UNIVERSIDADE FEDERAL DE PERNAMBUCO
DEPARTAMENTO DE ENGENHARIA ELETRÔNICA E SISTEMAS
CURSO DE ENGENHARIA ELETRÔNICA

Breno Pedro Assis Tenório Pinto

Desenvolvimento de API REST e interface gráfica em Java para o Asa Branca

Recife
2023

Breno Pedro Assis Tenório Pinto

Desenvolvimento de API REST e interface gráfica em Java para o Asa Branca

Trabalho de Conclusão do Curso de Graduação em Engenharia Eletrônica do Centro de Tecnologia e Geociência da Universidade Federal de Pernambuco como requisito para a obtenção do título de Bacharel em Engenharia Eletrônica.
Orientador: Sidney Lima

Recife

2023

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Assis Tenório Pinto, Breno Pedro.
Desenvolvimento de API REST e interface gráfica em Java para o Asa Branca / Breno Pedro Assis Tenório Pinto. - Recife, 2023.
54 : il., tab.

Orientador(a): Sidney Lima
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Tecnologia e Geociências, Engenharia Eletrônica - Bacharelado, 2023.

1. API REST. 2. Java. 3. Asa Branca. 4. Segurança de API. 5. JSON. I. Lima, Sidney. (Orientação). II. Título.

620 CDD (22.ed.)

AGRADECIMENTOS

Gostaria de agradecer aos meus pais, Breno e Ana Carolina, por me apoiarem e fazerem com que eu e minha irmã concluíssemos nossos estudos. Também agradeço imensamente a minha namorada, Débora, por aguentar às minhas reclamações e lamúrias durante todo esse tempo. Aos meus professores da faculdade que me faziam gostar de eletrônica: Professor Hermano Cabral, José Sampaio, Sidney Lima, Gilson Júnior.

Por fim, aos amigos que sofreram junto a mim na faculdade que levo para a vida: Pedro Victor, Matheus Roma, Thales Castro, Railton Rocha, Bruna Santos, Natália Kitaoka, Júnior Mendes, Allyson Vitor, Vitor Mendes.

RESUMO

O Asa Branca é um projeto de extensão da Universidade Federal de Pernambuco em que são construídos aeromodelos voltados ao setor aeroespacial como: drone, foguete e minissatélites. Os seus integrantes participam de competições com diferentes focos. Durante as competições e experimentos, faz-se necessário a coleta e manipulação de dados quanto ao funcionamento dos aeromodelos. Entretanto o projeto Asa Branca apresenta limitações quanto à visualização de dados produzidos em tempo real. Como agravante, os referidos dados estão passivos de perda ou corrupção. O presente trabalho tem como objetivo solucionar as limitações quanto à manipulação, armazenamento e visualização de dados produzidos nos experimentos e competições aeroespaciais produzidos pelo projeto Asa Branca. De forma autoral, é criada uma interface gráfica para que seja possível a visualização de diversos dados através de gráficos e tabelas de modo a salvar os dados num banco de dados robusto e confiável. Java foi a linguagem de programação escolhida para o desenvolvimento da aplicação devido a sua consolidação no mercado e seu uso diversificado. É utilizada um API (*Application Programming Interface*) de modo a confeccionar a interface de envio entre os sistemas do Asa Branca e da ferramenta autoral. Através da API é possível transmitir mensagens entre sistemas desenvolvidos com tecnologias diferentes, mas que "acordam" em um formato padronizado de comunicação. Mais especificamente, uma API REST (*Representational State Transfer*) que funciona com mensagens no formato JSON (*JavaScript Object Notation*). Para a validação da aplicação na nuvem da AWS (*Amazon Web Service*), são feitos dois testes: um de estresse e outro de usabilidade. Tanto a interface quanto a API funcionam e apresentaram resultados positivos nos dois testes. Como aspecto positivo, os participantes do teste da versão inicial da ferramenta atestaram que a experiência vivenciada atendeu as expectativas. A meta é que a ferramenta autoral possa ser útil nos requisitos e nas competições pleiteadas pelo projeto Asa Branca.

Palavras-chave: API REST. Java. Asa Branca.

ABSTRACT

Asa Branca is an extension project of the Federal University of Pernambuco in which model aircraft are built for the aerospace sector such as drones, rockets, and mini satellites. Its members participate in competitions with different focuses. During the competitions and experiments, it is necessary to collect and manipulate data regarding the functioning of the model aircraft. However, the Asa Branca project has limitations regarding the real-time visualization of data produced. As an aggravating factor, this data is prone to loss or corruption. This work aims to solve the limitations regarding the manipulation, storage, and visualization of data produced in aerospace experiments and competitions by the Asa Branca project. An authorial graphical interface is created so that it is possible to visualize various data through graphs and tables to save the data in a robust and reliable database. Java was chosen as the programming language for the development of the application due to its consolidation in the market and diversified use. An API (Application Programming Interface) is used to make the interface between the Asa Branca systems and the authorial tool. Through the API, it is possible to transmit messages between systems developed with different technologies but that "agree" on a standardized communication format. More specifically, a REST (Representational State Transfer) API that works with messages in the JSON (JavaScript Object Notation) format. To validate the application in the AWS (Amazon Web Service) cloud, two tests are performed: one for stress and one for usability. Both the interface and the API functioned and presented positive results in the two tests. As a positive aspect, participants in the initial version of the tool's test attested that the experience lived up to expectations. The goal is for the authorial tool to be useful in the requirements and competitions sought by the Asa Branca project.

Keywords: REST API. Java. Asa Branca.

LISTA DE FIGURAS

Figura 1 - Nível de uma API REST. 20
Figura 2 - Fluxo básico do OAuth. 22
Figura 3 - Fluxo do código de autorização. 22
Figura 4 - Fluxo do acesso implícito. 24
Figura 5 - Fluxo das credenciais do dono do recurso. 25
Figura 6 - Fluxo de credenciais do cliente. 26
Figura 7 - Representação de um objeto JSON. 27
Figura 8 - Cabeçalho do *token*. 28
Figura 9 - Corpo do token. 28
Figura 10 - *Software* do *mysql workbench*. 32
Figura 11 - Especificação OpenAPI. 33
Figura 12 – Documentação dos *endpoints* do *OpenAPI*. 39
Figura 13 – Objetos utilizados na API. 40
Figura 14 - Fluxo dos objetos da API. 41
Figura 15 - Fluxo das credenciais do cliente exemplificado. 41
Figura 16 - Fluxo da autenticação do dono do recurso exemplificado. 42
Figura 17 - *Token* JWT e suas informações. 43
Figura 18 - Tela de login da aplicação. 44
Figura 19 - Tela inicial da aplicação. 45
Figura 20 - Página principal do Drone. 45
Figura 21 - Gráfico da altitude do drone. 46
Figura 22 - Tabela da altitude do drone. 46
Figura 23 - Criação do *cluster*. 50
Figura 24 - Dados do cubo salvos na AWS mostrados na tabela. 53
Figura 25 - Exemplo dos polos do *AttrakDiff*. 54
Figura 26 - a) Portifólio de resultados. b) Diagrama da média de valores. 55

LISTA DE TABELAS

Tabela 1- Dados coletos nos aeromodelos do Asa Branca..... 35
Tabela 2 - Diferença entre método do JPA e código em SQL. 37
Tabela 3 - *Endpoints* dos aeromodelos. 37
Tabela 4 - Tempos de respostas de requisições feitas à aplicação na nuvem. 52

LISTA DE ABREVIATURAS E SIGLAS

API – *Application Programming Interface*

ATT – *Attractiveness*

AWS – *Amazon Web Service*

CSV – *Comma-Separeted-Values*

DTO – *Data Transfer Object*

EC2 – *Elastic Compute Cloud*

ECR – *Elastic Container Registry*

ECS – *Elastic Conteiner Service*

GMT – *Greenwich Mean Time*

HATEOAS – *Hypermedia as The Engine of Application State*

HQ – *Hedonic Quality*

HQ-I – *Hedonic Quality - Stimulation*

HQ-S – *Hedonic Quality - Identity*

HTML – *HyperText Markup Language*

HTTP – *HyperText Transfer Protocol*

IDE – *Integrated Development Environment*

IETF – *Internet Engineering Task Force*

IP – *Internet Protocol*

JAR – Java *Archive*

JOSE – JSON *Object Signing and Encryption*

JPA – Java *Persistence* API

JRE – Java *Runtime Environment*

JSON – *JavaScript Object Notation*

JTI – JSON *Token Identifier*

JWE – JSON *Web Encryption*

JWS – JSON *Web Signature*

JWT – JSON *Web Token*

MAC – *Message Authentication Code*

MVC – *Model View and Controller*

MySQL – *My Structured Query Language*

PaaS – *Plataform as a Service*

POX – *Plain Old* XML

PQ – *Pragmatic Quality*

RDS – *Relation Database System*

REST – *Representational State Transfer*

RFC – *Request for Comments*

RPC – *Remote Procedure Call*

SD – *Secure Digital Card*

SHA – *Secure Hash Algorithm*

SOAP – *Simple Object Access Protocol*

UFPE – Universidade Federal de Pernambuco

URI – *Uniform Resource Identifier*

URL – *Uniform Resource Locator*

vCPU – *Virtual Centralized Processing Unit*

VPC – *Virtual Private Cloud*

WAR – *Web Archive*

XML – *Extensible Markup Language*

YAML – *Yet Another Markup Language*

SUMÁRIO

1 INTRODUÇÃO 15
1.1 Objetivo Geral 16
1.2 Objetivos Específicos 16
2 EMBASAMENTO TEÓRICO 17
2.1 O QUE É O ASA BRANCA 17
2.2 O QUE É UMA API REST 18
2.3 RESTRIÇÃO DE ACESSO 21
2.3.1 Código de autorização 22
2.3.2 Implícito 24
2.3.3 Credenciais do dono do recurso 25
***2.3.4* Credenciais do cliente 26**
2.3.5 *Token de Acesso* 26
2.4 *DEPLOY* DE UMA APLICAÇÃO NA NUVEM 28
3 Metodologia 30
3.1 MATERIAIS E MÉTODOS 30
4 DESENVOLVIMENTO 35
4.1 API REST 36
4.2 SEGURANÇA DA API 41
4.3 INTERFACE GRÁFICA 43
4.4 *"DOCKERIZANDO"* A APLICAÇÃO 46
4.5 *DEPLOY* DO *DOCKER* PARA A AWS 47
4.6 TESTE DO CÓDIGO APÓS A SUBIDA 52
5 CONCLUSÃO 57
6 Referências 58

1 INTRODUÇÃO

O Asa Branca é um projeto de extensão da Universidade Federal de Pernambuco (UFPE) voltado para o setor aeroespacial. Os seus membros desenvolvem aplicações para três tipos de aeromodelos para competições: drone, foguete e satélites miniaturizados (comumente chamados de *Cubsat* pelo seu formato em cubo) (AEROSPACE, 2022). Atualmente, não possuem uma forma segura de armazenar os dados coletados nessas competições e nem de visualizar em tempo real tais valores.

Esse trabalho tem como proposta o desenvolvimento de um sistema de visualização e armazenamento de dados através de uma API Rest e páginas *web*. Para isso, foi desenvolvida a aplicação utilizando a linguagem de programação Java, fazendo a conteinerização através da ferramenta *Docker* e, por fim, disponibilizando a aplicação na Amazon AWS para que possa ser acessada de qualquer computador. Conteinerização agrupa o código de uma aplicação e todos os arquivos necessários para ser executados em qualquer infraestrutura (AMAZON).

De modo a validar a ferramenta autoral, foram desenvolvidos vários cenários de estresse da plataforma-*web* visando avaliar a tolerância em falhas em momentos de múltiplas requisições em nuvem. Em adição à avaliação da tolerância a falhas, há o teste de usabilidade da ferramenta autoral através de análises quantitativas e qualitativas. Membros e ex-membros do projeto Asa Branca experimentaram a ferramenta. Como aspecto positivo, os participantes do teste atestaram a experiência vivenciada atendeu as expectativas.

O presente trabalho está dividido em seis capítulos, sendo o primeiro a introdução, o capítulo atual. Os demais são:

- **Embasamento Teórico:** É apresentado todo o conhecimento teórico necessário para o desenvolvimento da aplicação e seu *deploy* na AWS.
- **Metodologia:** Neste capítulo é apresentado os materiais utilizados para o desenvolvimento da aplicação.
- **Desenvolvimento:** Neste capítulo é apresentado como foi desenvolvida a aplicação e realizada a subida para a nuvem.
- **Resultados:** Neste capítulo são mostrados os resultados e testes realizados na ferramenta desenvolvida.
- **Conclusão:** É apresentada uma conclusão, recapitulando os objetivos alcançados e o que deve ser melhorado para trabalhos futuros.

1.1 OBJETIVO GERAL

Criar um banco de dados, dotado de web-interface, para que as equipes do Asa Branca possam armazenar e recuperar os dados nas competições e visualizá-los em tempo real através de gráficos e tabelas de forma multidimensional.

1.2 OBJETIVOS ESPECÍFICOS

- Criar uma *Application Programming Interface* (API) para que seja possível receber e armazenar os dados em um banco de dados SQL.
- Criar um sistema de autenticação para restringir o uso da API para usuários cadastrados.
- Criar uma página web para a visualização dos dados online.
- Utilizar boas práticas de desenvolvimento de *software* para garantir boa performance e manutenção do código.
- Elaborar teste de estresse na plataforma-web autoral de modo a validar a tolerância a falhas da ferramenta.
- Empregar teste de usabilidade visando medir a satisfação dos usuários ao interagir com a ferramenta autoral.

2 EMBASAMENTO TEÓRICO

Nesta seção, serão explicados os conceitos teóricos utilizados no desenvolvimento deste trabalho. Primeiramente, será explanado o que consiste no projeto Asa Branca e qual o seu objetivo. Em seguida, haverá uma explicação do que é uma API e algumas dos tipos mais usadas, bem como de que modo se pode proteger uma API e limitar o acesso aos seus *endpoints*. Ademais, serão discutidas algumas formas de visualização de dados e, por fim, como implementar o código em algum servidor na nuvem.

2.1 O QUE É O ASA BRANCA

O Asa Branca é um projeto de extensão vinculado à UFPE, que tem como missões: desenvolver e ampliar o setor aeroespacial no nordeste brasileiro, desenvolver e capacitar profissionalmente os estudantes envolvidos ao incentivar o crescimento pessoal por meio de atividades, como gerenciamento de projetos, desenvolvimento de tecnologias de aplicações aeroespaciais, integração e testes e participação em competições de soluções para CubeSats, drones e foguetes, além da divulgação científica em escolas e periódicos (AEROSPACE, 2022).

Esse projeto participa de competições de modalidades aeroespacial, os quais diferem entre si dos objetivos principais, mas que possuem algumas coisas em comum, sendo a principal delas a aquisição de dados do aeromodelo. Tais dados podem variar entre cada aeromodelo e até mesmo para cada missão.

Atualmente, os dados são gravados em um arquivo de extensão CSV (*Comma-Separeted-Values*) na estação base ou diretamente no aeromodelo durante o voo em um cartão SD (*Secure Digital Card*). Essa forma de armazenagem de dados em um arquivo CSV é perigosa, pois, segundo Xiaofeng Tang (TANG), ela precisa de três coisas para ser segura: preservar a integridade das informações, controle de acesso dos dados e preservar a privacidade dos contribuintes dos dados. Um arquivo CSV não cumpre nenhum dos pontos apresentados pelo autor, já que qualquer pessoa com acesso ao computador em que ele esteja armazenado pode editar ou excluir os dados presentes nele.

Logo, uma forma de manter os dados de uma forma mais segura é usando banco de dados em que se pode restringir o acesso para usuários específicos e até mesmo limitar acesso de edição, exclusão e visualização deles. No entanto, é preciso de algo que transforme os dados adquiridos nos aeromodelos em um formato compatível com um banco de dados, e isso é uma API.

2.2 O QUE É UMA API REST

API é um conjunto de rotinas e padrões estabelecidos para que dois componentes de *softwares* consigam se comunicar. Ou seja, são regras em que uma há uma troca de informações entre duas partes, onde as mensagens trocadas obedecem a um modelo pré-estabelecido descrito na sua documentação. Ela geralmente funciona no modelo de cliente e servidor, em que a aplicação que envia a solicitação é o cliente e a que envia a resposta é o servidor (AMAZON). Há algumas maneiras diferentes pelas quais elas podem funcionar:

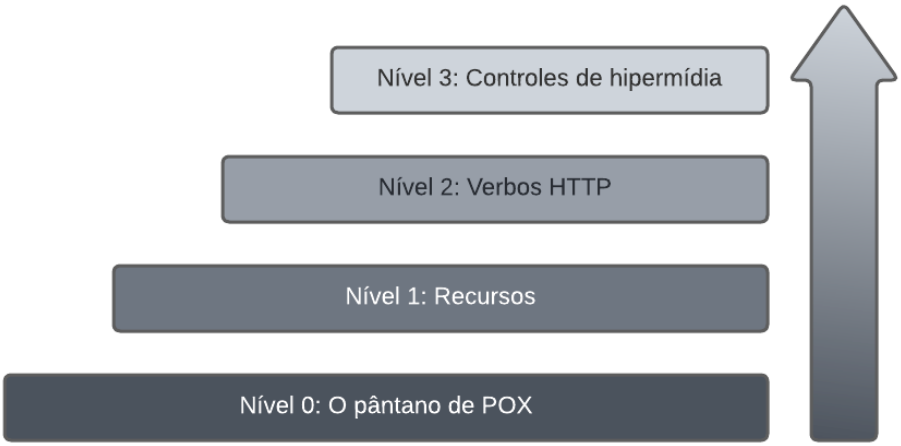
- APIs SOAP (*Simple Object Access Protocol*) trocam mensagens XML (*Extensible Markup Language*) utilizando o protocolo SOAP. Esse protocolo faz uso de regras específicas para o formato da mensagem, ela deve incluir um envelope, cabeçalho, corpo e uma falha que é opcional. Cada parte da mensagem tem sua funcionalidade e o envelope deve conter o *namespace* específico do SOAP.
- APIs RPC (*Remote Procedure Call*) são equivalentes a chamadas de funções numa aplicação, utilizando o nome de um método e seus argumentos. No caso de uma API HTTP (*HyperText Transfer Protocol*), os argumentos são passados no corpo da requisição e o método, na URL (*Uniform Resource Locator*).
- APIs *WebSockets* se utilizam da tecnologia do *websocket* que é um canal duplex, onde cliente e servidor criam uma conexão e podem se comunicar livremente ao contrário do comum em que a conexão é estabelecida a partir de uma mensagem de requisição partindo do usuário e finalizando com a resposta do servidor. Esse tipo de comunicação é muito útil quando há diversas requisições para o servidor partindo do mesmo cliente.
- APIs REST (*Representational State Transfer*) seguem um conjunto de restrições organizadas e publicadas em 2000 por Roy Fielding em seu PHD denominado *"Architectural Styles and the Design of Network-based Software Architectures"*.

As APIs REST, apesar de obedecerem a certas restrições para serem consideradas RESTful, ainda são as mais práticas e flexíveis na implementação, por isso que esse projeto irá seguir com ela. Segundo Roy Fielding (FIELDING, 2000), para uma API ser considera RESTful, ela precisa:

- Ter arquitetura cliente-servidor, pois melhora a portabilidade da interface com o usuário para várias plataformas e a escalabilidade. Separar os componentes permite a melhor manutenção e a adição de novas funcionalidades.
- A comunicação cliente-servidor precisa ser sem estado, ou seja, o cliente não armazena informações de transições anteriores logo, cada requisição feita do cliente precisa ter todas as informações necessárias para ser compreendida.
- Para melhorar a eficiência do uso da banda larga, é adicionado a memória cache. Para isso, é necessário que as respostas do servidor sejam anotadas, implicitamente ou explicitamente, como sendo possível armazenar em cache ou não.
- Possivelmente, a regra principal da arquitetura REST é a uniformidade das interfaces entre componentes. Fazendo isso, deixa-se mais visível a arquitetura do sistema e suas interações. Essas interações ficam desacopladas do resto do serviço, permitindo que cada componente evolua de forma independente.
- O sistema em camadas faz com que a arquitetura seja composta por camadas hierárquicas limitando que cada componente só possa "ver" o nível imediatamente próximo.
- A última regra é opcional e fala sobre código sob demanda, ou seja, permite que a expansão das funcionalidades do cliente posso ser baixada e a execução de extensões no formato de *applets* ou *scripts.* Isso reduz o número de código que precisa ser implementado de início e pode ser estendido conforme necessidade.

A partir dessas regras, o autor e especialista em API REST Leonard Richardson (FOWLER, 2010) desenvolveu um modelo de maturidade contendo quatro níveis: zero, o pântano de POX (*Plain Old* XML); um, recursos; dois, verbos HTTP; três, controles de hipermídia.

Figura 1 - Nível de uma API REST.



Fonte: Figura traduzida, original disponível em (FOWLER, 2010).

1. No nível zero é exposto apenas uma *Uniform Resource Identifier* (URI), normalmente usando o verbo POST, aceitando todos os tipos de operações para aquele serviço.
2. No nível um é introduzido os recursos e permite fazer requisições para URIs (ainda normalmente usando o verbo POST) individuais para diferentes ações ao invés de um endpoint universal. Nesse nível já é possível identificar os escopos de cada recurso.
3. No nível dois começa-se a usar os verbos HTTP, permitindo a especialização de cada recurso e assim diminuir a funcionalidade de cada operação individual com o serviço.
4. No nível três usa-se a representação de hipermídia ou HATEOAS (*Hypermedia as The Engine of Application State*), que são elementos contidos nas respostas das mensagens dos recursos os quais estabelecem relações.

Esses níveis servem para determinar o quanto uma API seguiu as restrições de Roy Fielding e diz que uma de nível três é uma RESTful pois ela segue todas as condições estabelecidas. Entretanto, é necessário analisar a necessidade e custo dessa implementação para a aplicação, pois pode não haver a necessidade de seguir tudo e ainda assim a API ser funcional para sua aplicação.

Em nenhuma das restrições é mencionado quem deve ter acesso aos *endpoints*, pois isso é algo que depende do uso da API. Elas podem ser ditas públicas ou privadas: a pública

sendo aberta para qualquer um acessar e a privada com restrições de acessos a usuários específicos e conhecidos.

2.3 RESTRIÇÃO DE ACESSO

O *Internet Engineering Task Force* (IETF) é um grupo internacional aberto que tem como objetivo fazer a Internet funcionar melhor. O RFC (*Request for Comments*) 3935 (ALVESTRAND, 2004) dita o que ele é, seu objetivo e sua missão. Lá, pode-se ver que sua missão é produzir documentos técnicos de alta qualidade e relevantes que influenciem como as pessoas projetam, usam e administram a Internet de um modo que a faça funcionar melhor. Esses documentos incluem protocolos padrões, melhores práticas e documentos informativos de vários tipos.

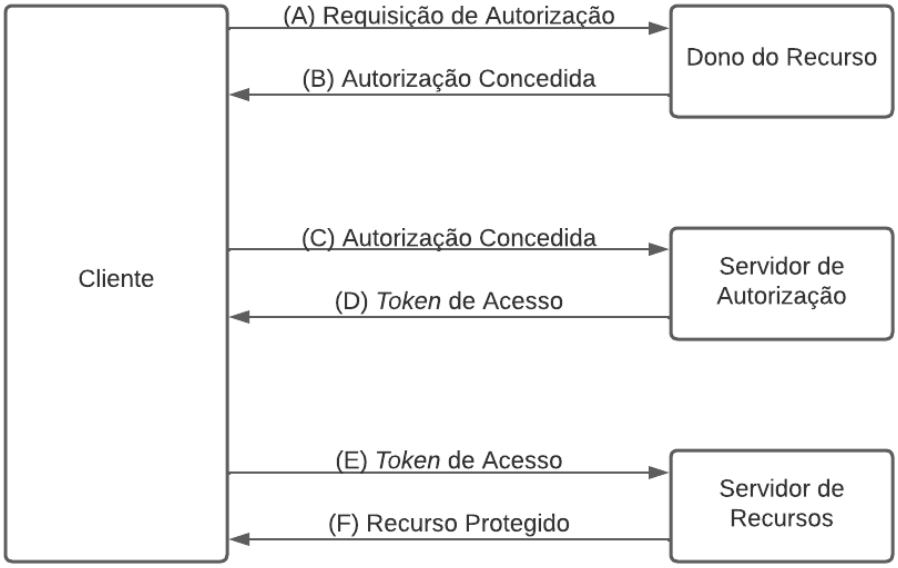
Seguindo o IETF para protocolos de autorização utilizando requisições HTTP, tem-se o OAuth 2.0 que substituiu a versão 1.0. O OAuth 2.0 é descrito no RFC 6749 (HARDT, 2012) e diz que ele é um *framework* que permite uma aplicação acesso limitado a um serviço HTTP. Por ele ser indicado pelo IETF como protocolo para requisições HTTP, será utilizado nesse projeto para limitar o acesso aos *endpoints* da API REST.

O OAuth define quatro papéis:

- O dono do recurso é uma entidade capaz de garantir acesso a um recurso protegido.
- O servidor de recursos é aquele que armazena os recursos protegidos, capaz de aceitar e responder requisições que utilizam *tokens* de acesso, que são textos contendo informações não sigilosas.
- O cliente é uma aplicação que faz requisições aos recursos protegidos em nome do dono do recurso e com sua autorização.
- O servidor de autorização distribui *tokens* de acesso aos clientes depois de autenticar com o dono do recurso e obter autorização.

Utilizando-se desses quatro papéis, o fluxo básico do OAuth funciona de acordo com a Figura 2. No passo (A), o cliente faz uma requisição de autorização para o dono do recurso, o qual responde em (B) retornando uma credencial concedendo acesso ao cliente; com ela, o cliente faz a requisição para o servidor de autorização em (C) e recebe a resposta com o *token* de acesso em (D); por fim, o *token* é usado em (E) para ter acesso ao servidor de recursos e receber o recurso protegido em (F).

Figura 2 - Fluxo básico do OAuth.

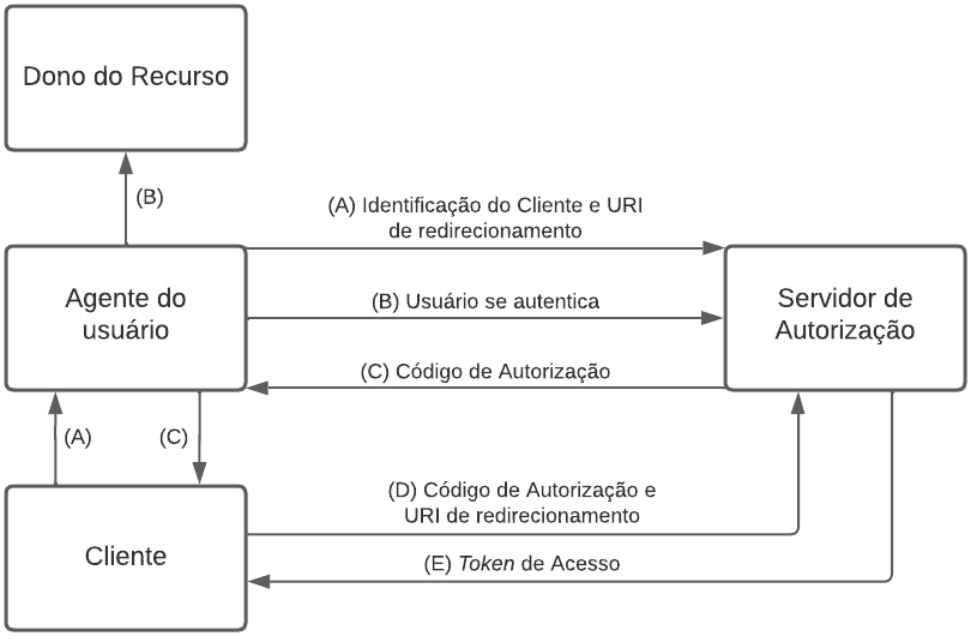


Fonte: Figura traduzida, original disponível em (HARDT, 2012).

A RFC define quatro tipos de concessões de credenciais: código de autorização, implícito, credenciais do dono do recurso e credenciais do cliente.

2.3.1 Código de autorização

Figura 3 - Fluxo do código de autorização.



Fonte: Figura traduzida, original disponível em (HARDT, 2012).

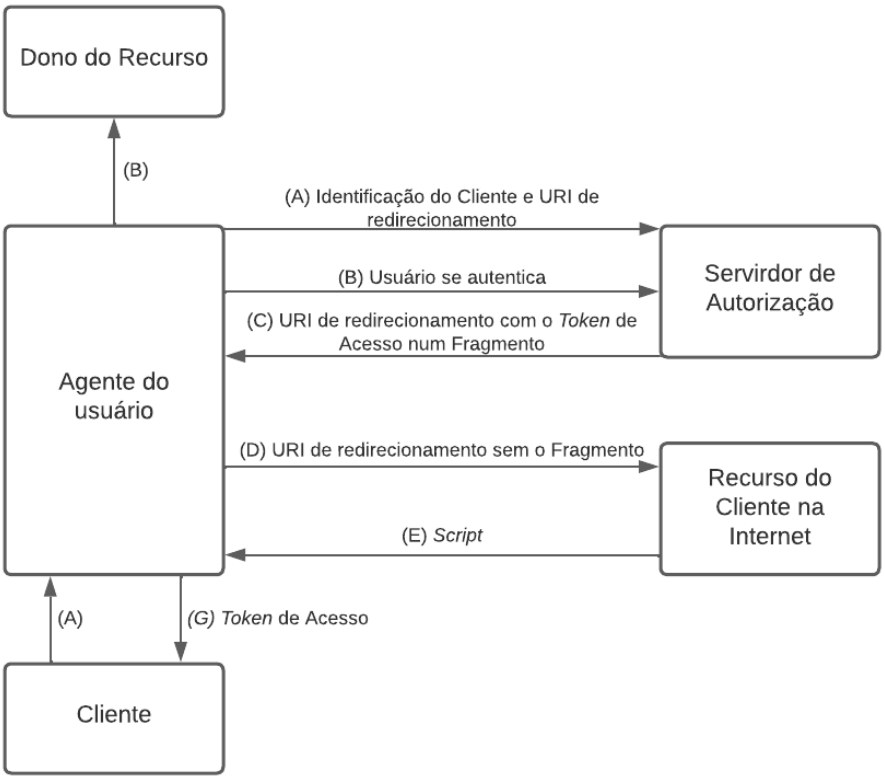
A Figura 3 ilustra o fluxo de acesso através do código de autorização, o qual consiste em ter o servidor de autorização como intermediário entre o cliente e o dono do recurso. Neste caso, o cliente deve ser capaz de interagir com o agente (normalmente um navegador) do dono e receber requisições do servidor de autorização.

Em (A), o cliente inicia o fluxo direcionando o agente para o *endpoint* de autorização, o qual inclui na requisição seu identificador de usuário, o escopo requerido, o estado local e a URI de redirecionamento na qual o servidor de autorização mandará o agente de volta assim que souber se o acesso foi garantido ou recusado. O dono é autenticado através do agente e garante ou não o acesso ao cliente em (B). Assumindo que o dono concedeu o acesso em (C), o agente é redirecionado pelo servidor de autorização, através da URI fornecida antes, incluindo um código de autorização e qualquer estado local que o cliente tenha fornecido. No passo (D), o cliente faz uma requisição ao *endpoint* do servidor de autorização para conseguir um *token* de acesso; nela, ele inclui o código de autorização recebido no passo anterior e a URI de redirecionamento que enviou para o agente. Por fim, o servidor de autorização valida o cliente, o código de autorização e verifica se a URI fornecida é a mesma do passo (C); se for tudo válido, ele responde com o *token* de acesso.

Este fluxo é bom para clientes privados, pois não expõe as credenciais do dono do recurso, apenas o servidor de autorização as conhece. Isso também inclui o cliente que se autentica diretamente com ele e o *token* de acesso é passado diretamente sem o agente do dono ter conhecimento.

2.3.2 Implícito

Figura 4 - Fluxo do acesso implícito.



Fonte: Figura traduzida, original disponível em (HARDT, 2012).

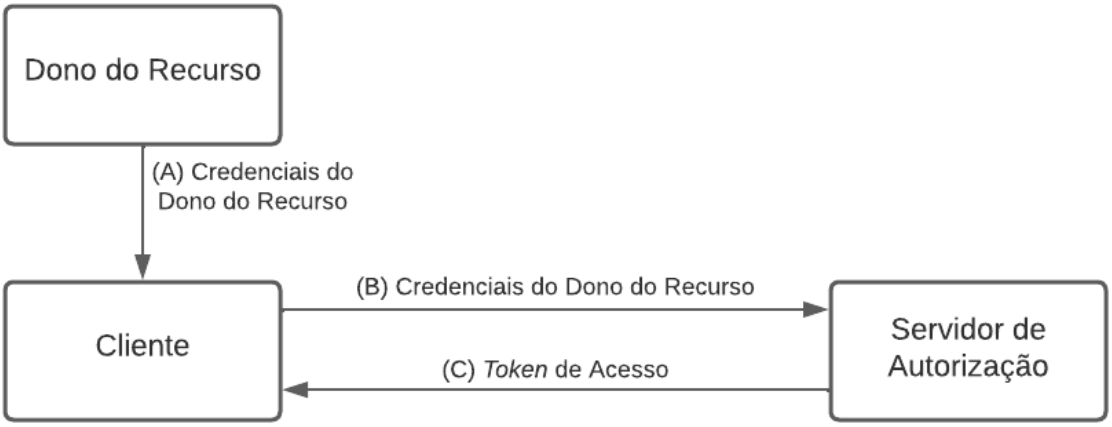
A Figura 4 mostra o fluxo do acesso implícito que é uma simplificação do código de acesso otimizada para clientes implementados em navegadores utilizando *scripts*, pois não há um código de acesso para o cliente obter o *token* de acesso ao qual é dado diretamente. O servidor de autorização não valida o cliente, porque ele pode ser identificado pela URI de redirecionamento em alguns casos.

Os passos (A) e (B) são iguais aos do código de autorização. Novamente assumindo que o dono do recurso garante o acesso ao cliente em (C), o agente é redirecionado pelo servidor de autorização para a URI fornecida nos passos anteriores e nela inclui o *token* de acesso no fragmento de URI. O agente faz uma requisição ao recurso do cliente seguindo as instruções de redirecionamento e que não contém o fragmento, retendo a informação internamente em (D). O recurso do cliente retorna uma página web capaz de acessar toda a URI de redirecionamento e extrair o *token* de acesso do fragmento em (E). No passo (F), o agente executa o *script* recebido anteriormente e extrai o *token* internamente passando em (G).

Esse fluxo expõe o *token* de acesso ao dono do recurso e às aplicações que tem acesso ao agente dele, trazendo uma grande falha de segurança caso alguém consiga obtê-lo. Porém, o fluxo implícito melhora a responsividade de alguns clientes já que reduz o número de idas e voltas para conseguir o *token*. É preciso ver se as falhas de segurança compensam o ganho de velocidade e a implementação quando se for utilizar.

2.3.3 Credenciais do dono do recurso

Figura 5 - Fluxo das credenciais do dono do recurso.



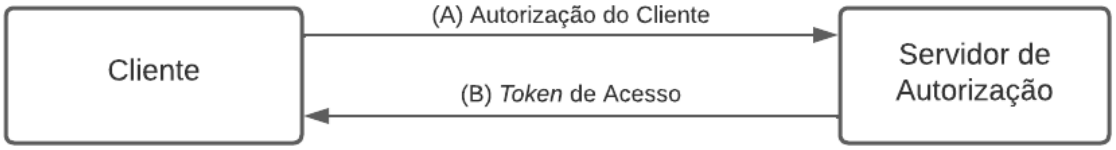
Fonte: Figura traduzida, original disponível em (HARDT, 2012).

Esse tipo de fluxo, mostrado na Figura 5, deve ser usado quando se há muita confiança entre o dono do recurso e o cliente e quando outros fluxos não são viáveis, pois o cliente terá acesso direto às credenciais do dono. Elas são usadas para adquirir o *token* que dure muito tempo para que não seja necessário o armazenamento das credenciais. É utilizado quando tanto o *backend*, quando o *frontend* são desenvolvidos pela mesma empresa e não terá nenhum acesso externo.

O fluxo se inicia em (A), onde o cliente recebe as credenciais do dono do recurso. Em (B) é feito a requisição para o servidor de autorização utilizando os dados recebidos do dono junto com a autenticação do cliente. Supondo que ambas as autenticações estão válidas, em (C), é enviado o *token* de acesso.

2.3.4 Credenciais do cliente

Figura 6 - Fluxo de credenciais do cliente.



Fonte: Figura traduzida, original disponível em (HARDT, 2012).

O mais simples dos fluxos de autorização está mostrado na Figura 6. Nele, o cliente se autentica utilizando suas credenciais em (A) e, se válido, o servidor de autorização responde com um *token* em (B). Esse tipo de autorização é normalmente utilizado entre aplicações *backend*, onde não há iteração com usuários.

2.3.5 *Token de Acesso*

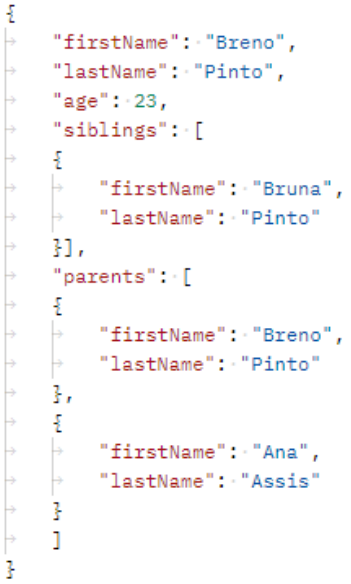
Tokens de acesso são credenciais usadas para acessar recursos protegidos. Elas são pedaços de texto aparentemente sem nenhum sentido, mas contém informações sobre o cliente, o escopo do acesso, tempo de duração da permissão, além de que podem ser personalizadas para conter algum dado em específico.

Há várias formas de ser codificar um *token*, mas existem duas classificações entre elas: os opacos e os transparentes. Os opacos não transmitem nenhuma informação contida dentro dele e é preciso do servidor de autorização para que seja autenticado, enquanto os transparentes não, pois eles podem ser lidos por qualquer um.

Este trabalho irá trabalhar com *tokens* JWT (*JSON Web Token*) que são transparentes e são apresentados na RFC 7519 (JONES, BRADLEY e SAKIMURA, 2015) do IETF. Segundo a especificação, um *JavaScript Object Notation* (JSON) *Web Token* é um texto contendo um conjunto de declarações como um objeto JSON que pode ser codificado como um JWS (JSON *Web Signature*) ou JWE (JSON *Web Encryption*), permitindo que as declarações sejam digitalmente assinadas, protegidas por *Message Authentication Code* (MAC) e/ou criptografadas. Um objeto JSON é descrito na RFC 7159 (BRAY, 2015), em que consiste em zero ou mais conjuntos de nomes e valores (formando um par), onde um nome é um texto e um valor, um JSON. A Figura 7 mostra um exemplo desse modelo de representação.

Um *token* JWT é representado por partes separadas por ponto, em que cada uma delas contém um valor codificado em base 64. O número de partes depende da representação usada, JWS ou JWE. Entretanto, a primeira parte, o cabeçalho JOSE (JSON *Object Signing and Encryption*), sempre aparece e define qual das duas representações é utilizada.

Figura 7 - Representação de um objeto JSON.



Fonte: Elaborada pelo autor.

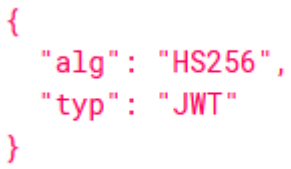
Esse projeto focará em *tokens* JWT utilizando JWS. Eles são divididos em três partes separadas por ponto: o cabeçalho, o corpo e a assinatura. Um exemplo de um *token* está abaixo:

- eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2NTU0NDU4MTgsInVzZXJfbmFtZSI6ImJyZW5vIiwianRpIjoiZDZmZGZkNTQtNDEzZC00NmJiLTlmMDAtZDk3NGE0YTk2YzVkIiwic2NvcGUiOlsiUkVBRCIsIldSSVRFIl19.c3d6md3a6mBwenOpDw_DIS9Ia5YVbBQQxaLQ2uY2ZQ8
    a. eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 é o cabeçalho e nele é possível dizer que é um *token* JWT e está codificado com o algoritmo SHA-256 (*Secure Hash Algorithm* – 256). Ele está mostrado na Figura 8.
    b. eyJleHAiOjE2NTU0NDU4MTgsInVzZXJfbmFtZSI6ImJyZW5vIiwianRpIjoiZDZmZGZkNTQtNDEzZC00NmJiLTlmMDAtZDk3NGE0YTk2YzVkIiwic2NvcGUiOlsiUkVBRCIsIldSSVRFIl19 é o corpo mostrado na Figura 9. Nele, contém algumas informações a respeito do cliente, seu nome, o escopo que ele terá acesso ao servidor de recursos, o tempo de

validade em que o *token* é válido (em segundos), e o JTI (JSON *Token Identifier*) que é um identificador único para cada objeto.

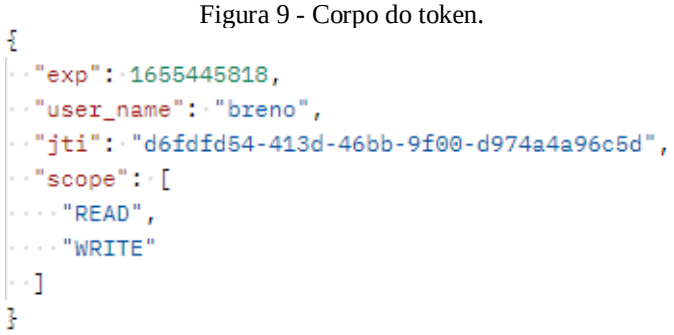
c. c3d6md3a6mBwenOpDw_DIS9Ia5YVbBQQxaLQ2uY2ZQ8 é a assinatura do *token* e é a partir dela que é vista a autenticidade da chave.

Figura 8 - Cabeçalho do *token*.



Fonte: Elaborada pelo autor.

Figura 9 - Corpo do token.



Fonte: Elaborada pelo autor.

2.4 *DEPLOY* DE UMA APLICAÇÃO NA NUVEM

A principal forma de se prover aplicações através da rede mundial de computadores é pela nuvem, onde são acessadas, armazenadas e podem ser acessados bancos de dados. Sendo assim, é muito importante que a aplicação seja implementada em algum servidor em nuvem para que todos possam ter acesso. Os provedores de espaço de armazenamento em nuvem, como a *Amazon* AWS (*Amazon Web Service*), *Google*, *Heroku e Microsoft*, são marcos que concretizam a massificação do referido serviço perante a sociedade contemporânea. Cada uma dessas empresas tem seus preços e tipos de serviço disponibilizados. Este projeto focará na *Amazon* AWS, pois fornece um plano anual com diversos serviços gratuitos no primeiro ano de uso, além de que só se paga pelo tempo de uso, então caso a aplicação não esteja ativa, não haverá custos adicionais (SERVICES, 2023).

Os projetos Java podem ser empacotados para *deploy* em duas principais formas: arquivos JAR (Java *Archive*) e WAR (*Web Archive*). O JAR é uma extensão que precisa de uma Java *Runtime Environment* (JRE) para ser rodado, pois foi desenvolvido pela *Oracle*, que é a empresa desenvolvedora do Java, para armazenar projetos em um único arquivo. Enquanto

o WAR é um aplicativo especificamente para a *web* contendo todo o projeto, mas que pode ser utilizado por outras linguagens além do Java. No caso deste projeto, será utilizado o formato JAR por ser mais versátil, pois o WAR tem uma estrutura pré-definida de pastas que devem ser seguidas e para ser executado é preciso de um ser servidor, enquanto o JAR pode ser executado em qualquer máquina que tenha uma JRE.

É possível fazer a implementação desses dois tipos de arquivos na nuvem, mas é preciso que haja o suporte delas, além de que a versão do JRE precisa ser compatível com a do projeto desenvolvido. Depender dessas variáveis não é o ideal, pois pode limitar o desenvolvimento para versões específicas do Java apenas para atender ao requisito de *deploy*, por isso uma solução é o *docker*.

O *docker* é um conjunto de produtos de PaaS (*Plataform as a Service*) que usam virtualização de nível de sistema operacional para prover *software* em pacotes chamados contêineres. Cria-se uma imagem de um contêiner contendo todo o código, dependências, configurações, bibliotecas, tudo o que for preciso para o código funcionar. Quando essa imagem é executada dá-se o nome de contêiner e pode ser executado em qualquer lugar. Isso evita o retrabalho de uma nova configuração de um ambiente de trabalho (INC, 2022).

A AWS tem suporte para a execução de contêineres e pode-se executar quantos quiser, mas paga-se um preço pela quantidade de recursos alocados. No primeiro ano, executando apenas um quando necessário, o valor econômico fica baixo, por isso foi uma preferência na hora da escolha do serviço de nuvem. Para subir um contêiner, usa-se o *Elastic Conteiner Service* (ECS) e é lá que é disponibilizado o IP (*Internet Protocol*) público da aplicação para se acessar de qualquer.

3 METODOLOGIA

Primeiramente foi feito uma reunião com representantes do Asa Branca para entender suas necessidades e saber as funcionalidades que desejavam na aplicação. Em seguida, foi feito o planejamento dos *endpoints* da API. Por fim, foi desenvolvido o esquema das páginas *web* para a visualização dos dados.

Após desenvolvido o projeto, foi feita uma pesquisa pelos provedores de espaço de armazenamento em nuvem para encontrar o melhor custo-benefício para se hospedar e manter uma aplicação. Depois de escolhida a AWS, foi disponibilizado o IP público para uso na rede mundial de computadores.

3.1 MATERIAIS E MÉTODOS

O projeto foi feito utilizando o *framework Spring Boot* do Java. Ele é um *framework* que facilita a configuração de projetos baseando-se nos padrões de projeto de inversão de controle e injeção de dependência. A inversão de controle trata-se do desacoplamento das classes, dando a responsabilidade de criação instanciação de um elemento para o *Spring*. Essa inicialização é feita pela injeção de dependências evitando o acoplamento de código na aplicação e facilitando a manutenção.

O *Spring* utiliza os *beans*, que são classes configuradas para poderem ser utilizadas na injeção de dependências. Configurar, entretanto, cada classe individualmente teria sido muito trabalhosa, então foi desenvolvido o *Spring Boot*. Ele faz a configuração automática dos *beans* a partir dos subpacotes da classe principal.

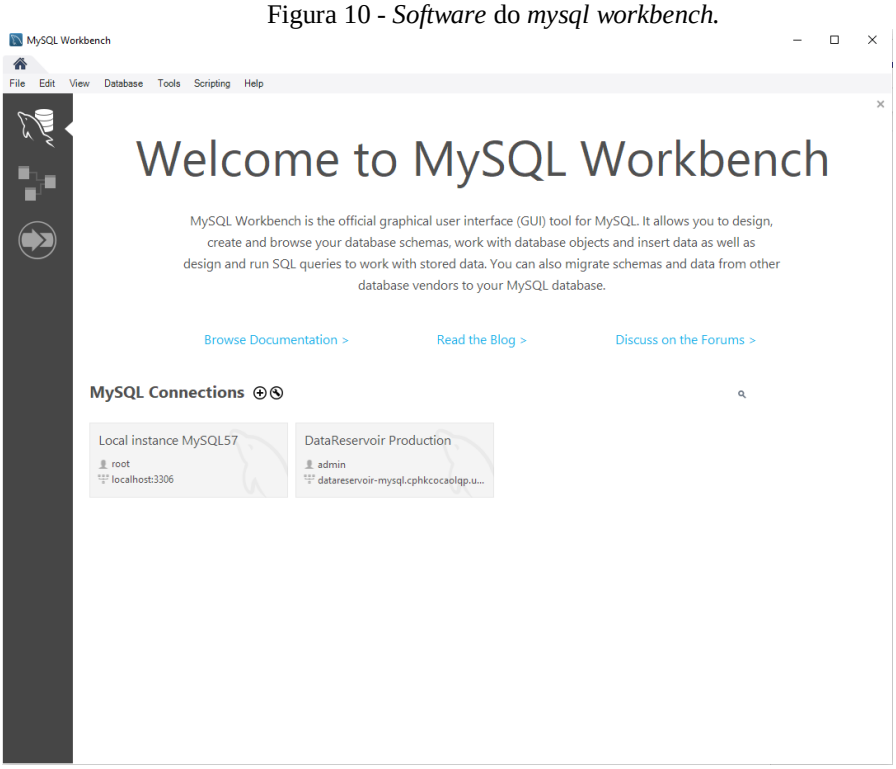
Muitos projetos se utilizam dos mesmos componentes e para facilitar e diminuir a configuração das mesmas coisas para todos, foram criados projetos base que podem ser usados como padrão e mudados quando preciso. Alguns exemplos são o *Spring Data* para conexão a banco de dados, *Spring Security* para segurança da aplicação e *Spring Web* para serviços *web* como criação de APIs. Para administrar todos os projetos bases utilizados, usa-

se um gerenciador de dependências, o *Maven* ou o *Gradle*. Ambos têm a mesma funcionalidade, mas utilizam linguagens e são mantidos por empresas diferentes, então por preferência pelo XML do *Maven* ao invés do *Groovy* do *Gradle*, será utilizado o primeiro. Ele baixa bibliotecas Java dinamicamente de repositórios remotos, sendo o principal o *Maven* 2 *Central Repository*, e armazena-os localmente para que possa ser utilizado no projeto. Além disso, também é possível fazer com que projetos locais sejam postos como dependências.

Durante o desenvolvimento do código fonte, foi utilizado a Eclipse IDE (*Integrated Development Environment*) como ambiente de desenvolvimento. Ela é um *software* gratuito distribuído pela Eclipse *Foundation* tendo várias versões disponíveis, sendo a principal delas para desenvolvimento *web*. Uma IDE corresponde a um ambiente de desenvolvimento integrado em que se pode editar o código, compilar, fazer o *deploy*, encontrar erros através da ferramenta de *debug*, tudo que é preciso para fazer um código funcionar.

Para o armazenamento dos dados, foi utilizado o banco de dados MySQL (*My Structured Query Language*) que é gratuito para testes, mas é necessário pagar uma taxa quando se é utilizado comercialmente ou em algum fornecedor de espaço na nuvem. Junto com o *mysql* pode ser instalado o *workbench*, Figura 10, que é basicamente uma interface gráfica para visualização dos bancos, tabelas, dados e os usuários permitidos. O pacote da AWS fornece suporte para esse banco além de estar incluso no pacote gratuito do primeiro ano.

Figura 10 - *Software* do *mysql workbench.*

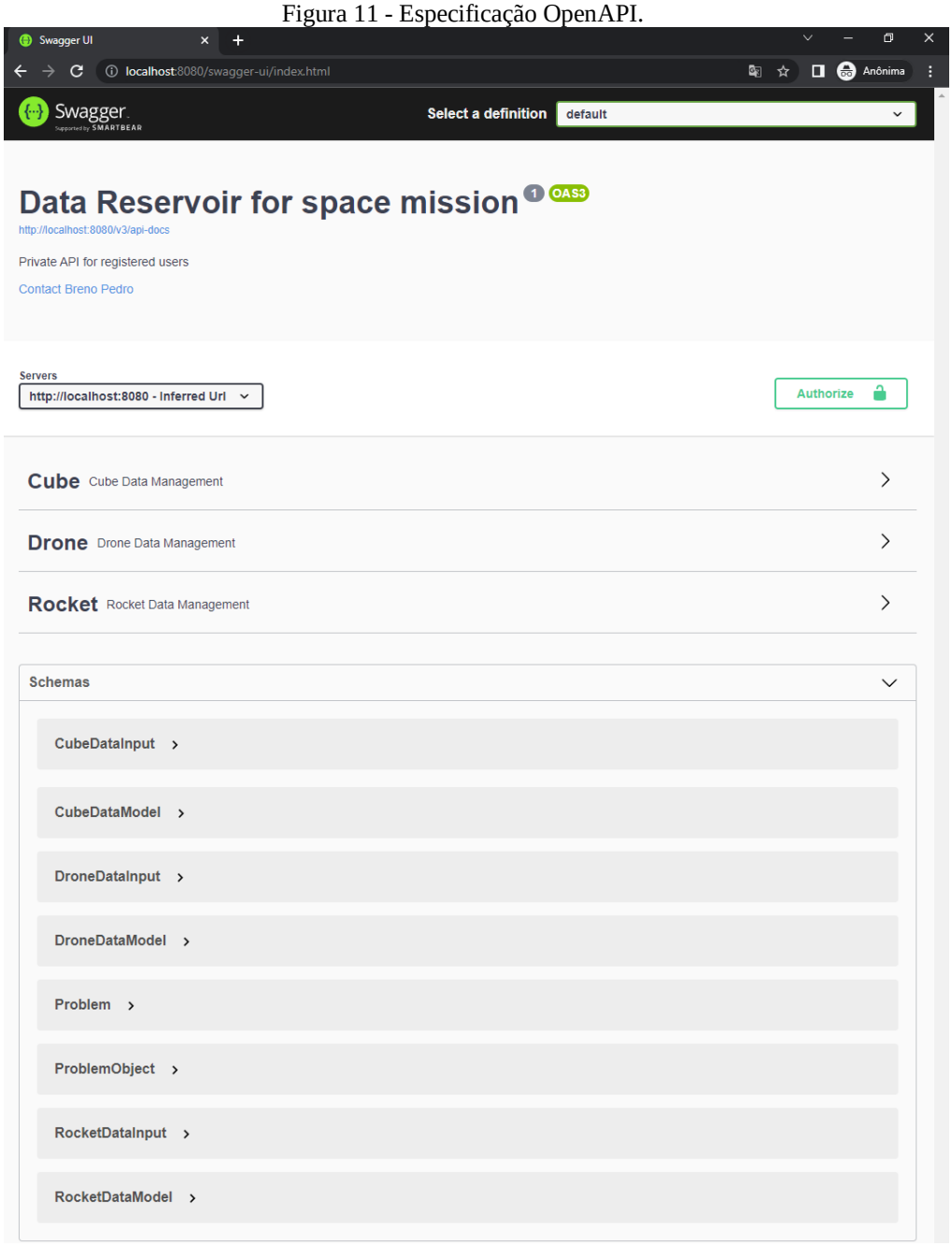


Fonte: Elaborada pelo autor.

Ao longo do desenvolvimento foram feitos testes fazendo o *deploy* no computador local através do Eclipse e fazendo requisições HTTP com o *Postman*. Ele é um *software* majoritariamente gratuito feito para o desenvolvimento de APIs. A versão paga oferece planos para desenvolvimento em times de larga escala, mas a versão gratuita já fornece uma gama de usos como: fazer requisições HTTP, criar um arquivo YAML (*Yet Another Markup Language*) representando uma API seguindo as convenções do OpenAPI, criação de servidores falsos que devolvem respostas prontas simulando uma API, fazer rotinas periódicas para testar a performance de uma API.

A especificação do OpenAPI define um padrão para interfaces de APIs RESTful, permitindo o entendimento das funcionalidades e capacidades dos serviços oferecidos por ela, sem precisar verificar código fonte, documentação ou interromper o funcionamento do servidor para pegar uma mensagem. Assim, é possível interagir com a API sem precisar implementar nada. Depois, pode-se usar essa especificação para se implementar clientes ou servidores em qualquer linguagem de programação. Foi usado o OpenAPI da Swagger em que é disponibilizado uma URL onde é possível fazer requisições para a API sem precisar implementar nada, apenas acessando de um navegador como mostrado na Figura 11.

Figura 11 - Especificação OpenAPI.



Fonte: Elaborada pelo autor.

Diferentemente da Figura 11 que já é implementada pelo OpenAPI, as telas de gráficos e tabelas para visualização dos dados não são e para isso foi usado o *Thymeleaf*. Ele é um *template engine* permitindo que linguagens de programação como o Java possam ser incorporadas nas páginas HTML (*HyperText Markup Language*) e usar estruturas de repetição, herança, condições. O *Thymeleaf* é muito utilizado em projetos que usam o padrão de projeto MVC (*Model View and Controller*) e é gerado no lado do servidor, por isso é bem integrado ao *backend.*

Após desenvolvido o código, foi utilizado o *Docker Desktop* para administrar e rodar as imagens e containers *docker*. Ele é um *software* gratuito que pode ser instalado nos sistemas *Windows,* Linux e *Mac* disponibilizado pela empresa Docker. Nele, é possível iniciar, parar e excluir os containers, assim como excluir imagens e enviá-las para o *Docker Hub* ou outro serviço de hospedagem de imagens.

O *deploy* da aplicação foi feito seguindo as instruções fornecidas pela própria AWS na sessão ECS, onde é possível hospedar e executar um contêiner *docker.* Além disso, também foi utilizado o *RedisLabs* para armazenamento de sessão de usuário.

4 DESENVOLVIMENTO

O presente trabalho tem como objetivo o desenvolvimento de uma interface gráfica para visualização em tempo dos dados coletados pelos modelos aeroespaciais desenvolvidos no projeto Asa Branca da UFPE. Além da visualização em tempo real, também serão armazenados os dados para análise posterior.

O projeto pode ser dividido em duas partes, a parte do *backend*, a comunicação com a estação base, o armazenamento e a segurança da aplicação; e o *frontend*, a página *web* com os gráficos e tabelas para visualização dos dados.

Antes de começar o desenvolvimento do código, foi feita uma reunião com representantes do Asa Branca para decidir quais dados gostariam de serem gravados em cada um dos aeromodelos que estão representados na Tabela 1.

Tabela 1- Dados coletos nos aeromodelos do Asa Branca.

CubSat	Foguete	Drone
Temperatura Externa	Posição X do GPS	Altitude
Corrente da Bateria	Posição Y do GPS	Corrente da Bateria
Tensão da Bateria	Posição Z do GPS	Tensão da Bateria
Temperatura da Bateria	Campo Magnético X	Posição X do GPS
Campo Magnético X	Campo Magnético Y	Posição Y do GPS
Campo Magnético Y	Campo Magnético Z	Posição Z do GPS
Campo Magnético Z	Velocidade Linear X	Data e Hora no Drone
Ângulo de Euler X	Velocidade Linear Y	Data e Hora na Estação Base
Ângulo de Euler Y	Velocidade Linear Z	
Ângulo de Euler Z	Velocidade Angular X	
Velocidade Linear X	Velocidade Angular Y	
Velocidade Linear Y	Velocidade Angular Z	
Velocidade Linear Z	Data e Hora no Foguete	
Velocidade Angular X	Data e Hora na Estação Base	
Velocidade Angular Y		
Velocidade Angular Z		
Potência Transmitida do Transceptor		
Potência Recebida do Transceptor		
Data e Hora no CubSat		
Data e Hora na Estação Base		

Fonte: Elaborada pelo autor.

Começando pelos dados do CubSat tem-se a temperatura externa do cubo, a temperatura de onde ele está; os dados da bateria, corrente, tensão e temperatura dela; o campo magnético que é obtido nas três coordenadas; o ângulo de Euler[1] também obtido em três coordenadas; as velocidades linear e angular nas três coordenadas; as potências transmitida do transceptor no cubo e recebida na estação base; a data e hora em que os dados foram coletos no cubo e chegaram na estação.

Em relação ao foguete há algumas similaridades como o campo magnético, velocidades linear e angular e as datas de horas coletados no aeromodelo e recebida na estação base. Mas há um campo novo que é a posição do GPS[2] também coletada em três dimensões. No drone, além dos campos semelhantes aos outros aeromodelos, há a altitude também armazenada.

4.1 API REST

Em posse dos dados a serem armazenados, foi iniciado o desenvolvimento da API utilizando um dos principais padrões de *design* em desenvolvimento *web* com Java; o MVC. Nesse padrão, existem três papéis: o modelo representando informações do domínio, geralmente sendo um modelo de domínio em linguagens orientadas a objeto como o Java; a vista que representa o modelo mostrado ao usuário; o controlador que recebe os dados do usuário, os manipula e entrega a resposta (FOWLER, 2006).

No caso da API, o modelo seria de domínio simples, ou um objeto de domínio em que uma classe representaria uma tabela no banco de dados. Além dos dados mostrados na Tabela 1, também há a chave primária. Todos esses campos são discernidos em uma classe representando uma tabela no banco.

O projeto *Spring JPA* (Java *Persistence API*) faz parte do ecossistema *Spring* e facilita o trabalho com persistência de dados. Ele permite utilizar um objeto Java como uma tabela em um banco de dados e já tem alguns métodos prontos visando recuperar e armazenar como *findById* que busca um objeto pelo seu Id e o *save* que persiste o dado no banco. Além de ser trivial sua extensão através do mecanismo de derivação de consulta. Com ele, é possível consultar o banco utilizando palavras chaves ao invés de *queries* em linguagens como SQL (GIERKE, DARIMONT, *et al.*, 2023). Um exemplo disso é mostrado na Tabela 2.

[1] O Ângulo de Euler descreve a orientação do corpo girante em relação a um sistema inercial fixo.

[2] GPS ou o *Global Positioning System* (Sistema de Posicionamento Global) é um sistema para se localizar globalmente.

Tabela 2 - Diferença entre método do JPA e código em SQL.

Método criado através do *Spring JPA*	**Código em SQL**
findExternalTemperatureGreaterThan100	*SELECT * FROM cube_data WHERE external_temperature > 100;*

Fonte: Elaborado pelo autor.

No segundo papel do *design*; a vista seria um elemento JSON retornado nos casos de consulta (verbo *GET*) e quando é salvo algum dado (verbo *POST*). Porém não é recomendado utilizar o mesmo objeto do modelo já que ele é usado na persistência dos dados. Além de que também é possível disponibilizar *endpoints* que não precisem retornar todos os dados do modelo, por isso é utilizado um padrão de projeto chamado DTO (*Data Transfer Object*).

O DTO ou Objeto de Transferência de Dados dá liberdade para a manipulação e exibição dos dados já que é separado do modelo. Além disso, um objeto não fica sobrecarregado com mais de uma função no sistema e facilita a sua manutenção.

No presente projeto, foram feitos dois DTOs para cada aeromodelo, um de entrada e um de saída. O de entrada contém os dados mostrados na Tabela 1 e é nesse objeto que se faz o controle das obrigatoriedades dos campos na persistência. Como não é possível garantir o recebimento de todos os dados nas competições e nem sempre são coletados todos, o único campo obrigatório é a Data e Hora da Estação Base, pois é possível ter esse valor quando for enviar os dados à API e ele será usado para filtrar os resultados nos gráficos e tabelas. O DTO de saída, o que é mostrado para o usuário nos retornos da API, contém todos os dados do modelo.

Por fim, no terceiro papel do *design, o* controlador é uma classe que recebe e identifica as requisições HTTP de acordo com o verbo utilizado (*GET, POST, DELETE*) e direciona para o método que trata e envia a resposta específica. Cada aeromodelo tem seu controlador específico e todos tem os *endpoints* na Tabela 3 a qual emprega o *cubesat* como exemplo. Na referida Tabela, é possível ver as entradas e saídas para cada requisição, além dos possíveis parâmetros que são usados para filtrar os resultados.

Tabela 3 - *Endpoints* dos aeromodelos.

Verbo HTTP	***Endpoint***	**Parâmetro de caminho**	**Corpo de entrada**	**Reposta HTTP**	**Corpo da saída**
GET	/v1/cube-data	Data e hora de início; Data e hora do final;	Não há	200	Lista de JSON com todos os resultados
GET	/v1/cube-/data/{id}	Não há	Não há	200	JSON do cubo

POST	/v1/cube-data	Não há	JSON de entrada	204	Não há
DELETE	/v1/cube-data	Data e hora de início; Data e hora do final	Não há	204	Não há
DELETE	/v1/cube-data/{id}	Não há	Não há	204	Não há

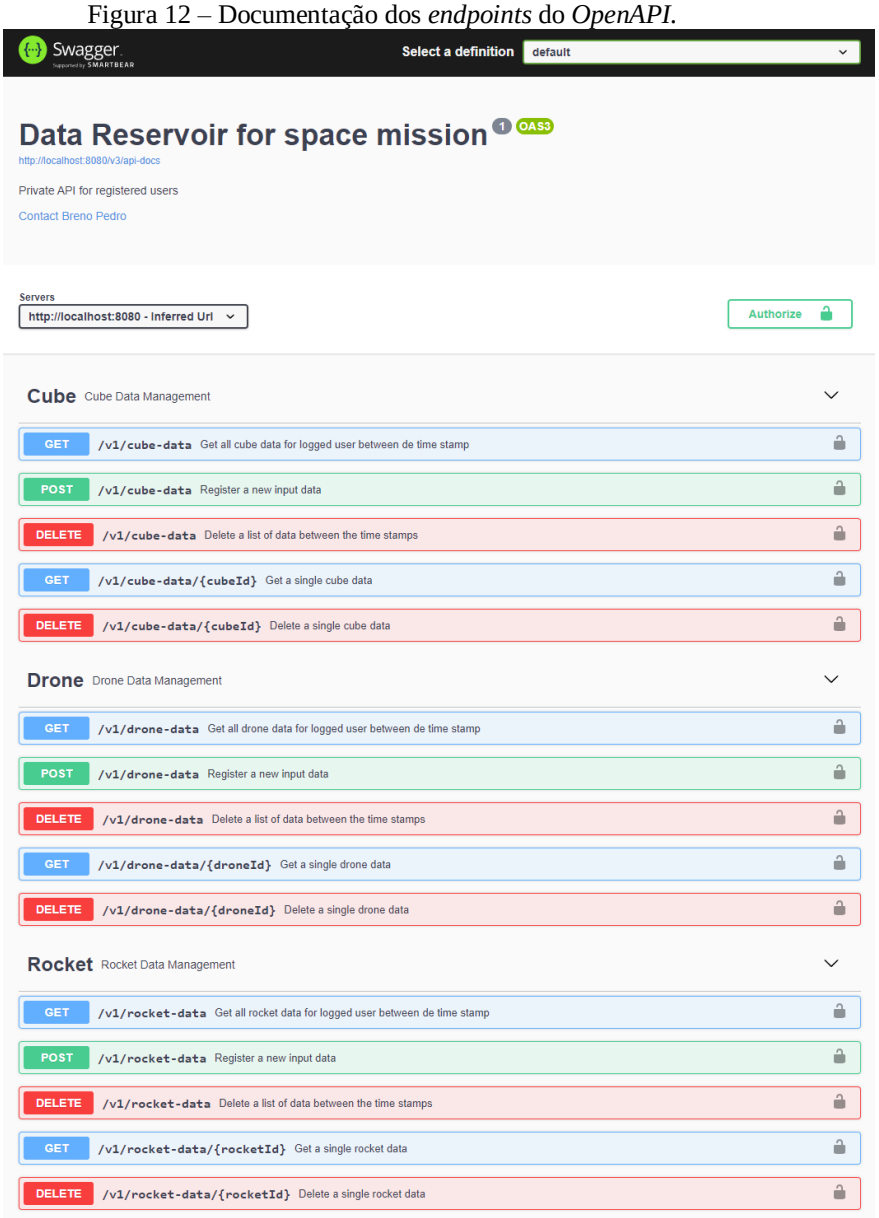
Fonte: Elaborada pelo autor.

O primeiro *endpoint* da Tabela 3 é para consulta de dados. Nele, é possível utilizar alguns filtros para limitar a quantidade de dados retornado, caso não seja usado, retornará todos os dados salvos. A data e hora devem seguir o exemplo 2022-12-10T19:56:21+03:00, que é o indicador universal de data e hora com fuso horário. A API irá retornar apenas os dados inseridos no intervalo daquele filtro.

O segundo *endpoint* utiliza o identificador gerado no banco de dados e retorna apenas aquela dado específico. O terceiro é o *POST* e é através dele que se insere novos valores. Os dois últimos são para deleção e funcionam similares aos de consulta, apagam dados entre um intervalo de tempo caso seja fornecido ou tudo caso contrário e apenas um dado quando é utilizado o identificador do banco.

Todos esses *endpoints* podem ser consultados e testados através da documentação do OpenAPI mostrado na Figura 12.

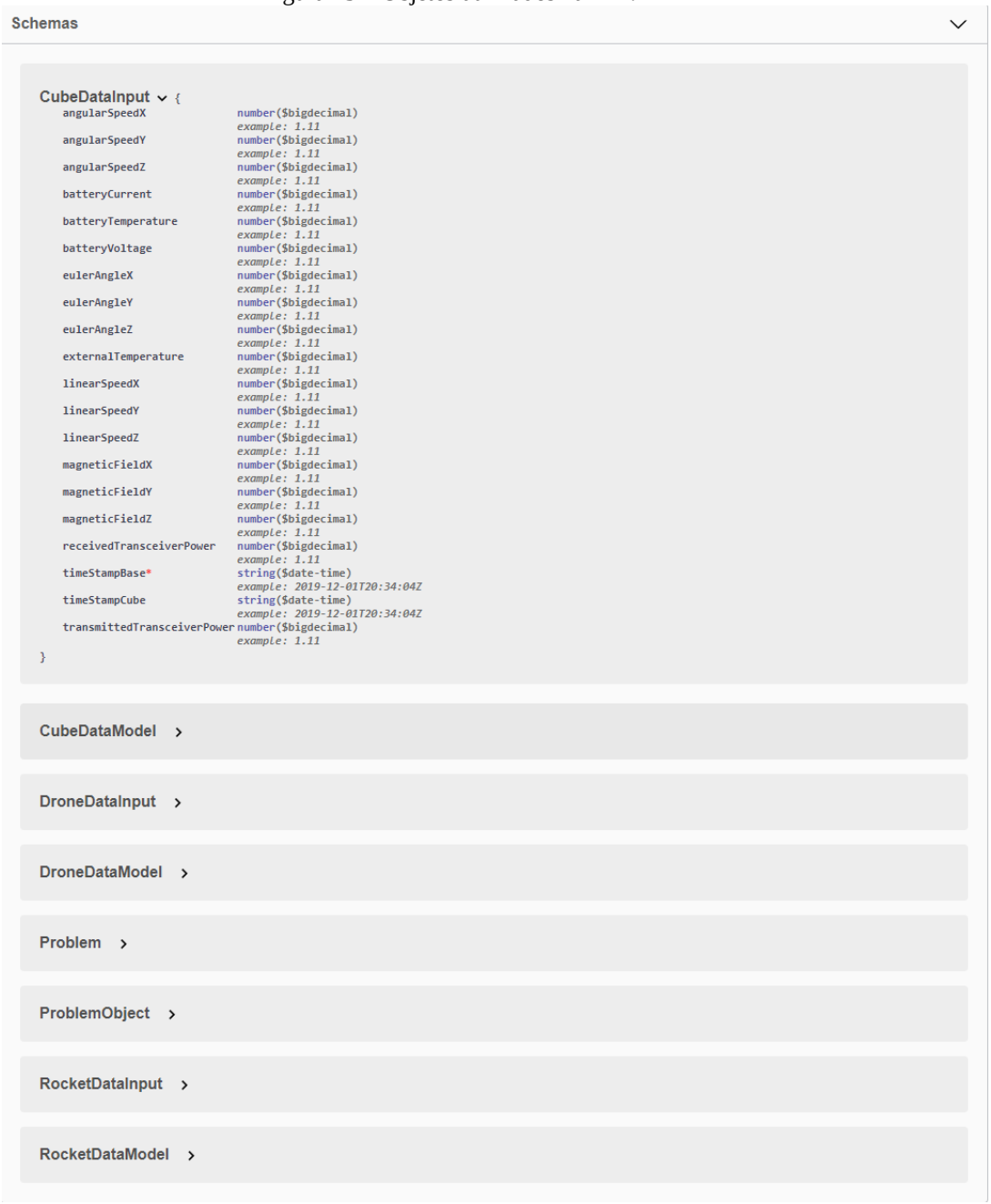
Figura 12 – Documentação dos *endpoints* do *OpenAPI*.



Fonte: Elaborada pelo autor.

Além dos *endpoints*, também é possível ver os dados necessários para adicionar um novo elemento e o que será retornado como mostrado na Figura 13. Também é possível ver os tipos de cada dado e o que são ou não obrigatórios.

Figura 13 – Objetos utilizados na API.

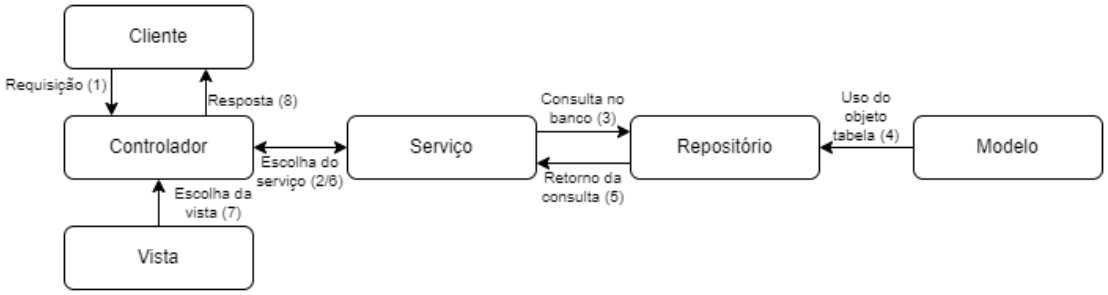


Fonte: Elaborada pelo autor.

Outros objetos importantes que devem ser destacados são as classes de serviço e as interfaces repositório. Ambos fazem parte de outro padrão de projeto chamado DAO, o qual faz a separação das regras de negócio das regras de acesso ao banco de dados. O controlador do MVC não acessa diretamente o banco, mas sim a classe de serviço contendo as especificações para esse acesso, enquanto o controlador mantém as especificações de negócio, como por exemplo qual DTO ser utilizado no retorno do *endpoint*. A interface repositório estende uma outra interface do *Spring JPA* para fazer uso das *queries* mais simples como

mostrado na Tabela 2. Para mostrar visualmente o fluxo das requisições foi elaborada a Figura 14.

Figura 14 - Fluxo dos objetos da API.



Fonte: Elaborada pelo autor.

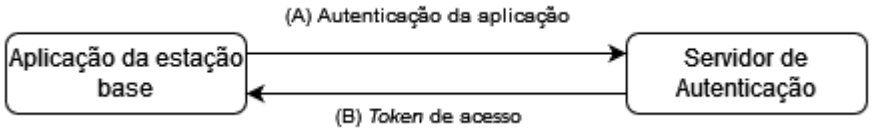
4.2 SEGURANÇA DA API

Depois de feita toda a parte de requisições e respostas da API, é necessário fazer sua segurança com os fluxos do *OAuth*. Para isso, foram criados três perfis de usuários. Um para fazer requisições *POST*, ou seja, armazenar dados; outro para fazer a visualização, ou requisições *GET*; e o último que chamado de admin que pode tanto armazenar quanto visualizar, além de poder deletar dados.

Esses perfis foram criados para o aeromodelo, a interface *web* e fazer a exclusão através do *Postman* ou outra ferramenta que faça requisições HTTP, respectivamente. Seria possível utilizar apenas o usuário admin nos três casos já que ele contém todas as permissões. Cada usuário, no entanto, utiliza um fluxo de autenticação diferente, então foi escolhido fazer três autenticações para cada um dos fluxos.

O primeiro usuário é o da estação base, ele usa o fluxo das credenciais do cliente já que são duas aplicações *backend* se comunicando e não há um usuário físico para ter suas credenciais validadas. A Figura 15 mostra a estação base mandando suas credenciais para o servidor de autenticação e ele retornando o *token* de acesso para que seja possível fazer as requisições na API.

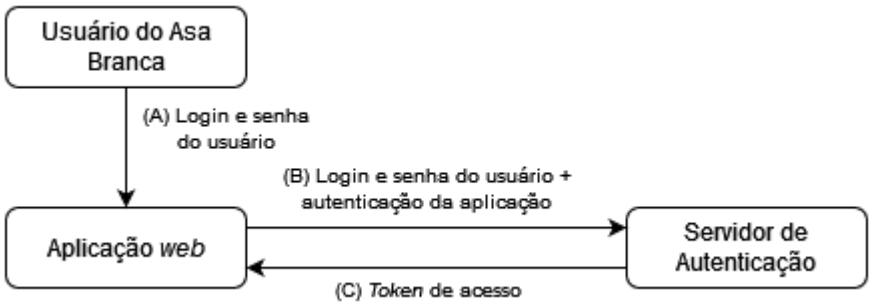
Figura 15 - Fluxo das credenciais do cliente exemplificado.



Fonte: Elaborada pelo autor.

O segundo usuário é o da aplicação *web*, o qual utiliza o fluxo das credenciais do dono do recurso. Assim que se acessa a página *web* é pedido as credenciais do dono do recurso, ou seja, do usuário que está utilizando para ser utilizada no fluxo. Com elas, e com as do cliente que já estão fixas no código (a autenticação da aplicação), pode-se conseguir o *token* para fazer as requisições para a API e visualizar os dados. A Figura 16 ilustra o fluxo começando a partir de algum usuário do Asa Branca inserindo um login e senha na página *web*, em seguida a aplicação utiliza essa informação junto com sua própria autenticação para enviar ao servidor e ser validada; após isso, é retornado o *token* para a visualização dos dados.

Figura 16 - Fluxo da autenticação do dono do recurso exemplificado.



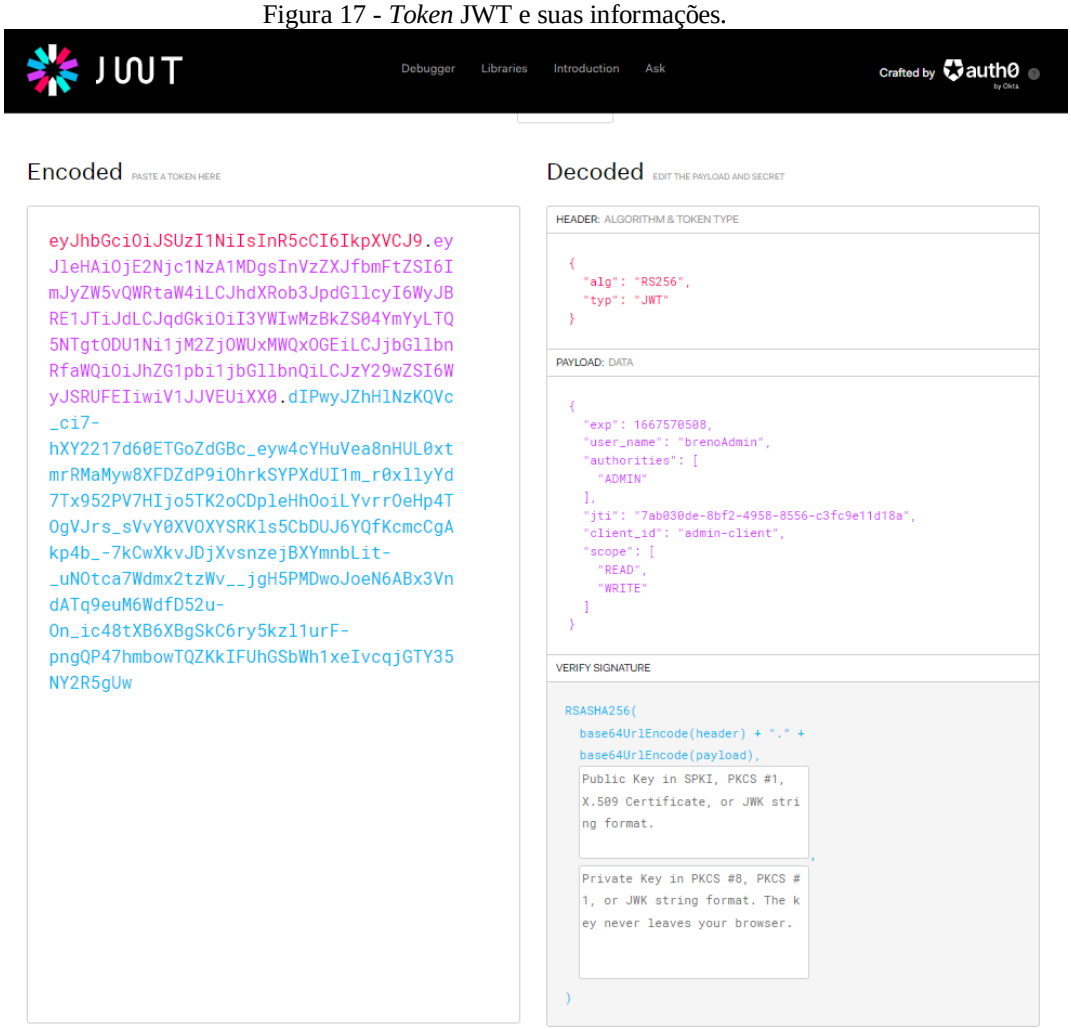
Fonte: Elaborada pelo autor.

O último usuário é o *admin* com todas as permissões e só é possível utilizá-lo em aplicações como o *Postman* para fazer requisições HTTP. Ele foi feito principalmente para excluir dados de teste ou dados incorretos salvos no banco. Assim com o usuário da aplicação *web*, ele utiliza o fluxo das credenciais do dono do recurso.

Esses usuários têm permissões e são com elas que é possível limitar o acesso aos métodos do controlador. Na

Figura 17 é possível ver que essas permissões ficam visíveis no *token* que é facilmente lido em algum tradutor de *token* JWT como é o caso em que foi utilizado o site https://jwt.io. Nesse exemplo, é possível ver que no cabeçalho do *token* foi armazenado as informações do tipo e o algoritmo de criptografia utilizado, sendo esse último o RS256. Ele é o algoritmo RSA, o qual utiliza uma chave pública e outra privada para assinatura e validação do *token*, junto do SHA-256. Ambos os algoritmos são um conjunto de funções matemáticas usadas para criptografar e verificar a integridade do objeto. No corpo do *token* é possível ver o tempo de expiração dele, o dono do recurso, as autoridades, o JTI, o cliente para quem está sendo expedido o *token* e o escopo de acesso. Por último é possível verificar a autenticidade colocando as chaves pública e privada.

Figura 17 - *Token* JWT e suas informações.



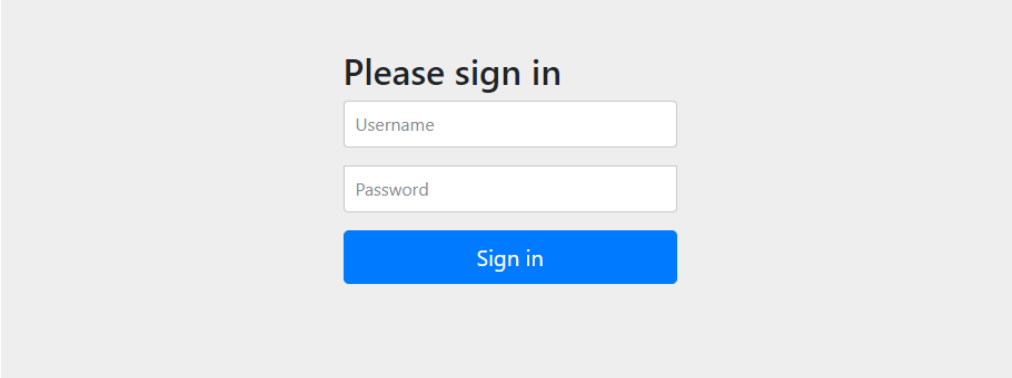
Fonte: https://jwt.io.

Toda a configuração dos usuários foi feita utilizando, ao menos, um dos projetos do ecossistema *Spring;* o *Spring Secuity OAuth* e o *Spring Security JWT*. O primeiro para as implementações dos fluxos de autorização e o segundo para ter suporte a utilização do *tokens* JWT.

4.3 INTERFACE GRÁFICA

Como dito anteriormente, foi utilizado o *Thymeleaf* como *template engine* para fazer as interfaces gráficas. Excetua-se a tela inicial de login que foi gerada automaticamente com o *Spring Security OAuth* mostrada na Figura 18.

Figura 18 - Tela de login da aplicação.

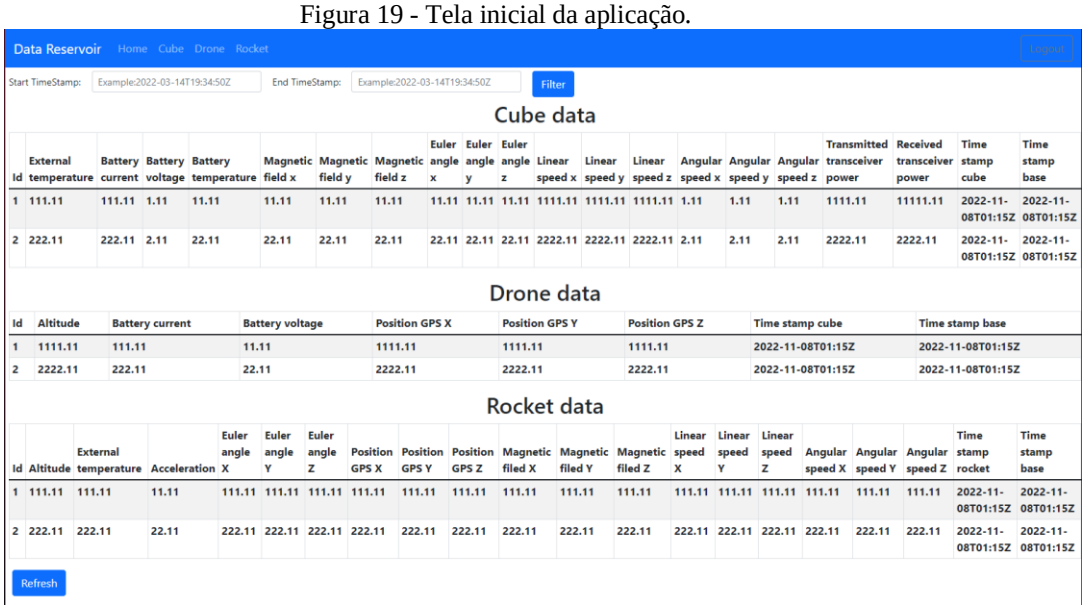


Fonte: Elaborada pelo autor.

Após o login, entra-se na tela inicial onde é mostrado todos os dados dos aeromodelos no formato de tabelas como mostrado na Figura 19. Nessa tela, também é possível fazer um filtro com a data e hora inicial e final para que aparecem dados entre os dados fornecidos, caso não haja nada informado será mostrado tudo. Esse filtro de data deve ser no formato *date-time* definido na RFC 3339 (KLYNE e NEWMAN, 2002): ano-mês-diaThora:minuto:segundoZ ou ano-mês-diaThora:minuto:segundo±hora:minuto. A primeira representação indica que não há deslocamento de hora, ou seja, será considerado o horário de UTC, enquanto a segunda considera algum fuso horário. Para exemplificar esse formato, irá s[3]e considerar o dia 29 de novembro de 2022, às 22h 51min na cidade do Recife que contêm o fuso horário de menos três horas em relação ao UTC logo, a representação seria: 2022-11-29T22:51:00-03:00. Caso considerar no UTC, então ficaria: 2022-11-30T01:51:00Z.

[3] UTC é o horário oficial a partir do qual se calculam os horários das outras zonas horárias do mundo. Em contextos mais informais, o UTC pode ser considerado igual ao GMT (*Greenwich Mean Time*) que é o horário do Meridiano Greenwich, mas há diferenças de frações de segundos.

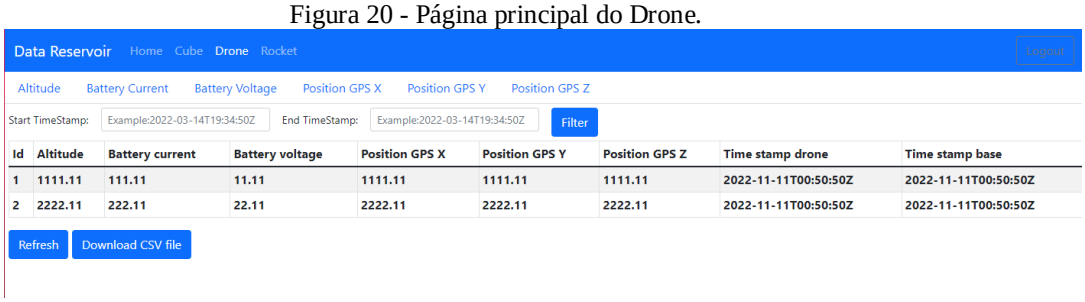
Figura 19 - Tela inicial da aplicação.



Fonte: Elaborada pelo autor.

Na barra de tarefas, acima dos filtros, é possível se deslocar para alguma das outras páginas específicas de cada aeromodelo ou retornar para a página principal através do botão "*Home*". A Figura 20 mostra um exemplo da página do Drone que se assemelha muito aos outros aeromodelos, diferenciando na quantidade de abas dos dados. Nessa página é possível filtrar pela data, assim como na principal, e baixar um arquivo CSV dos dados que estão sendo mostrados.

Figura 20 - Página principal do Drone.



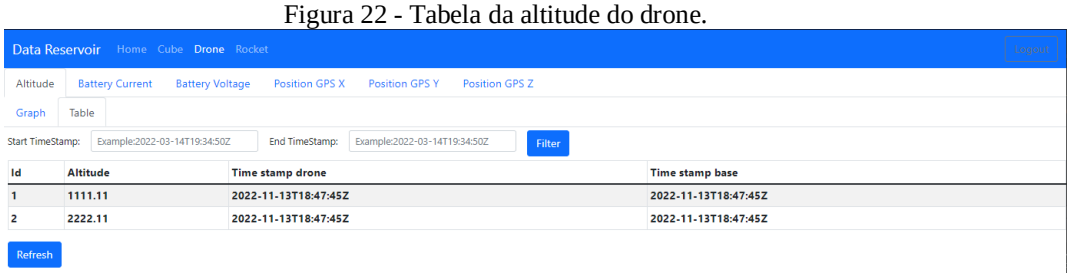
Fonte: Elaborada pelo autor.

Além disso, também é possível ir para as abas individuais de cada dado contendo um gráfico e uma tabela como mostrado na Figura 21. Nela, é possível ver a altitude do Drone em forma de gráfico. Enquanto na Figura 22 é mostrado a tabela.

Figura 21 - Gráfico da altitude do drone.

Fonte: Elaborada pelo autor.

Figura 22 - Tabela da altitude do drone.



Fonte: Elaborada pelo autor.

Todas as páginas têm o filtro para ser usado e todas atualizam automaticamente em um minuto ou pode ser feito manualmente com o botão de atualizar que fica ao final. Como o *Thymeleaf* é renderizado do servidor, e não no cliente, é necessário recarregar a página por completo para receber as atualizações dos novos dados caso existam.

4.4 *"DOCKERIZANDO"* A APLICAÇÃO

Para fazer a imagem da aplicação foi utilizado um *plugin* integrado ao *Maven* capaz de integrar as imagens descritas em um arquivo YAML e conectá-las. O arquivo tem o nome de "docker-compose.yml" e nele são declaradas variáveis gerais que as imagens precisam quando forem serem executadas nos containers, como por exemplo a porta em que a aplicação disponibilizará para ser acessada.

Foram usadas imagens *docker* que são representações estáticas do *software*, sua configuração e dependências. Ao total foram usadas quatro imagens: *mysql* para o banco de dados, *redis* para armazenamento de memória cache da sessão do usuário, a imagem da aplicação desenvolvida e o *nginx* para o *proxy* reverso.

A imagem do *mysql* é a oficial que se pode encontrar no *DockerHub*, uma comunidade que disponibiliza imagens *docker* para uso público, mantidos pela comunidade ou pela própria empresa criadora do *software*. Nesse caso, é mantida pela própria empresa que a criou. Na descrição do *docker-compose* foi declarada a porta disponível para uso e a senha do usuário. A imagem do *redis* não precisa de qualquer configuração, pois a conexão para ele é feita nas variáveis da aplicação desenvolvida.

Um p*roxy* reverso funciona como um servidor, mas que não tem autonomia para responder as requisições por si mesmo, então as encaminha para outro lugar, que no caso será a aplicação desenvolvida. Isso é necessário porque quando é feito o *docker*, não é possível acessá-lo "de fora", ou seja, apenas com uma aplicação sendo executada dentro do próprio container é possível fazer as requisições HTTP para a API. A imagem do *nginx* irá expor uma porta para que seja possível acessar de qualquer lugar e irá encaminhar as requisições para a aplicação desenvolvida.

A última imagem é a da própria aplicação e no arquivo *docker-compose* são feitas várias declarações de variáveis como: a porta a ser executada, os usuários e senhas e o tipo de armazenamento da sessão do usuário. Além dessas configurações de variáveis, também é preciso indicar a rede que todos esses containers serão executados, pois uma rede *docker* é uma abstração criada para facilitar o gerenciamento da comunicação de dados entre containers e os nós externos ao ambiente *docker*, ou seja, para que eles possam se comunicar, é preciso que estejam na mesma rede.

4.5 *DEPLOY* DO *DOCKER* PARA A AWS

A AWS oferece serviços gratuitos com alguns limites no primeiro ano de assinatura, os referidos serviços são aqui utilizados. Eles são:

- RDS (*Relation Database System*): serviço de banco de dados;
- ECR (*Elastic Container Registry*): gerenciador das imagens dos containers;
- AWS *System Manager:* declaração de variáveis usadas pelo *software;*
- VPC (*Virtual Private Cloud*): gerenciador dos recursos isolados;
- ECS (*Elastic Conteiner Service*): criação de serviços para que as imagens sejam executadas.

Além dos serviços da AWS, também será necessário utilizar a nuvem do *Redis*, o *RedisLabs*. A primeira configuração deve ser feita no RDS, para criar o banco de dados onde serão armazenados os dados coletados. Nela é possível definir a porta usada, verificar os *logs* do banco, monitoramento de acessos e verificar os *backups*. Além disso, também é possível ver o *endpoint* para acessar o banco através do *workbench* instalado junto ao *mysql*. e criar o usuário utilizado pela aplicação.

É na criação do banco que é possível criar, caso não exista, a VPC da conta da AWS. Ela normalmente é criada automaticamente na criação da conta, mas em raras exceções não é. Será preciso selecionar uma VPC para a criação do banco e caso já exista uma, ela aparecerá nas opções, caso não, há uma opção de criar uma nova.

Na criação do banco no RDS, é feita a configuração de um grupo de segurança dentro da AWS e é preciso configurar a regra de entrada dele para que seja possível entrar através do *workbench*. Para isso, é preciso acessar a VPC, onde é possível ver as configurações dentro da AWS, ir para os grupos de segurança e adicionar o IP local da máquina nas regras de entrada do grupo utilizado pelo *mysql*.

Depois de feita essa configuração, é factível acessar o banco pelo *workbench* e criar o usuário que será utilizado na aplicação e definindo suas permissões. No presente trabalho, foi criado um usuário com todas as permissões de modo a validar os experimentos, e só terá acesso o IP definido no grupo de segurança na VPC.

Após a criação do usuário no banco de dados, é preciso subir a imagem do container usando o serviço ECR. Nele, é preciso criar um repositório e defini-lo como sendo privado ou público: o privado limita o acesso ao repositório através da ferramenta permissões da AWS e a pública será visível para todos e poderão fazer uma cópia da imagem. Nesse caso, foi criado um repositório privado. Além disso, também é preciso escolher um nome para ele.

Logo em seguida da criação do repositório, fica disponível a visualização dos comandos *push*, ou seja, os comandos necessários para subir a imagem no servidor. O passo a passo é bem estruturado, os comandos podem ser inseridos no *PowerShell*, no caso do *Windows*, ou no terminal de comando nos casos do *Linux* e *macOS*. Após a execução, a imagem se torna visível no repositório e poderá ser usada pelos outros serviços da AWS.

Faz-necessário criar uma conta no *RedisLabs* para que se possa utilizar sem que haja nenhum dado de pagamento. No plano gratuito, é possível criar apenas um banco com um armazenamento de 30 MB, o que será suficiente para a aplicação. Para criar um banco é necessário apenas criar um nome e armazenar a senha fornecida na hora da criação, pois ela

será usada no *SystemManager* na AWS. Após a criação, será fornecido o *endpoint* para acesso do banco.

Não é de bora prática deixar as variáveis utilizadas no código no arquivo *properties*, pois elas facilmente visíveis para alguém que tenha acesso e nesse arquivo que normalmente ficam armazenadas senhas e outros variáveis importantes como chaves privadas. Por isso, a AWS fornece um serviço em que é possível colocar essas variáveis em local para que o container possa utilizar e não ficar expostas. É o *AWS System Manager*, na opção de Armazenamento de parâmetros. Cria-se parâmetros com um nome em específico, pode-se adicionar uma descrição para ficar mais fácil a identificação, o tipo: *string* para textos comuns, *stringList* para uma lista de textos *e secureString* que é uma *string* mas normalmente usado para senhas já que fica criptografado. Serão nessas variáveis que a senha e o *endpoint* do *RedisLabs* serão colocados.

O ECS gerencia os serviços que executam os contêineres, mas para isso é preciso criar um *cluster*, que é um aglomerado lógico de serviços ou tarefas. Na Figura 23 é mostrado a primeira tela para a criação de um *cluster*. Nela, pode-se ler várias opções para criação e a escolhida será a “Somente redes”, pois não será usado o EC2 (*Elastic Compute Cloud*) em nenhum momento do presente experimento. O EC2 é um serviço para o aluguel de computadores virtuais, o que não é algo necessário para essa aplicação. O principal produto usado na opção escolhida será o *Fargate*, trata-se de um mecanismo de computação sem servidor e com pagamento conforme o uso. Ele é compatível com o ECS, o gerenciador de serviços dos contêineres. Após isso, é preciso apenas escolher um nome para o cluster e ele será criado.

Figura 23 - Criação do *cluster*.

Fonte: Amazon (SERVICES, 2023).

Após a criação do *cluster*, faz-se necessário criar uma definição de tarefa também dentro do ECS. É escolhida a opção de *fargate*, a qual é constituída das seguintes configurações;

- Criação de um nome;
- Escolha do *Linux* como família de sistemas operacionais;
- Alocação de 1 GB de memória usada pela tarefa, pois não é uma aplicação que necessita de muita e quanto mais alocada maior será o preço pago;
- A CPU da tarefa será 0,5 vCPU (*Virtual Centralized Processing Unit*);
- Para selecionar um container é preciso dar um nome para ele (que pode ser qualquer um, não necessariamente o dado na subida da imagem);
- URI do container, que é possível ser visualizada no ECR;
- Seleção da opção de limite flexível de memória e colocar a porta como sendo a 8080;
- Configuração das variáveis contidas no arquivo *.properties*. Os nomes precisam estar em letras maiúsculas e separadas por *underscore* (_), enquanto os valores serão acessados do *SystemManager* através dos nomes definidos nesse serviço.

Com a criação da definição de tarefa criada, deve-se providenciar o serviço para que essa tarefa seja executada. A referida providência também é feita no ECS e é preciso indicar um nome para o serviço e selecionar a definição de tarefa criada anteriormente. Além disso,

deve-se indicar a quantidade de tarefas a serem executadas, ou seja, quantas instâncias o serviço executará, ao mesmo tempo, a tarefa escolhida.

Para o escopo desse projeto, será feito com apenas uma instância de tarefa por questão de custo, já a execução de mais de uma aumentaria o custo financeiro do presente experimento. É indicado uma aplicação ter mais de uma instância de modo a haver tolerância a falhas. Caso uma instância saia do ar, as demais continuariam prestando serviço. Várias instâncias de uma mesma aplicação também seriam úteis para balanceamento de carga e de tráfego de acesso sem que uma(s) instância(s) ficasse(m) sobrecarregada(s). Também como vantagem, o emprego de várias instâncias reduziria o tempo de resposta da aplicação. Dentre as instâncias, seria escolhida aquela geograficamente mais próxima do *host* de modo a otimizar o tempo de resposta ao serviço. O presente experimento contém uma única instância. Trata-se de uma aplicação que será usada em momentos específicos e não terá acessos em larga escala.

5 RESULTADOS

Nas seções seguintes serão apresentados os resultados os testes feitos com a aplicação autoral na nuvem da AWS. Foram feitos dois testes: um teste de estresse para grandes quantidades de requisições e um teste de usabilidade da interface com integrantes e ex-integrantes do Asa Branca.

5.1 TESTE DE ESTRESSE

O código foi testado ao se criar uma aplicação em Java dotada de API. Na sequência, faz-se sucessivas gravações de dados. Foi utilizado como teste os dados do *CubeSat*[4], por ter mais campos a serem gravados, que estão na Tabela 1. Foram feitos alguns testes como mostrado na Tabela 4.

Tabela 4 - Tempos de respostas de requisições feitas à aplicação na nuvem.

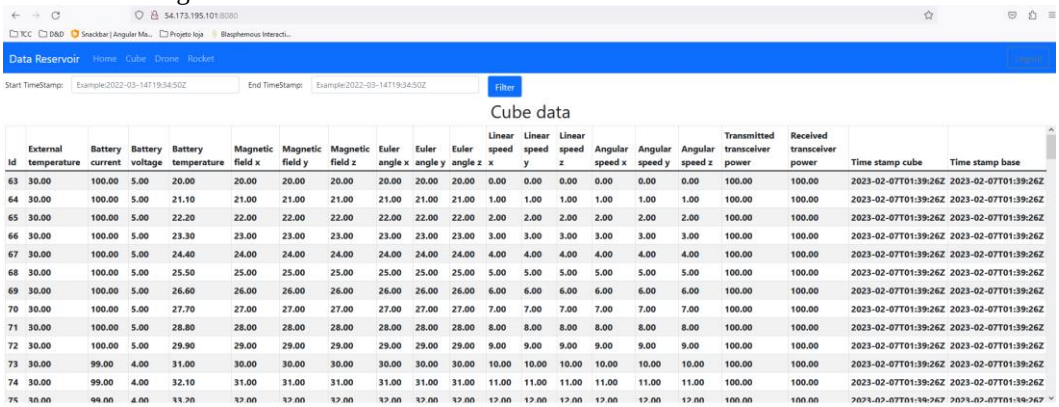
Quantidade de requisições	**Tempo de resposta sem o *token***	**Tempo total**
20	4 segundos	5 segundos
100	15 segundos	16 segundos
200	27,5 segundos	28 segundos
500	63,6 segundos	64 segundos
1000	123.6 segundos	124 segundos

Fonte: Desenvolvida pelo autor.

Os tempos de resposta não cresceram proporcionalmente ao aumento do número de requisições entre os testes. Em termos técnicos, o tempo de resposta não cresce de forma linear à medida que há um aumento das requisições. Logo é possível ver uma otimização para grandes números de requisições da plataforma autoral e intermediária AWS. A visualização dos gráficos e tabelas também foi possível com os dados adicionados como é mostrado na Figura 24.

[4] CubeSat: satélite em forma de cubo.

Figura 24 - Dados do cubo salvos na AWS mostrados na tabela.



Id	External temperature	Battery current	Battery voltage	Battery temperature	Magnetic field x	Magnetic field y	Magnetic field z	Euler angle x	Euler angle y	Euler angle z	Linear speed x	Linear speed y	Linear speed z	Angular speed x	Angular speed y	Angular speed z	Transmitted transceiver power	Received transceiver power	Time stamp cube	Time stamp base
63	30.00	100.00	5.00	20.00	20.00	20.00	20.00	20.00	20.00	20.00	0.00	0.00	0.00	0.00	0.00	0.00	100.00	100.00	2023-02-07T01:39:26Z	2023-02-07T01:39:26Z
64	30.00	100.00	5.00	21.10	21.00	21.00	21.00	21.00	21.00	21.00	1.00	1.00	1.00	1.00	1.00	1.00	100.00	100.00	2023-02-07T01:39:26Z	2023-02-07T01:39:26Z
65	30.00	100.00	5.00	22.20	22.00	22.00	22.00	22.00	22.00	22.00	2.00	2.00	2.00	2.00	2.00	2.00	100.00	100.00	2023-02-07T01:39:26Z	2023-02-07T01:39:26Z
66	30.00	100.00	5.00	23.30	23.00	23.00	23.00	23.00	23.00	23.00	3.00	3.00	3.00	3.00	3.00	3.00	100.00	100.00	2023-02-07T01:39:26Z	2023-02-07T01:39:26Z
67	30.00	100.00	5.00	24.40	24.00	24.00	24.00	24.00	24.00	24.00	4.00	4.00	4.00	4.00	4.00	4.00	100.00	100.00	2023-02-07T01:39:26Z	2023-02-07T01:39:26Z
68	30.00	100.00	5.00	25.50	25.00	25.00	25.00	25.00	25.00	25.00	5.00	5.00	5.00	5.00	5.00	5.00	100.00	100.00	2023-02-07T01:39:26Z	2023-02-07T01:39:26Z
69	30.00	100.00	5.00	26.60	26.00	26.00	26.00	26.00	26.00	26.00	6.00	6.00	6.00	6.00	6.00	6.00	100.00	100.00	2023-02-07T01:39:26Z	2023-02-07T01:39:26Z
70	30.00	100.00	5.00	27.70	27.00	27.00	27.00	27.00	27.00	27.00	7.00	7.00	7.00	7.00	7.00	7.00	100.00	100.00	2023-02-07T01:39:26Z	2023-02-07T01:39:26Z
71	30.00	100.00	5.00	28.80	28.00	28.00	28.00	28.00	28.00	28.00	8.00	8.00	8.00	8.00	8.00	8.00	100.00	100.00	2023-02-07T01:39:26Z	2023-02-07T01:39:26Z
72	30.00	100.00	5.00	29.90	29.00	29.00	29.00	29.00	29.00	29.00	9.00	9.00	9.00	9.00	9.00	9.00	100.00	100.00	2023-02-07T01:39:26Z	2023-02-07T01:39:26Z
73	30.00	99.00	4.00	31.00	30.00	30.00	30.00	30.00	30.00	30.00	10.00	10.00	10.00	10.00	10.00	10.00	100.00	100.00	2023-02-07T01:39:26Z	2023-02-07T01:39:26Z
74	30.00	99.00	4.00	32.10	31.00	31.00	31.00	31.00	31.00	31.00	11.00	11.00	11.00	11.00	11.00	11.00	100.00	100.00	2023-02-07T01:39:26Z	2023-02-07T01:39:26Z
75	30.00	99.00	4.00	33.20	32.00	32.00	32.00	32.00	32.00	32.00	12.00	12.00	12.00	12.00	12.00	12.00	100.00	100.00	2023-02-07T01:39:26Z	2023-02-07T01:39:26Z

Fonte: Elaborada pelo autor.

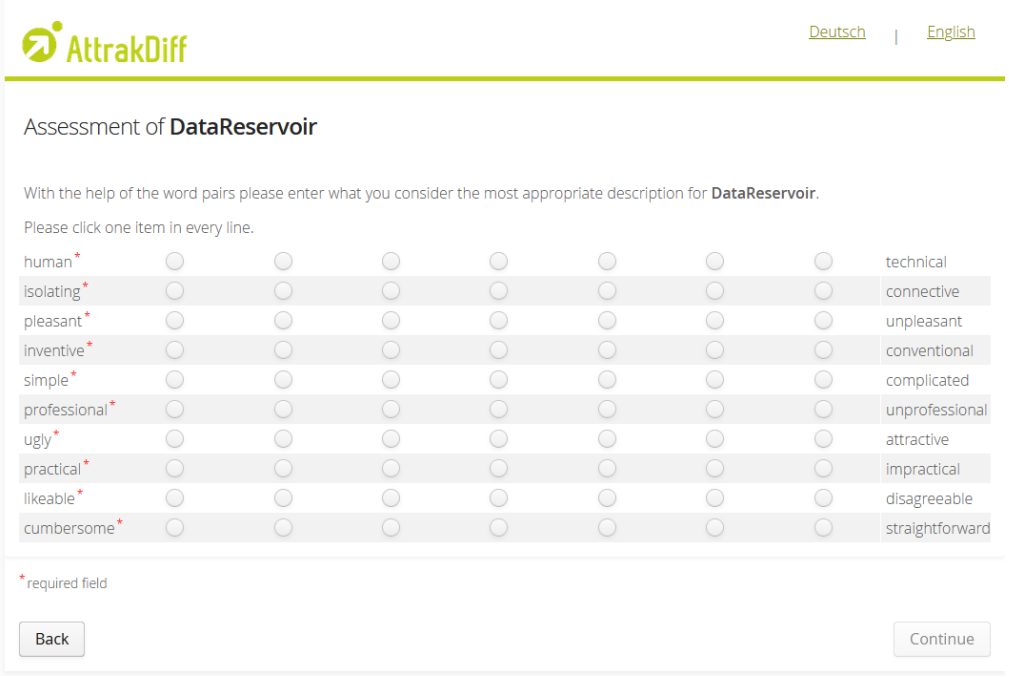
Logo com um acesso à Internet estável, o armazenamento dos dados na nuvem será feito com poucos atrasos e será visível nos gráficos e tabelas em tempo real. Além de ser possível a análise e leitura deles a qualquer momento.

5.2 TESTE DE USABILIDADE

O teste de usabilidade da interface gráfica foi feito da seguinte forma: foi pedido aos participantes para acessarem o link da aplicação e explorarem o site, sem indicações do que deveriam fazer; após isso, foi solicitado que respondessem dois questionários, um do *AttrakDiff* e outro no *GoogleForms*. O teste foi feito com 15 participantes, integrantes e ex-integrantes do Asa Branca, ao longo de 7 dias.

O *AttrakDiff* é um questionário desenvolvido para avaliar a usabilidade e *design* de produtos interativos. Para medir a atratividade, ele utiliza um formato com diferenciais semânticos. Ele é formado por vinte e oito polos com adjetivos opostos e usa uma escala de sete valores, cada conjunto de polos é ordenado em uma escala de intensidade. Um exemplo é mostrado na Figura 25.

Figura 25 - Exemplo dos polos do *AttrakDiff*.



Fonte: *AttrakDiff* (AttrakDiff).

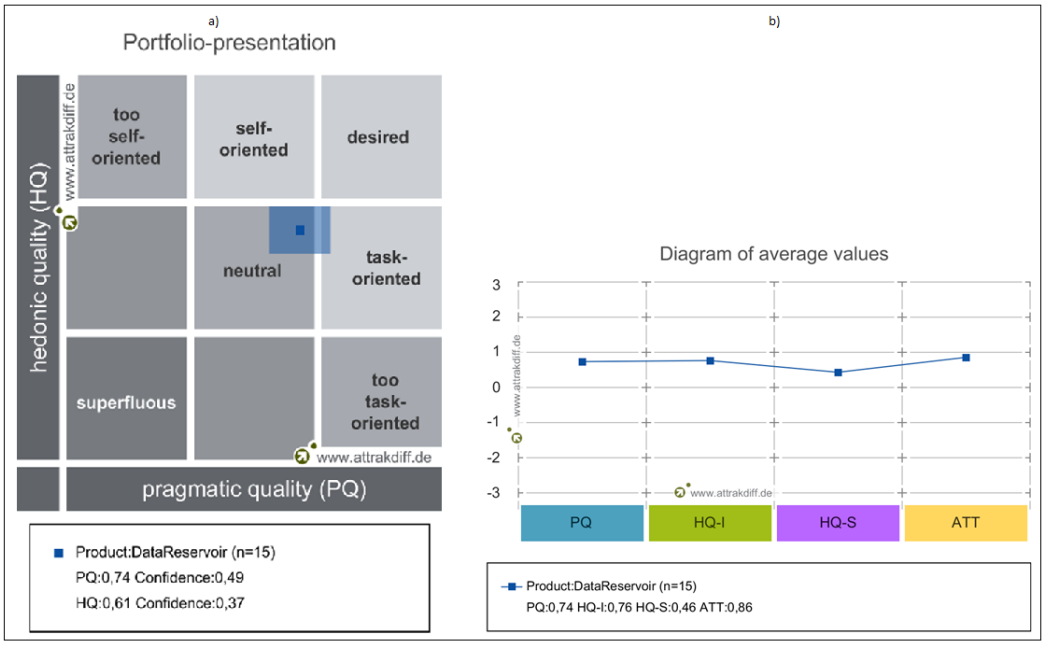
O questionário agrupa os adjetivos em quatro grupos avaliativos:

- Qualidade Pragmática (PQ): Descreve a usabilidade de um produto e indica como os usuários alcançam seus objetivos ao usar o produto;
- Qualidade Hedônica – Estimulação (HQ-S): Essa dimensão explora o potencial evolutivo do produto. Ela indica até que ponto o produto pode suportar as necessidades dos usuários em termos de inovação, interesse e funções estimulantes, conteúdo e estilos de interação e apresentação;
- Qualidade Hedônica – Identidade (HQ-I): Essa dimensão indica até que ponto o produto permite ao usuário se identificar com ele;
- Atratividade (ATT): Representa através de um valor global a qualidade do produto percebida pelo usuário.

Os resultados são mostrados nas Figura 26 a) e na Figura 26 b), é possível observar o Portifólio de apresentação com coeficientes de confiabilidade da PQ igual a 0,49 e da Qualidade Hedônica (HQ) geral sendo 0,37. Isso indica uma alta confiabilidade nos resultados, pois quanto menor a área em azul claro, mais consistente foram as respostas dos participantes da pesquisa. Os valores da PQ e HQ geral são 0,74 e 0,61, respectivamente, colocando o resultado na zona neutra e orientada à tarefa. Tal fato denota um produto bonito, mas não muito intuitivo, precisando melhorar em sua usabilidade.

Na Figura 26 b) é possível ver o diagrama de valores médios. Nele, observa-se os valores individuais de cada grupo avaliativo. Quanto aos grupos avaliativos, a ferramenta autoral obteve PQ, HQ-I, HQ-S e ATT iguais à 0,74, 0,76, 0,46 e 0,86, respectivamente. Todos os valores ficaram na zona média, entre os valores 0 e 1, ou seja, atendem a necessidade do usuário, mas há possibilidades de melhora em todos os aspectos, tanto pragmática, como hedônica.

Figura 26 - a) Portifólio de resultados. b) Diagrama da média de valores.



Fonte: Gerado pelo *AttrakDiff* (AttrakDiff).

O segundo questionário, no *GoogleForms*, foi focado em saber o que poderia ser melhorado, deixando o participante livre para responder como quisesse. A principal sugestão foi em relação aos gráficos e na forma de filtrar os dados, pois segundo um dos participantes:

> "*em certas partes como na leitura dos gráficos fica bem poluído e a informação no eixo horizontal (timestamp) fica muito difícil de ler*."

Logo é preciso rever o *design* dos gráficos e a forma de se filtrar para facilitar a visualização e diminuir a poluição visual quando há muitos dados para serem apresentados. Como aspecto positivo, os participantes do teste atestaram que a experiência vivenciada atendeu as expectativas.

O teste de usabilidade assume papel importante no sentido de analisar o impacto causado por futuras versões da presente ferramenta. Determinadas alterações podem ocasionar

em problema(s) mais grave(s) do que o(s) original(is). Portanto, o controle da usabilidade da ferramenta é fundamental mesmo em um momento de recém lançamento.

Espera-se que ao longo do ingresso de novos participantes na equipe Asa Branca, testes de usabilidade sejam realizados de forma contínua na ferramenta autoral. O objetivo é sanar as limitações atuais sem que haja a degradação das virtudes apontadas pelos participantes do teste de usabilidade inicial.

6 CONCLUSÃO

APIs estão sendo muito utilizadas em diversos sistemas. O desenvolvimento de APIs é fundamental pois facilita o acesso a serviços em nuvens. O emprego de APIs em grandes *datacenters* é um marco que sinaliza a sua importância em tempos contemporâneos. O presente desenvolvimento de um API autoral foi feito através de consultas em documentações oficiais, utilizando-se as melhores práticas e dotado um código-fonte estruturado.

Como qualquer *software* é necessário fazer atualizações, pois linguagens de programação estão constantemente se atualizando e práticas que antes eram consideradas "boas", não são mais. Além de projetos que são descontinuados e não serão mais possíveis de serem utilizados. Nesse contexto, atualizar o código-fonte em consonância com possíveis atualização das documentações das linguagens é importante. O objetivo é manter o funcionamento da aplicação e melhorar a sua performance, sua usabilidade, além de adicionar novas funcionalidades.

Por se tratar de um primeiro projeto, o Asa Branca foi o projeto escolhido como projeto a receber a API autoral. Pôde-se levantar informações *in loco* de modo a se confeccionar uma aplicação mais ajustada possível. A meta é que API autoral possa ser útil nos requisitos e nas competições pleiteadas pelo projeto Asa Branca.

Como trabalhos futuros:

- Atualizar o *template engine* por um *framework* como o Angular para maior desacoplamento dos sistemas de *back* e *frontend*;
- Atualizar as bibliotecas de documentação da *OpenAPI*, pois já foi notificada a sua descontinuação;
- Fazer um desacoplamento do Servidor de Autenticação e do Servidor de Recursos para melhor manutenção do código;
- Fazer a atualização da biblioteca de autenticação devido a sua separação em duas: uma para o Servidor de Autenticação e outra para o de Recursos;
- Melhorar as páginas para diminuir a poluição visual quando há muito dados.
- Criar a política de contínuos testes de usabilidade na API autoral mediante o ingresso de novos participantes na equipe Asa Branca.

7 REFERÊNCIAS

AEROSPACE, A. B. **EDITAL DO PROCESSO SELETIVO 2022**. [S.l.]. 2022.

ALVESTRAND, H. **RFC 3935: A Mission Statement for the IETF**. [S.l.]. 2004.

AMAZON. O que é Conteinerização? Disponivel em: <https://aws.amazon.com/pt/what-is/containerization/>.

AMAZON. O que é um API? Disponivel em: <https://aws.amazon.com/pt/what-is/api/>.

ATTRAKDIFF. **AttrakDiff**. Disponivel em: <https://www.attrakdiff.de/>.

BRAY, T. **RFC 7159: The JavaScript Object Notation (JSON) Data Interchange Format**. [S.l.]. 2015.

FIELDING, R. **Architectural Styles and the Design of Network-based Software Architectures**. University of California. Irvine. 2000.

FOWLER, M. **Padrões de Arquitetura de Aplicações Corporativas**. Porto Alegre: bookman, 2006.

FOWLER, M. Richardson Maturity Model, Março 2010. Disponivel em: <https://martinfowler.com/articles/richardsonMaturityModel.html>.

GIERKE, O. et al. **Spring Data JPA - Reference Documentation**. [S.l.]. 2023.

HARDT, D. **RFC 6749: The OAuth 2.0 Authorization Framework**. [S.l.]. 2012.

INC, D. Docker Documentation, 2022. Disponivel em: <https://docs.docker.com/>.

JONES, M.; BRADLEY, J.; SAKIMURA, N. **RFC 7515: JSON Web Signature (JWS)**. [S.l.]. 2015.

JONES, M.; BRADLEY, J.; SAKIMURA, N. **RFC 7519: JSON Web Token (JWT)**. [S.l.]. 2015.

JONES, M.; HILDEBRAND, J. **RFC 7516: JSON Web Encryption (JWE)**. [S.l.]. 2015.

JWILDER. jwilder/nginx-proxy, 2023. Disponivel em: <https://hub.docker.com/r/jwilder/nginx-proxy>.

KLYNE, G.; NEWMAN, C. **RFC 3339: Date and Time on the Internet: Timestamps**. [S.l.]. 2002.

LTD, R. Redis Enterprise Cloud, Dezembro 2022. Disponivel em: <https://docs.redis.com/latest/rc/>.

SERVICES, A. W. AWS Documentation, 2023. Disponivel em: <https://docs.aws.amazon.com/index.html>.

TANG, X. **The Ethics of Data management**. The Pennsylvania State University. [S.l.].

WEBB, P. et al. **Spring Boot Reference Documentation**. [S.l.]. 2023.