

Sumário

Projeto Nêmeses – Rafael da Rocha Ferreira	1
1) Introdução	1
2) O Broker MQTT	3
3) Parser.....	3
4) DBexport.....	4
5) RedisDB e MySQL	5
6) Backend Server	6
7) Deploy.....	8
8) Organização Empresarial	9
9) Melhorias Propostas e Finalização	10

1) Introdução

O documento a seguir tem como função apresentar as diretrizes tomadas e os elementos selecionados para se fazer o projeto proposto inicial. Deve-se ressaltar que o nome ‘Nêmeses’ se deu para que haja maior facilidade ao se referenciar os programas necessários e para homenagear o vilão de um jogo presente na minha infância infância 😊

Inicialmente, deve ser esclarecido que todas as decisões tomadas e as tecnologias utilizadas tem como premissa o meu próprio julgamento pessoal. Dessa forma ficará evidente, de modo honesto, meu trabalho de projeção e arquitetura de backend.

a) Conceitos iniciais e reflexões

A cartilha apresentada em PDF propõe a formação de um modelo backend para acesso às informações recebidas por um sensor. Além disso, tal modelo deve ser escalonável e priorizar a amostragem da última informação recebida apresentando os valores de posição do sensor em tempo real.

Tendo isso em mente, nota-se o comportamento IoT do sensoriamento baseado em telemetria de modo a definir o projeto em si como tendendo à modelos de IoT. Logo, a presença de um gateway que apresente as mensagens de forma distribuída é necessário e, por consequente, um modelo de parseamento de dados binários e importação em banco de dados também serão requeridos.

Além disso, nota-se que o projeto requer um fornecimento de respostas o mais próximo de tempo real *Real Time (RT)*. Dessa forma os armazenamentos e coletas de informações (assim como o seu trânsito) devem ser planejados a fim de gastar o menor tempo o possível requerendo, assim, modelos de cache e armazenamentos indexados para facilitar a busca.

Por fim, de acordo com as diretrizes da cartilha apresentada, a requisição e envio de mensagens provenientes de um usuário deve ser através do protocolo HTTP, logo é necessário a utilização de uma linguagem de programação que possua facilidade na interação com tal protocolo além de possuir boa performance na comunicação com sistemas de armazenamentos e bancos de dados.

Vale a pena ressaltar que, como todo o trabalho de programação está sendo realizado de forma pessoal e há certo desconhecimento a respeito do nível de conhecimento do avaliador/usuário, é necessário que todos os programas sejam realizados de forma clara com comentários pertinentes, com poucas dependências e os mais genéricos o possível. Dessa forma, terá uma maior acessibilidade e compreensão pelos leitores. Para a tarefa, a programação funcional será mais explorada (podendo ter cruzamento com demais paradigmas, entretanto privando a orientação a objeto ao máximo)

b) Arquitetura proposta

De acordo com as necessidades aparentadas no item anterior, a arquitetura proposta é apresentada na figura abaixo:

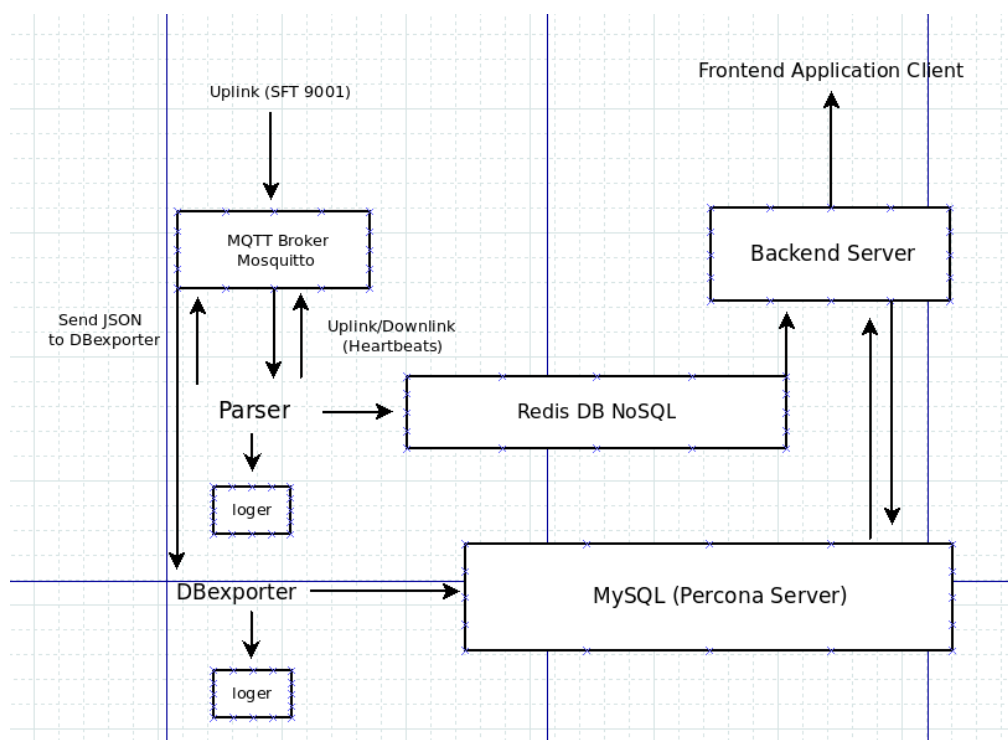


Figura1. Arquitetura proposta para projeto

Trata-se de um sistema de mensageria Broker MQTT assinado por um parser e um exportador de dados para um banco relacional. Nota-se também a presença de dois banco de dados Redis e Mysql que trocam informações tanto com o parser e com o exportador quanto ao backend que, por sua vez, serve dados ao usuário por meio de um modelo REST.

Tal modelo apresentado é composto por 2 linguagens de programação requisitadas: Python e JavaScript (aplicada em NodeJS), sendo que o parser e o exportador são formados à base de Python enquanto o backend é formado por JavaScript em NodeJs com definição das rotas utilizando a framework Express. (Mais adiante será explicado o porquê)

A proposta de tal modelo de arquitetura foi baseada em escalabilidade, separamento de responsabilidades, facilidade de ação para equipes técnicas e melhor deploy em nuvem (Também serão explicados mais tarde)

Em seguida, será apresentado de modo direto e rápido como funciona o projeto e o motivo de se utilizar tais tecnologias.

2) O Broker MQTT

A cartilha do desafio proposto inicia apresentando o rápido conceito de gateway, entretanto para que fosse possível a simulação real do sistema e, por consequente, a projeção de uma arquitetura mais próxima do real, foi necessário a utilização de um Broker de mensageria. Para esse caso, se considerou mais adequado a utilização de um Mosquitto Broker que utiliza o protocolo MQTT

O MQTT (*message queue telemetry transport protocol*) é um protocolo leve mais dedicado a internet das coisas IoT. Possui uma grande simplicidade e um baixo custo computacional, sendo mais adequado que outros protocolos como AMQP ou STOMP. Além disso, trata-se de um protocolo mais interessante para o uso em aplicações de Telemetria.

Para o MQTT utilizou-se Mosquitto Broker pela sua simplicidade. Ao contrário do RabbitMQ, o Mosquitto Broker apresenta apenas interface com MQTT e portanto rejeita redundância de funcionalidades indesejadas.

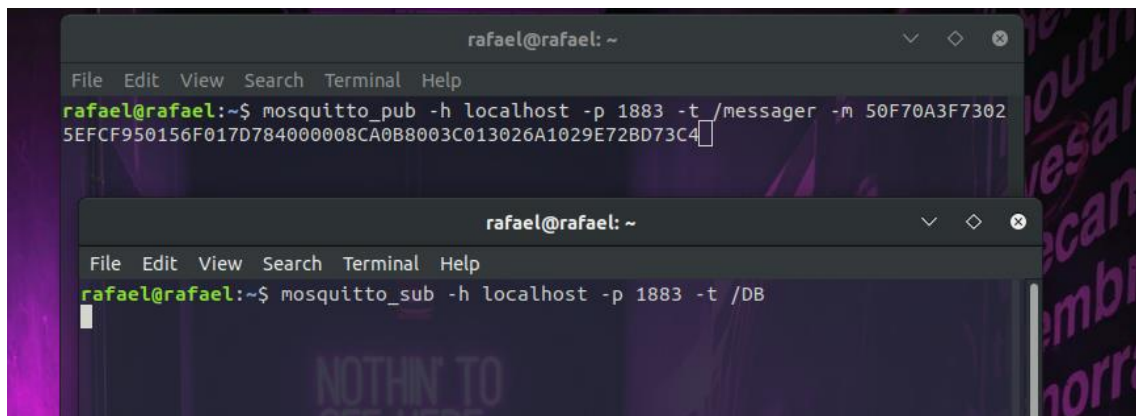
Foram utilizadas 3 tópicos para o projeto:

/messenger - Tópico a qual o Parser recebe as mensagens dos dispositivos

/ping - Tópico a qual o Parser retorna a mensagem de Ping para o dispositivo

/DB - Tópico de comunicação Parser - DBexport

A imagem abaixo apresenta de modo resumido os comandos de Pub/Sub no Mosquitto:



```
rafael@rafael: ~  
File Edit View Search Terminal Help  
rafael@rafael:~$ mosquitto_pub -h localhost -p 1883 -t /messenger -m 50F70A3F73025EFCF950156F017D784000008CA0B8003C013026A1029E72BD73C4  
rafael@rafael: ~  
File Edit View Search Terminal Help  
rafael@rafael:~$ mosquitto_sub -h localhost -p 1883 -t /DB
```

Figura2. Exemplos de Pub Sub em Mosquitto Broker

3) Parser

A necessidade de um programa de parseamento de informação se dá pelo simples fato da mensagem recebida ser apresentada em binário. Para isso, será necessário um padrão de fatiamento da mensagem e tradução/descompressão. Portanto, o programa segue a seguinte máquina de estados:

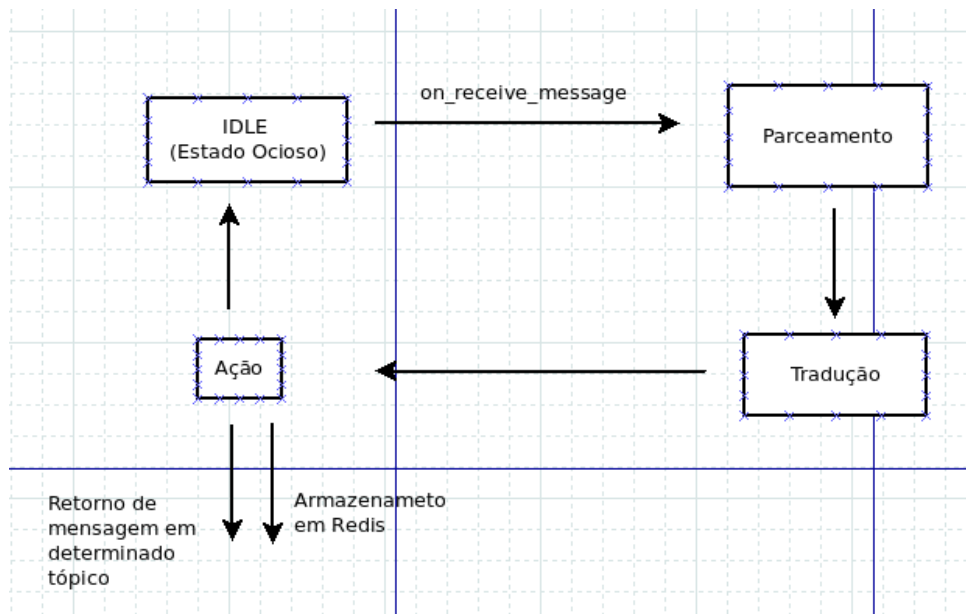


Figura3. Esquematização do Programa Parser

A utilização da linguagem python se deu por alguns motivos especiais que serão retratados agora:

- **Apresenta uma interface amigável para tratamento de informações binárias** - Python possui o bytearray como estrutura interna podendo, assim tratar informações binárias com facilidade.
- **Linguagem indicada para paradigmas de programação funcional e procedural** - Python é multiparadigma, entretanto sua máquina interna (PVM) trata os comandos apresentado de forma sequencial. Além disso, seus módulos de importação favorecem a aplicação funcional. Tais características são desejadas para o processo de parseamento (máquina de estados).
- **É bastante popular e de fácil entendimento para programadores Não-WEB** - A comunidade em python é bastante vasta se expandindo até para programadores não focados em WEB. Logo, o entendimento e a manutenção do Parser será mais facilitado.

Obs: Devido ao curto período de tempo, os Logs do Parser e do DBexport não foram gerados, entretanto é necessário uma nota afirmando a grande relevância de tais elementos. Além disso, vale a pena mencionar que apenas os dados Não históricos (Live) são enviados para o banco RedisDB

4) DBexport

O DBexport segue a seguinte máquina de estados:

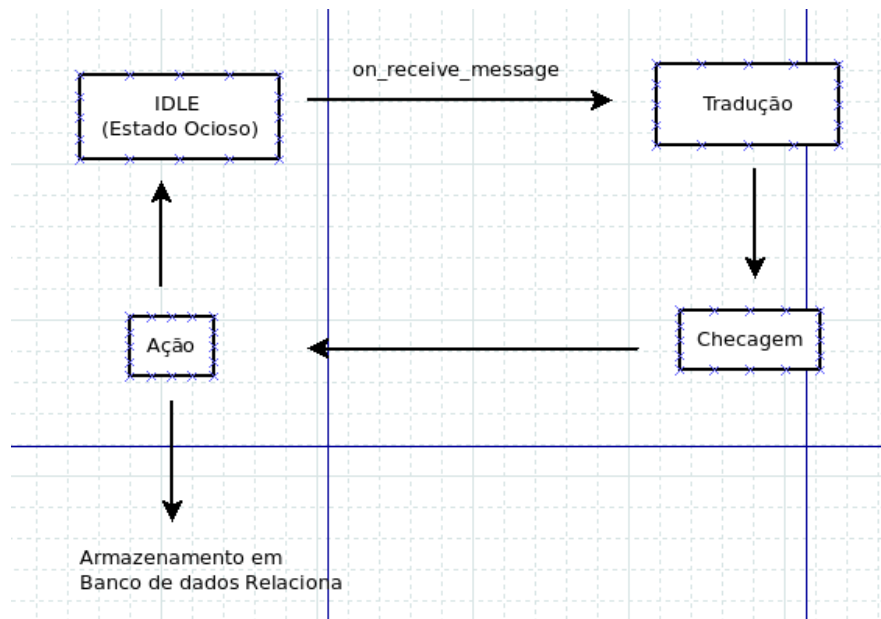


Figura4. Esquematização do Programa DBexport

Os motivos para ser projetado em python segue os mesmos princípios que o Parser. Entretanto, vale a pena ressaltar o motivo de sua separação do Parser (Porque o Parser e o DBexport não se encontram em um único programa?):

Separando ambos os programas tem-se a possibilidade de um escalonamento horizontal, de modo a colocar mais exportadores em paralelo caso necessário ou mais passeadores em paralelo dependendo da demanda dos dados.

obs: Para tal caso, foi utilizado o banco de dados sqlite a título de apresentação pois a instalação de um banco MySQL em servidor percona é um pouco trabalhoso (criação de usuario, dependencias, inserção de usuario e senha...)

5) RedisDB e MySQL

Um dos pontos de interesse é entender o motivo da utilização de 2 bancos: Redis DB será utilizado para Cach enquanto MySQL será utilizado para armazenamento de informação e serviços secundários. Vamos por partes....:

RedisDB se trata do banco NoSQL mais performaticamente rápido em transactions e comumente utilizado no mercado. Seu *sweetspot* baseia-se em seu dataset inteiro ser suportado em RAM de modo a fazer operações de I/O rápidas. Além disso, possui um gerenciamento de memória capaz de produzir dados com TTL (*Time To Live*) ou seja, dados que são apagados após certo tempo em segundos.

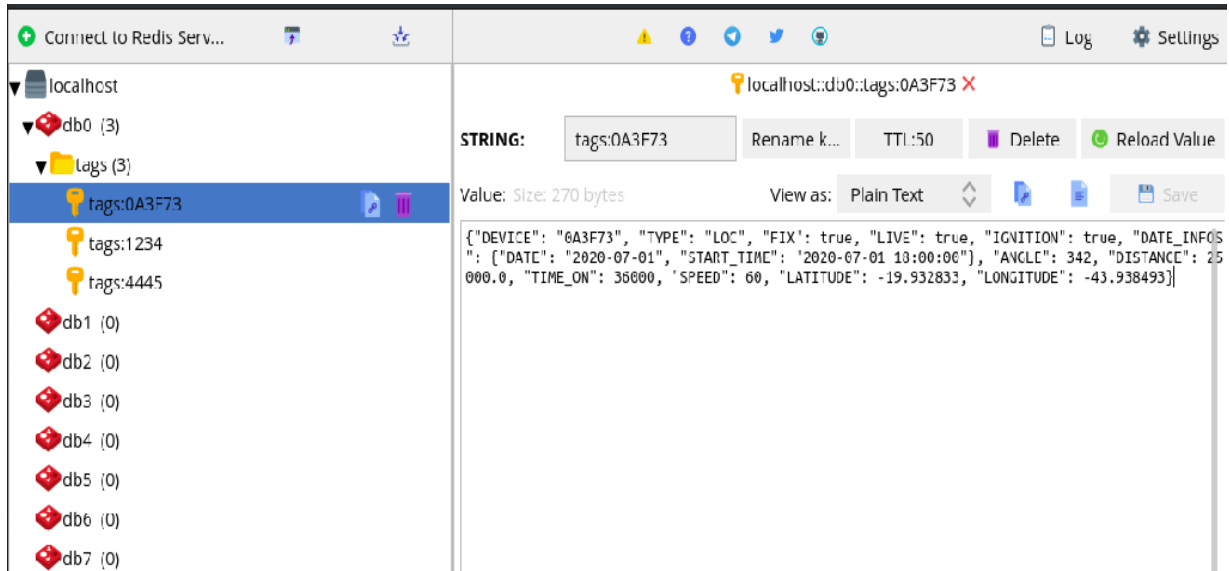


Figura5. Banco RedisDB a partir do programa Redis Desktop Manager

A utilização de um banco RedisDB fará com que a consulta da última mensagem apresentada pelo dispositivo seja apresentada de forma bastante rápida e sem fazer grandes buscas passando por muitos registros. Logo, servirá como um cache de dados com o último dado se sobrescrevendo a medida que chega e reinicia o TTL. Caso não chegue mais dados e o TTL termine, pode-se considerar uma perda de sinal do dispositivo.

A utilização do banco **MySQL** se dá unicamente pelo ser fork Percona Server (software livre) que possui a engine TokuDB. TokuDB se baseia na idéia de indexação com *Fractal B-Tree* e consegue armazenar grande volume de dados (próprio para Big Data) apresentando inserts e selects mais rápidos mesmo com vários registros presentes (Tendo uma boa indexação, claro). TokuDB possui arquitetura ACID, transacional e MVCC.

Logo, um banco **Percona MySQL** será necessário para geração de relatórios, apresentação de informações históricas, registros passados e outros.

6) Backend Server

Para a pojeção do servidor, desejava-se que o I/O de dados através do protocolo HTTP fosse realizado de maneira rápida e simples. A partir disso, a utilização de JavaScript é desejável pela utilização de evento assíncronos (proporcionados por um paradigma de programação orientada a eventos), simplicidade na sintaxe e popularidade com vasta comunidade. Mais precisamente, foi utilizado o framework Express para a projeção de rotas.

O servidor apresenta as seguintes rotas para utilização:

- 127.0.0.1:5000/api/v1/location/?device_id=###
- 127.0.0.1:5000/api/v1/report/?device_id=###
- 127.0.0.1:5000/login
- 127.0.0.1:5000/register/user

Para interagir com o sistema, basta seguir os seguintes passos:

- Registro

Na rota /register/user fazer o registro de um usuário padrão

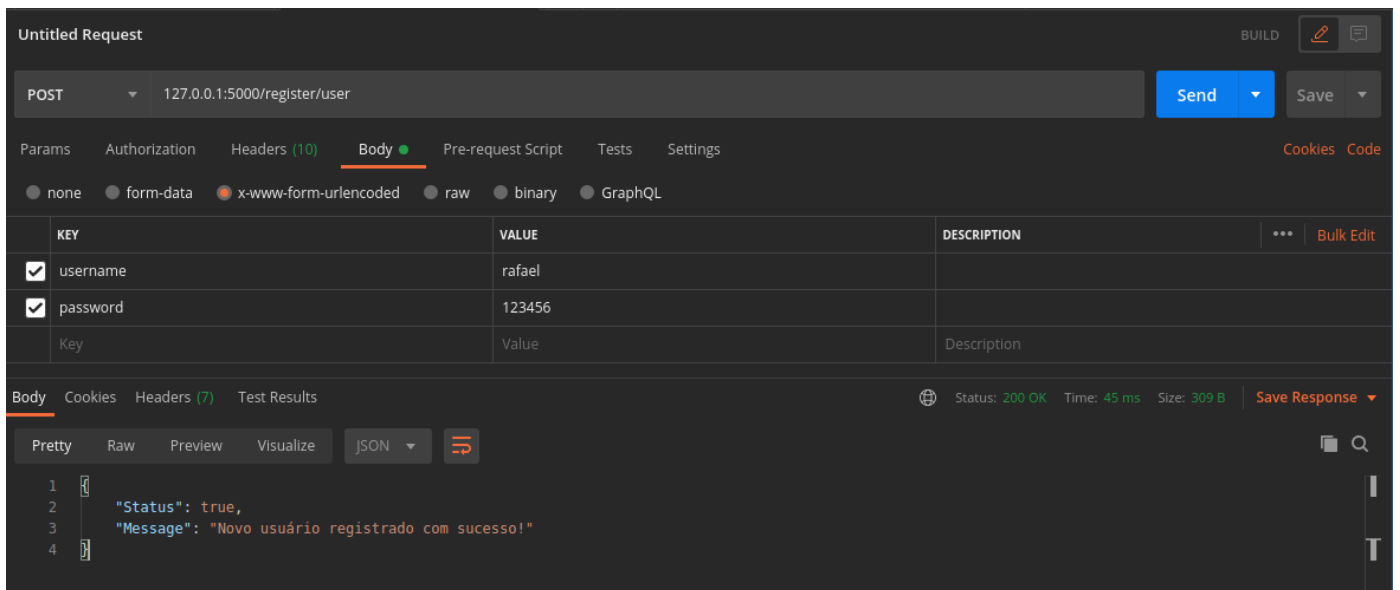


Figura6. Cadastro de usuario

- Login:

No login, será dado um Token JWT de acesso e o número de ID do usuario. Armazenar ambos:

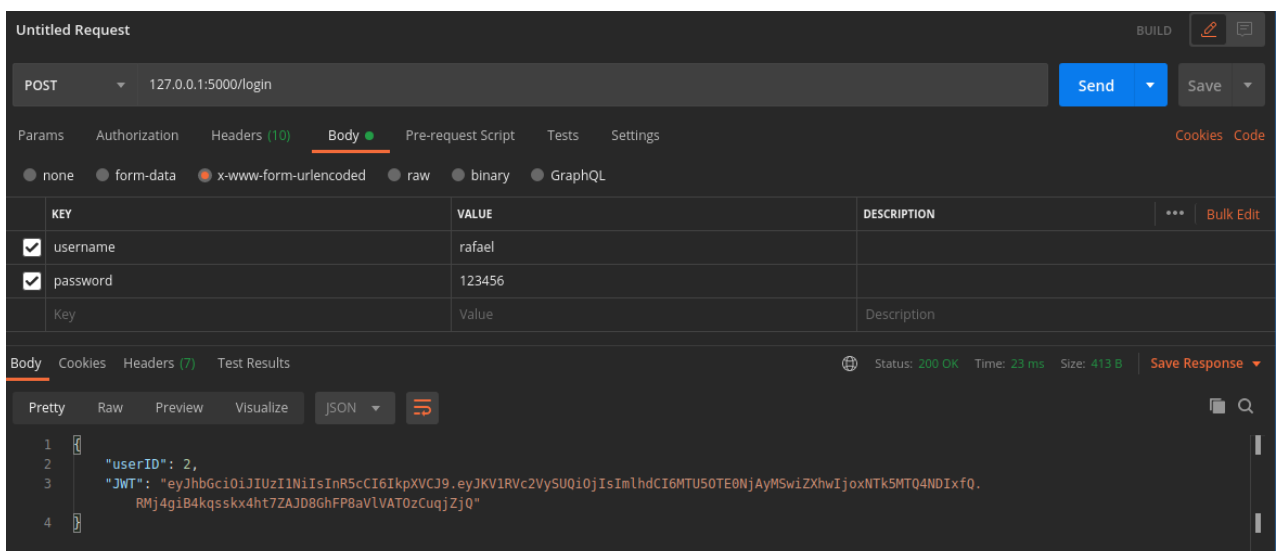


Figura7. Login do usuario

- Utilização das demais rotas com autenticação:

Basta colocar o Token e o id do usuário no header da requisição e acessar as outras rota desejadas

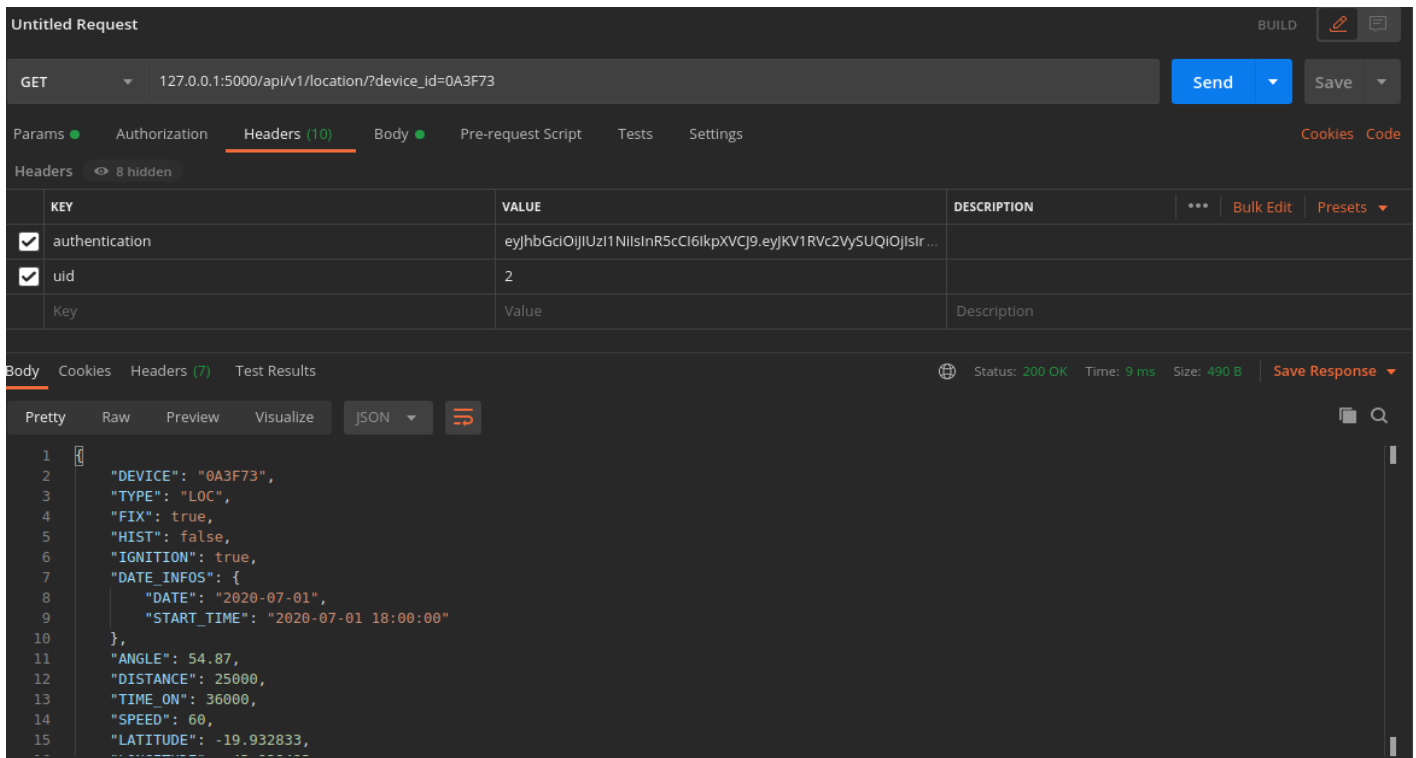


Figura8. Coleta de dados

Deve-se ressaltar que, caso o TTL do dispositivo no banco RedisDB expire, outra mensagem informativa será apresentada a respeito de indisponibilidade offline

Alem disso, é necessário dizer que a rota 127.0.0.1:5000/api/v1/location/?device_id=#### se trata de informações de real time presentes no RedisDB enquanto a rota 127.0.0.1:5000/api/v1/report/?device_id=#### se trata de informações armazenadas em banco relacional.

Por questão de tempo, não foi realizado a projeção de um modelo de autorização de acesso à certas informações de determinados dispositivos, entretanto seria implementado a partir da criação de um novo Token cujo payload tivesse exatamente o ID dos dispositivos permitidos. Também não foi criado um modelo de *Refresh* do token

7) Deploy

Por questão de Know How, o deploy seria feito utilizando o serviço de nuvem da AWS seguido pelo seguinte esquema:

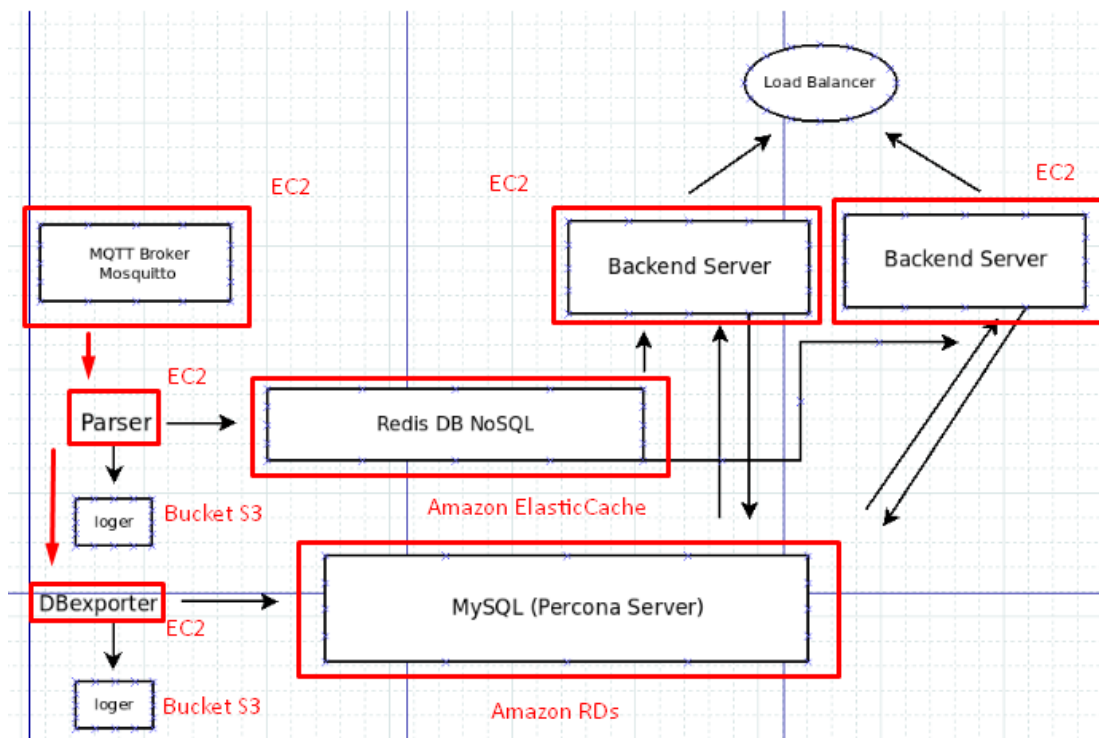


Figura9. Esquematização do Deploy em AWS

Nota-se uma duplicação do Backend Server justamente para que se tenha um balanceamento de usuários acessando o serviço proveniente do Load Balance. Além disso, nota-se a utilização de EC2 ao invés de Containers em Docker que, por consequência, também se dá por Know How já que pessoalmente possuo pouca prática e conhecimento sobre o assunto ☺

8) Organização Empresarial

Caso o projeto cresça e tenha um público mais abrangente, a própria arquitetura é amigável a presença de diferentes equipes de demais departamentos:

- **Suporte Técnico (e/ou N1):** Ficará responsável pela verificação de problemas de usuários em frontend assim como a verificação dos Loggers do Parser e do DBexport. Além disso, fará consultas e trabalhos limitados no Banco Relacional.
- **Project Setup (e/ou Manutenção ou N2):** Ficará responsável pelas configurações necessárias no Broker e pela manutenção no código dos programas Parser e DBexport. Terá acesso ao Banco RedisDB e fará o versionamento de novos programas
- **Developers (Frontend / Backend):** Ficarão responsáveis pelas devidas áreas de atuação Frontend e Backend Server de modo a escalonar os serviços propostos, versionamento de serviço e melhorias desejadas.

- **Database Administrator (DBA):** Manutenção do banco de dados relacional (Tunning, partitions, indexação, performance) e do cache RedisDB. Estará em constante contato com as equipes de Project Setup e Suporte Técnico

9) Melhorias Propostas e Finalização

Para melhorias propostas, foram pensados os seguintes tópicos:

- Utilização de Containers em Docker para se ter uma maior abstração na substituição de VMs em EC2
- Trabalhar com redundâncias em bancos de dados de tal modo a possuir mais de um ponto de acesso. Dessa forma, caso um banco se indisponibilize o outro ainda suportará o sistema.
- Criação de um outro programa para deletar os registros muito antigos em banco de dados relacional, de tal forma a não necessitar mais de tais dados.
- Geração de Views materializadas em banco de dados relacional de modo a capturar dados do dia presente e fornecer informações de forma mais rápida e precisa.

Thats All folks...