger system on paper, can help such a task, as long as you're working on a small scale. But once this scales, the need for assistance grows.

And that's the point where comes the possibility to make use of automated tasks to do the job. Automating the management, making it easier to manage with as little human intervention as possible.

As of the current time, there is already a solution put in place as shown in figure **??** which are hosted on AWS implementing an SOA architecture. The SaaS relies on various managed services provided by AWS, such as EC2, S3, RDS, etc.

The MMSoft Saas is constituted of:

- Four front-end services:
  - Easy-truck-in (ETI) interface which is the main interface
  - Driver application (Angular 11)
  - Registration Tablet App (React 16)
  - Gatekeeper App (Cermate)
- Four Back-end services:
  - MQTT Broker communicating with the Gatekeeper app (VerneMQ)
  - Orchestration Back-end Service (Java 11 Spring Boot 2.6) used by the ETI interface
  - Registration Service (Node JS 16) used by the Registration Tablet App
  - TMS Registration Back-end which exposes REST API for integration (Java 11 Spring Boot 2.6)
- Two databases:
  - MySQL for users, site, configuration
  - Redis as a database for Daily/Weekly data and for messaging communication with other services

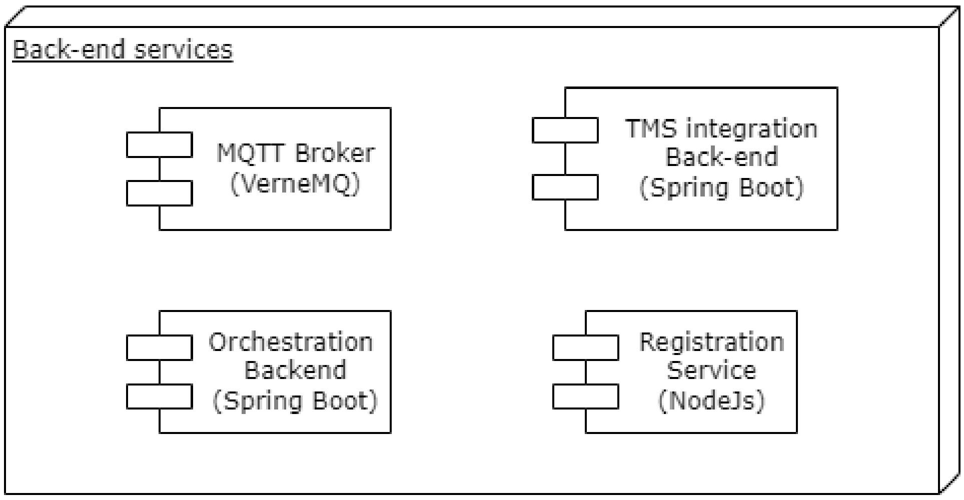

Back-end services
MQTT Broker (VerneMQ)
TMS integration Back-end (Spring Boot)
Orchestration Backend (Spring Boot)
Registration Service (NodeJs)

Figure 3.1: Backend services

## 3.2 Criticism

This already existing solution comes with its own set of problems.

Beginning from the most obvious one we have at hand is the usage of different technologies in the backend without any advantage in place for opting to this approach, such as NodeJs and Spring Boot, which in the long term affects the maintainability of the solution, as reported by the CTO, was developed by a freelancer and went untouched since. So rather than a maintainable solution, they opted for a momentary solution to have a PoC. While it's only handling the registration process, it would be better if it just migrated to the same stack of the whole system. That will allow the reusability of code, as there are already implementations for handling the Users Database on the Spring code base, and there is really no purpose to separate the systems.

For the databases as of the current solution, there are already two databases put in place a MySQL database to deal with any data that relates to users, the other is a Redis database to deal with real-time data such as tracking truck entries etc. Redis being the database to deal with real-time data, but the problem with redis is that it's load everything in memory so it's can be overloaded with costs of having memory coming at a great expense for some data that won't be casually needed. So a archival system for the data, where cost of the storage comes cheaper but, has low throughput for reads and writes would be perfect for such usage, the archival database would only take daily backups as afterwards this data would mostly be used either for KPIs, and thus the Redis database would get cleaned on a daily basis not having extra data taking out memory.

Furthermore, going through the existing code base, the secrets that relate to anything that is credentials or encoding keys are all exposed in the code, in terms of industrialization, it means there's a lack of security.

Lastly, the current deployment of the solution is not scalable, as it's was mostly developed as a PoC, so the there are no heavy tests for updating the system, pipelines for deployment, etc. So if there is a big update to the application, it would be more susceptible to have downtime due to bugs or other issues, that were not caught beforehand.

# Chapter 4

# Contributions

## 4.1 Business process

So for the business process, as stated previously, we have a set of activities that organizes the trucks and their drivers. In the goal of have their workflow optimized.

And it goes as follows:

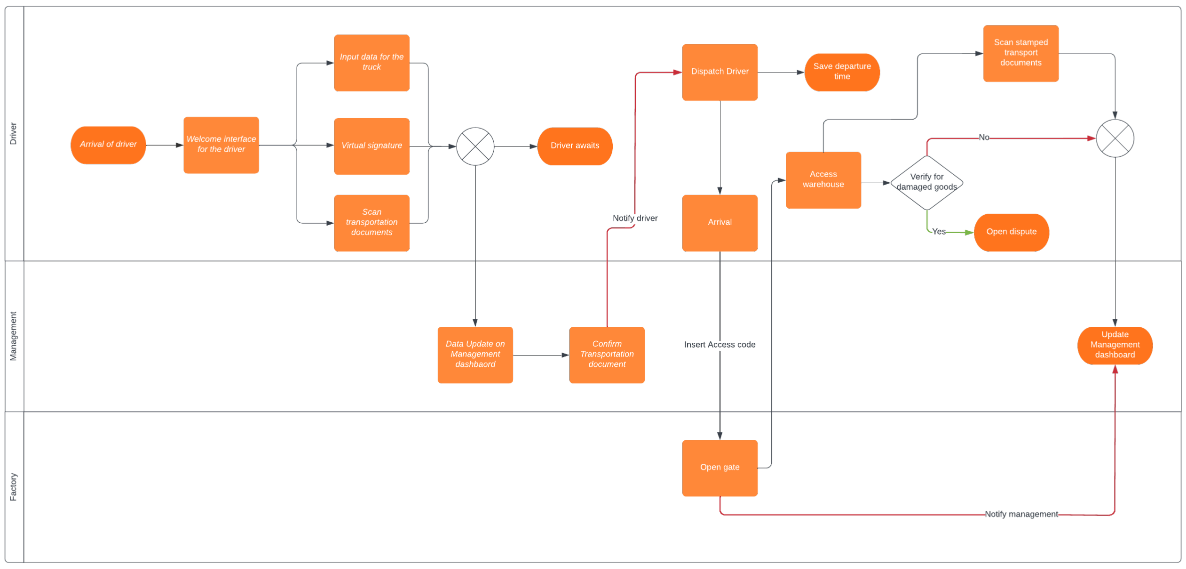


Figure 4.1: Business process flowchat

To give more context to the diagram above, we will dive into the details of it processes. So first of all we have the following core processes:

- Welcome of the driver
- Digitalization of documents
- Secure access to logistics platform
- Dashboard for the management

Starting by welcoming the driver, this process relies on the driver to introduce himself, his credentials, scan any documents related to the transportation and also have a signature of the driver for the legal purposes all that present at a kiosk, once all that data has been input the driver is invited to wait at the truck awaiting the response from the management.

Then there is the digitalization of the documents which is the process of the driver to have a digital copy of the documents that are all sent to the management dashboard for verification and approval, to give sign to the driver that the documents are correct and that he can proceed to the next step, sending the approval is done through sending a notification either through the phone application, an sms or calling by pager.

Next comes the secure access to the logistics platform, this process is mainly related to infrastructure gate, the driver has a special access code to the platform, this code is generated by the management and is sent to the driver by sms or through the application, once the driver accesses the platform the management is notified by his arrival.

Finally, the dashboard for the management is the main interface for the management and it's divided into multiple workspaces depending on the role and priveledges of the user depending on the needs of the clients.

As of the current time, the application is as follows:

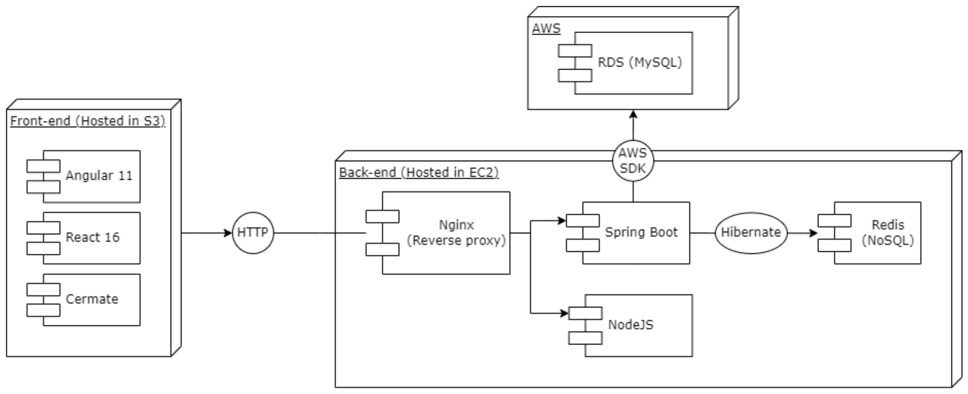


Figure 4.2: Current state of the application

So it's built with different technologies for the front depending on the platform or device that is handling it, and for the backend it's handled mainly in Spring and NodeJs, for the database it's handled for the realtime data with Redis, and for the user data it's handled in RDS MySQL database, for everything that is related to handling notifications it's done through the MQTT broker and Google Firebase service.

The Spring application is handling everything from the user data, managemenet of the sites, the trucks and drivers, the geolocation, the notification mapping, the realtime data handling.

## 4.2 Conceptual model

As of now the software is still in it very early stage, and being used mainly as a proof of concept. And it's following the architecture below:

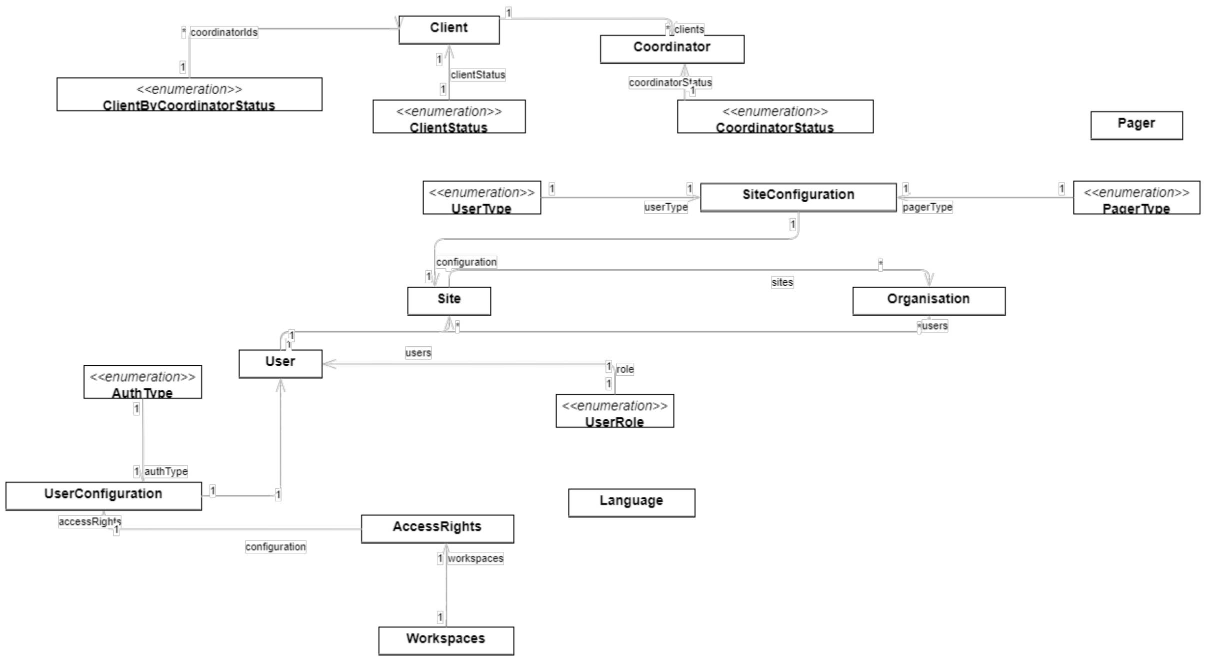


Figure 4.3: Backend model

## 4.3 Target

As of the current moment the forecast component diagram for the system is as follows:

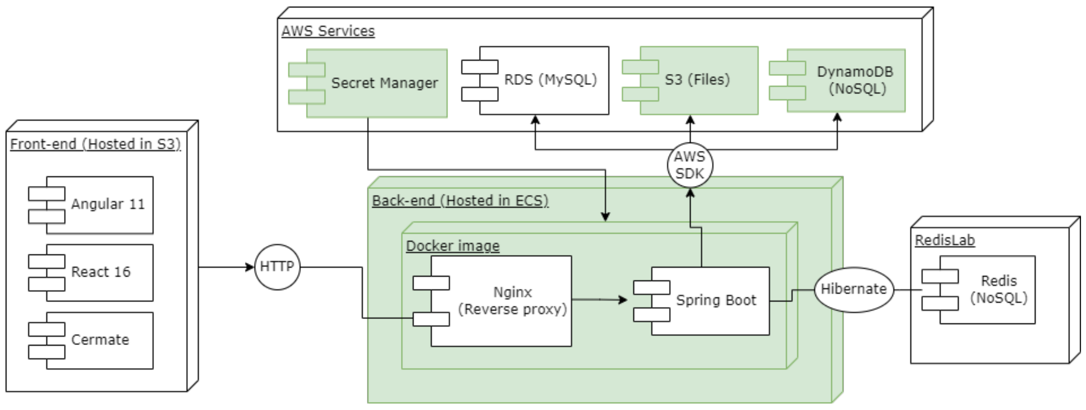


Figure 4.4: Forecast component diagram

### 4.3.1 Target

As it's shown in **??** there are additional databases that are waiting to get implemented mainly ones for archiving files as is the case of S3 and also archiving data for statistics and KPIs down the line thus using DynamoDB. Also containerizing the backend for the goal of having it run different instances seamlessly, spawn the run times duplicate efficiently to have better load handling and manage usage spikes. For the creation of the images, it should be noted that they will be generated automatically, aswell as their deployment. Finally comes adding new features to the backend, and refactoring the majority of the code to make it more readable, maintanable and optimized.

# Chapter 5

# Implementation

## 5.1 Databases

For the database we had two subjects to study:

- Databases for archival purposes
- Databases for storing files

### 5.1.1 Archival Database

So taking the first one, we had to study performance to cost of three solutions and their scalability:

- RDS (SQL)
- DynamoDB (NoSQL)
- Aurora (SQL)

As all three were AWS solutions, the performance of each was pretty much the same, and as the database was going to be used for archival it relied less on having a high throughput and more on how much storage costs. So we only relied on the cost of the solution, how easy is it to maintain.

So going off by cost first Aurora was the most expensive, with no value added to the problem we'll be solving with it.

So that left us with two options:

- RDS (SQL)
- DynamoDB (NoSQL)

We dived into the RDS solution, and it was a bit of a challenge to get it to scale, in terms of storage compared to DynamoDB which offered autoscaling options and variable throughput for high load time which could fit the needs of archival as it could take higher input in the daily backup process, than the rest of the day. And for the output there wasn't much to be gained, for neither of the solutions as it was a rare process that wouldn't be used alot only on demand by the user.

Going to studying the cost, the primary tool that was used for that is the AWS Cost Calculator, which gives rather good indicators to approximate the charges.

**Case study:**

For the data we used the following: - 200 Entries per site. - 500 Site. - 2 KB per entry.

Site represents the factories and the entries represents the trucks.

So with those inital values, we came to the following results: - Daily storage of 200 MB or monthly of 6 GB that increments after the period passes. - 3000000 writes / month - 100000 reads / month

the reads were just second guessed as there was no significant data about it, other than it will be a low read database.

So for the calculations, for RDS we used the following options: - Instance db.t3.large (vCPU: 2 Memory: 8 GB) - Instance reserver for one year - Single-AZ (1 node) - 6 GB storage incremented monthly - 6 GB backup incremented monthly

which resulted in the yearly cost of $1185.769

For DynamoDB, we used the following options: - Standard-Infrequent Access (Reduces cost of storage) - 3 Mil writes per month - 100K reads per month - 6 GB storage incremented monthly - 6 GB backup incremented monthly

Which came down to the cost of $236.00 per year.

As we can notice the RDS cost was much higher than DynamoDB, so we decided to use DynamoDB as it was cheaper, and also had better maintenance options offered by AWS as it's a fully hosted service.

Lastly, the only missing part was integration within the back-end which went on different steps.

First we had to create a service on the Spring Back-end that took on the handling of the communication layer between the DB and the back taking on mulitple functionalities such as:

- Creating - delete tables
- Creating indexes to allow fast access to the data
- Adding - deletting data
- Finding data using specific fields

Then implementation of unit tests for the service to ensure that it's working as intended, it had taken on the following tests:

- Create table for tests with required indexes
- Adding data to table, and then finding it
- Adding data to table, and find it by date
- Adding data to table, and find it by different keys
- Cleaning up the table

Finally the creation of the tables and their indexes was automated, to allow for the reproduction of the same environment for the data we had during testing, and in the dev environment.

Final the archival process was as follows:

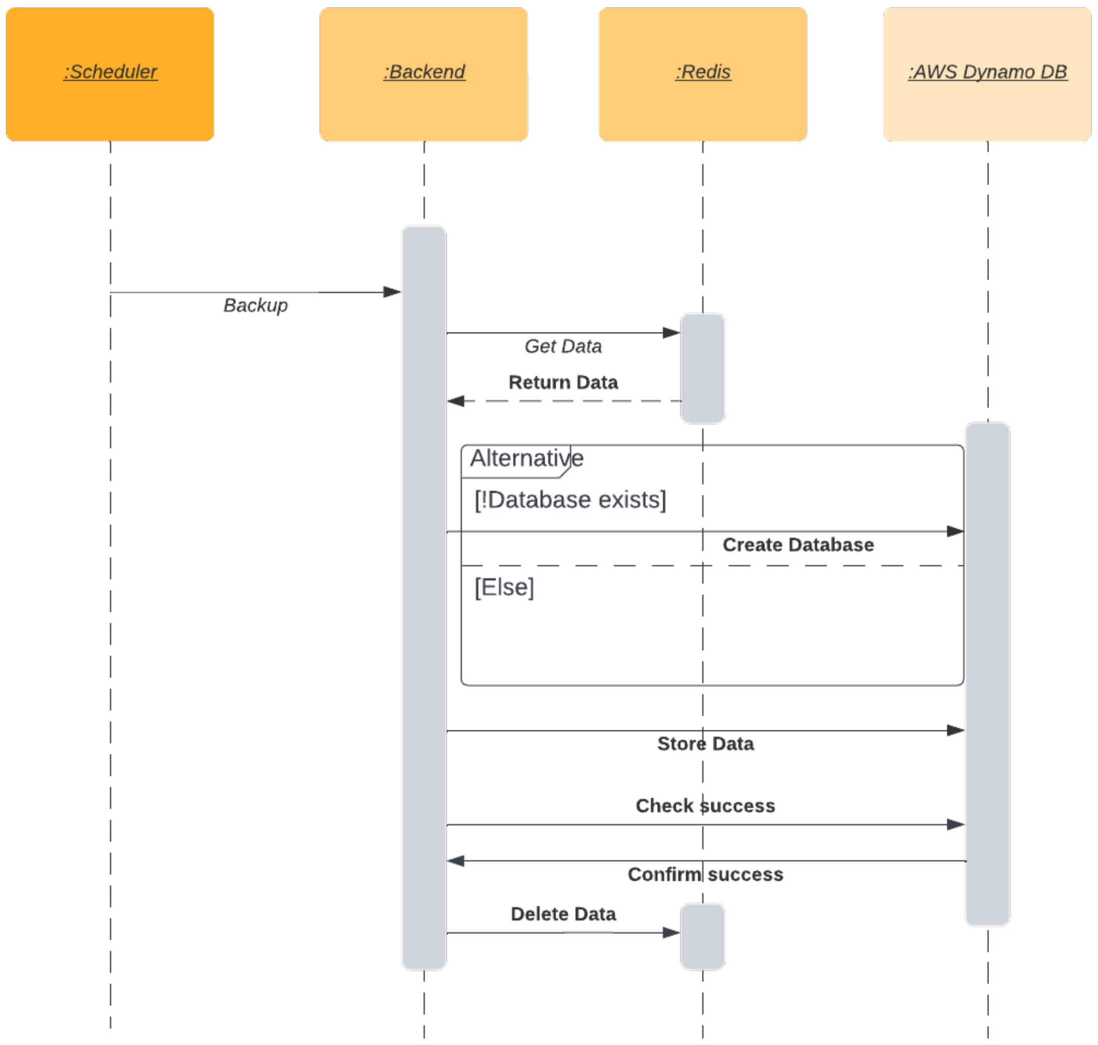


Figure 5.1: Archival Cronjob

The cronjob for handling the archival had a pretty simple process, it would take the data each day at 00:00 and archive it from the redis database to the DynamoDB while checking that everything was successfully archived before cleaning the data from Redis.

### 5.1.2 File storage Database

For this second database, that is more opted to be used for the storage of files. The solution was already known, and was the Amazon S3 service.

The file storage database was used for the following:

- Storing site configuration files
- Storing client files

The storage was pretty simple, it was just a bucket with a specific folder structure. So for the first case it was a folder for each site, and for the second it was a folder for each client which was handled with the archival cronjob as these files were related to the data from the trucks that have been handled during the day.

For the tests we used the following:

- Creating a bucket
- Creating a folder structure
- Adding data to the bucket and finding it
- Deleting data from the bucket
- Deleting the bucket

For the creation of the bucket it was made such as it's done automatically once the server started in case it doesn't exist.

So that we can reproduce the same bucket that there was in the dev environment and during the tests.

## 5.2 Security and Authentication

### 5.2.1 Ratelimiting

Ratelimiting was one interesting aspect for security, as it's a way to prevent the server from being flooded with requests. As it could stop DOS attacks, brute force attacks, and any other kind of attack that would require sending lot of request from the same IP or same user.

For the ratelimiting, as we were using already a Spring backend we had a couple options to choose from the first one was Bucket4j, which is a ratelimiter that relies on having a bucket for each key, the key could be a user, IP address, APi key ...

The second option was to use Resilience4j, which relied on a circuit breaker to handle the ratelimiting but it was not easy to implement for the case of a distributed backend, as it relied heavily on single machines threads and not clusters.

So we went with the former, which was implemented as a filter in front of the API gateway in the Spring backend.

As seen in the figure , that's the basic handling of services within the Java Spring backend, so the ratelimiting was implemented in the API gateway such as to stop any coming requests before they can have access to any service.

The ratelimiting was handled in two ways, the first was to have a bucket for each user, and the second was to have a bucket for each IP address.

Ratelimiting the user was specifically setup for handling the private endpoints and that was able to be modified by the admin to allow for more or less requests per second depending on the user, and the ratelimiting the IP address was setup for the public API.

And lastly there was the ratelimiting for the login, which was individually handled for the purpose of limiting the number of login attempts per minute to avoid possibility of a dictionnary attack.

### 5.2.2 Authentication

The authentication was also one of the interesting aspects for security, as it's a way to prevent confidential data from being accessed without having the right credentials. But the prior used solution was having a permanent JWT token that is acquired during the login, meaning that once the that Token is generated it could be permanently used to access the data as it didn't use any additional timing parameter to expire the token.

As for that, came the need to implement a ephemeral token, which is a token that gets expired after a certain time, to access the data and opted to add a refreshing mechanism which was to make the user experience better to avoid the need of having them reconnect within their sessions.

But not only that, we also opted to move from a Bearer token transportation of the token to a HTTP Only cookie, which is a cookie that is only accessible through the HTTP protocol, not the Javascript and is proclaimed to be the safest option to store the JWT. And then proceeded to embed the token within the cookie, as so we can avoid the possiblity of exploitation through JS injections to acquire the token.

The flow of the authentication then came as follow:

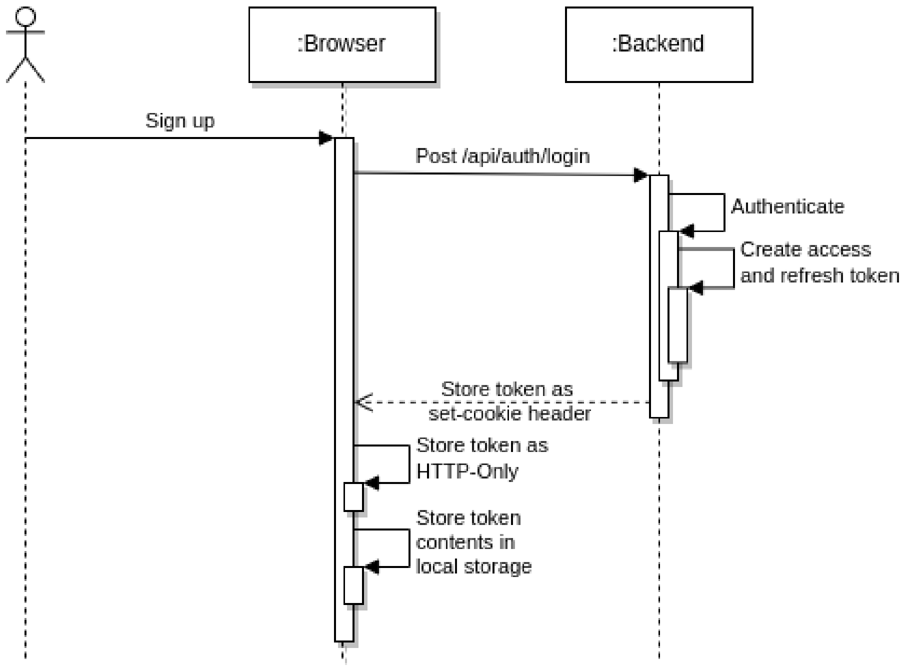

:Browser
:Backend
Sign up
Post /api/auth/login
Authenticate
Create access and refresh token
Store token as set-cookie header
Store token as HTTP-Only
Store token contents in local storage

Figure 5.2: JWT HTTP Only Cookie

The refresh token flow is as follow:

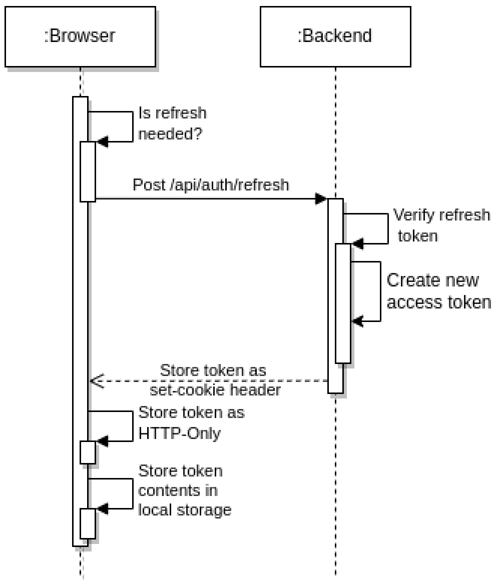


Figure 5.3: Refresh token

## 5.3 Containerization and Deployment

Before we dive in into the specifics, we first will showcase the final architecture in the ECS, which is below:

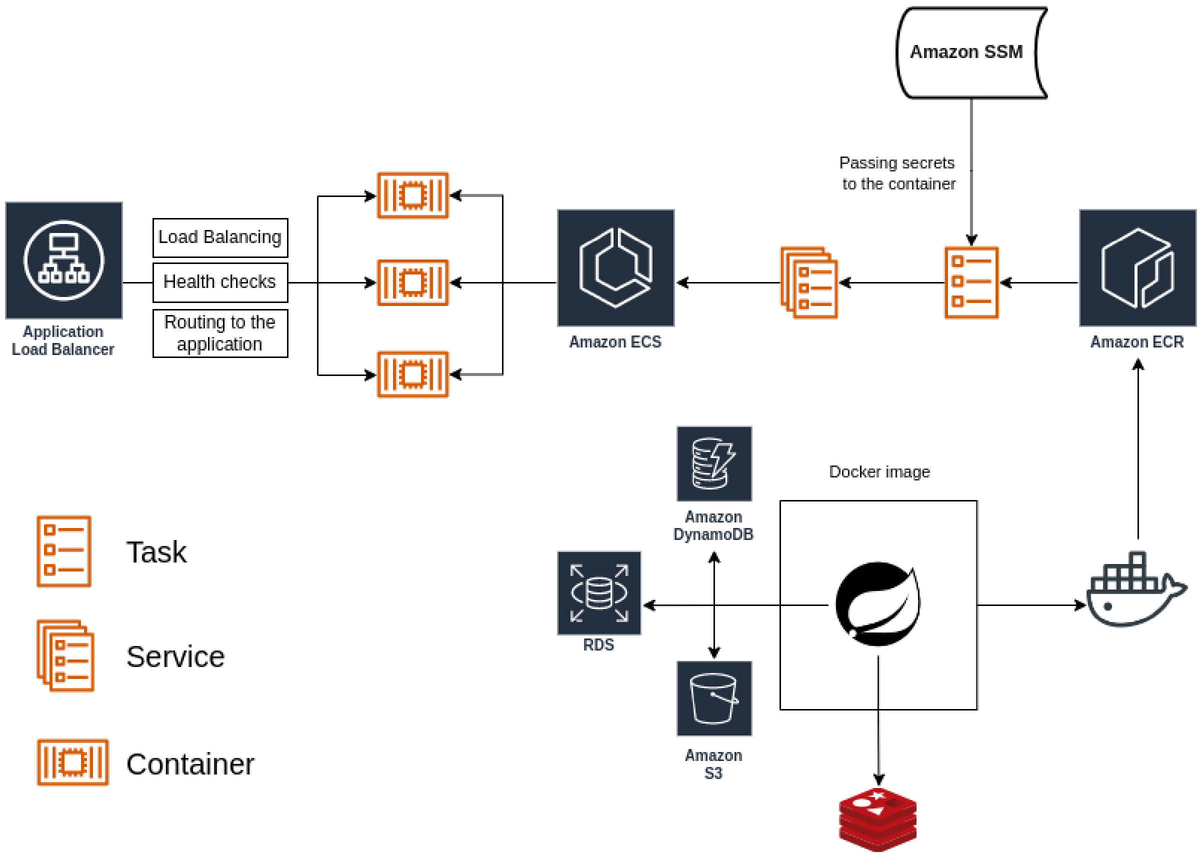


Figure 5.4: ECS Architecture

Task: the component which takes care of running the containers.
Service: the component which manages the tasks.
Container: the image that is running on the task.

### 5.3.1 ECS Containerization and secrets

The ECS solution was a bit of a challenge to get it to work, as it was a containerized solution, and not a native one. So the first step was to containerize the solution, and have a safe image that doesn't expose any secrets or environment variables.

In the docker way, we can just pass them through the command line, and the solution will run perfectly.

But here relied a problem that was the ECS solution, it didn't have a simple way to feed it the credentials, environment variables neither files, so we had to opt for using the SSM or Secrets Manager which is yet another hosted solution to store strings as secrets, and then those could be passed to the ECS services.

With the Secrets Manager, for the strings secrets it was a simple writing / reading process to store and retrieve the secrets. But for the files it was a bit more complicated, there wasn't a direct way to do that.

Finally we came up with the following solution, which is as follows:

- Serialize the content of the file into a base64 string

- Store the base64 string in the SSM
- Pass the SSM secret to the ECS service
- Retrieve the secret from the SSM as a environment variable
- Decode the base64 string
- Store the deserialized base64 in a file in the container
- Point an environment variable to the file

As such we managed to pass aws credentials, google credentials and Redis Labs credentials.

### 5.3.2 Load Balancer and routing

As shown in the ECS Architecture, we had a load balancer in front of the running tasks, which has four main goals:

- To take care of the incoming traffic from the client side
- To ensure that the traffic distributed evenly
- To ensure that the tasks are healthy
- To ensure that the tasks are not overloaded and manage it's scalability.

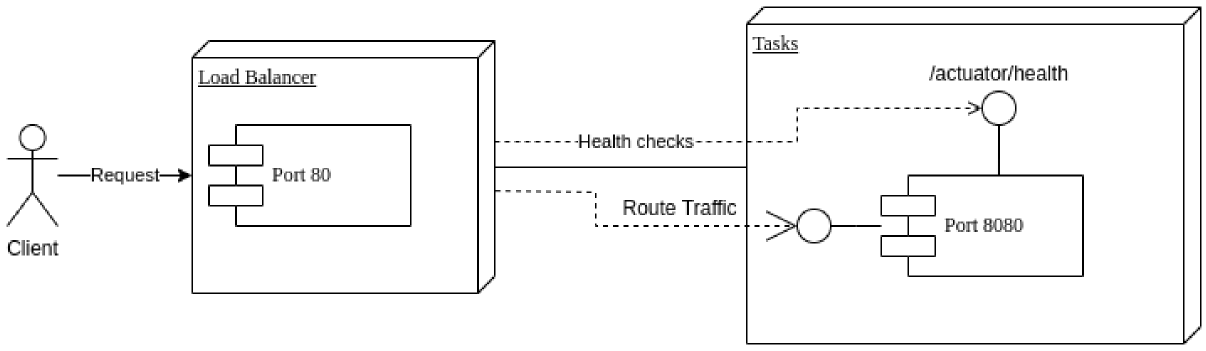


Figure 5.5: Load Balancer Interaction with the backend

So the load balancer is as shown in figure 5.4, it took traffic from port 80 where it had an exposed dns link, then it routed it through a VPS to the containers that are exposing post 8080 in their local network, having healthchecks done on the endpoint /actuator/health .

With that set in place, we had pretty much everything setup to start testing for the load handling by the ecs to check for the health of the containers and the measures to be set to ensure that the auto scaling is setup with the right parameters.

So first, we written scripts that did kind of overload the server with the requests, asynchronously deploy the containers and check monitoring metrics.

The script worked as follows:

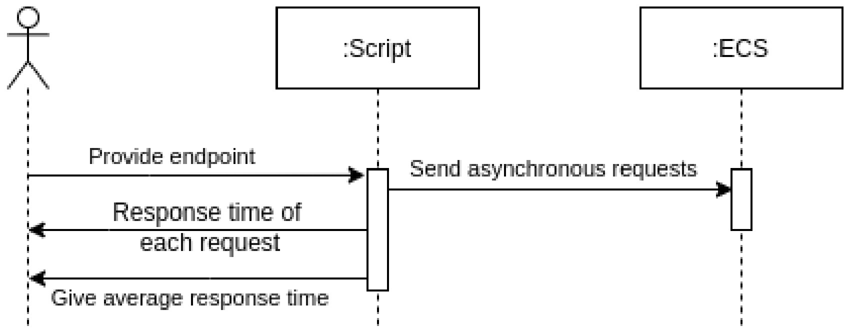

:Script
:ECS
Provide endpoint
Send asynchronous requests
Response time of each request
Give average response time

Figure 5.6: Load Test Script

So firstly it took the endpoint to be hit, in case it was a private endpoint you'd need to provide the credentials to. Then put in place the number of requests to be sent and finally run it.

Once it started running it was a simple case of sending requests asynchronously in fibers or virtual threads, and then printing the response time of each request, as such we could monitor the handling time of the requests being affected by the load of requests.

Thus it we could find the right parameters for the auto scaling, based on the number of requests being sent per second, which at the time of tests was around 1000 request per second without the response time being affected.

And as most the back-end services relying heavily on the memory, we ensured to put a 70% memory usage limit on the containers before spawning new instances to handle the load, in case of autoscaling by memory.

### 5.3.3 Deployment

Finally after testing everything, and having deployed one ECS service succesfully, we had to automate the deployment process, so that we could handle new versions and code update without much effort.

This came in the form of a gradle task, which was as follows:

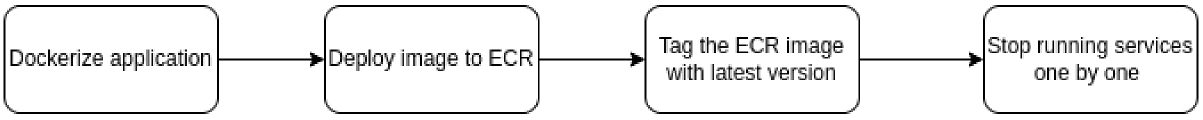

Dockerize application
Deploy image to ECR
Tag the ECR image with latest version
Stop running services one by one

Figure 5.7: Gradle Task

The dockerization part of the gradle task was handled using a third party plugin, then the deployment to ECR is done using the AWS CLI, by linking the ECR to docker, then pushing the image is automatically uploaded to ECR. Once this is done, the ECS service has to handle the update of the image, and this was done through a bash script that took in charge of getting a list of the running tasks, then stopping one of them, and wait for a new one to spawn such as to avoid down time during the update of the service.

For clearer representation of the flow of this task, here's a sequence diagram:

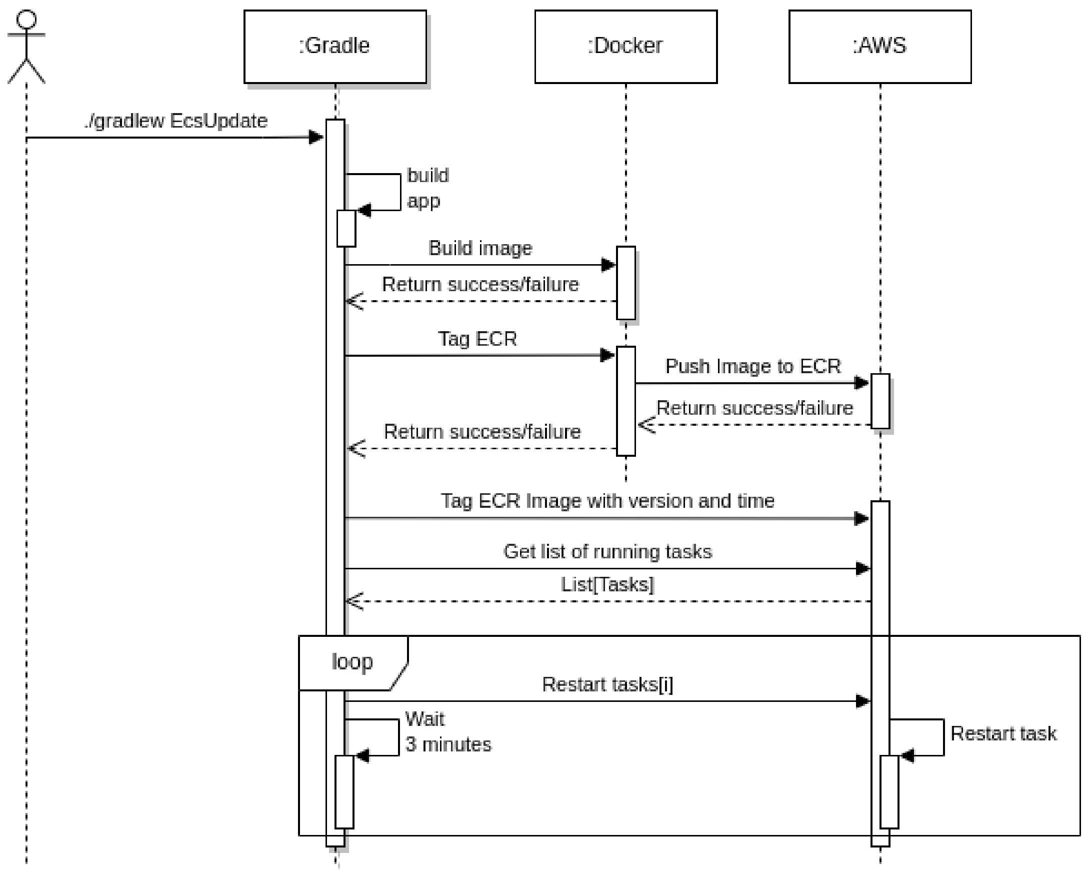


Figure 5.8: Deployment Sequence Diagram

## 5.4 Refactoring and Improvements

### 5.4.1 Adding user management system

The old system didn't have a user management system, as it relied on the site being the user so there wasn't really users to manage. So we had to integrate users within the system then add a way to manage them, and this was done by migrating the site from being the user to being an entity that could be managed by users who have the right to do so. And that required rework through the majority of the backend as this architecture was relied on by all the backend services and controller and also frontend.

So we had to change this architecture to be able to manage users and their roles, but also keep the old behaviour the same.

### 5.4.2 Implementation of tests for existing codebase

TODO: Still not touched on yet

# Chapter 6

# Deployment

# Chapter 7

# Conclusion