PM

Implementação de Mecanismos de API REST no Projeto GesFoGO e na Plataforma Low-code DISME

PROJETO DE MESTRADO

José Valentim Roseira Caires
MESTRADO EM ENGENHARIA INFORMÁTICA

UNIVERSIDADE da MADEIRA
A Nossa Universidade
www.uma.pt

setembro | 2024

Implementação de Mecanismos de API REST no Projeto GesFoGO e na Plataforma Low-code DISME

PROJETO DE MESTRADO

José Valentim Roseira Caires
MESTRADO EM ENGENHARIA INFORMÁTICA

ORIENTAÇÃO
David Sardinha Andrade de Aveiro

Faculdade de Ciências Exatas e da Engenharia

Mestrado em Engenharia Informática

Implementação de Mecanismos de API REST no Projeto *GesFoGO* e na Plataforma *Low-code* DISME

José Valentim Roseira Caires

Orientado por:

David Sardinha Andrade de Aveiro

Constituição do júri de provas públicas:

Karolina Baras (Professora Auxiliar da Universidade da Madeira), Presidente

Fábio Rúben Silva Mendonça (Professor Auxiliar Convidado da Universidade da Madeira), Arguente

10 de dezembro de 2024

Resumo

Na gestão de dados organizacionais, é crucial o desenvolvimento de APIs REST para a integração com diversos sistemas de informação. Esta necessidade leva, frequentemente, à realização de tarefas repetitivas, trabalhosas e demoradas. As plataformas *low-code/no-code* vêm transformar significativamente este panorama, viabilizando a geração rápida/automática de APIs para o envio e receção de dados e execução de ações operacionais.

O trabalho desenvolvido no âmbito desta tese começa com a realização do Projeto *GesFoGO*, em parceria com a *Universidade de Las Palmas de Gran Canaria* (ULPGC) e com a colaboração de várias outras entidades oficiais, como Serviço Nacional de Proteção Civil e Bombeiros de Cabo Verde, Instituto das Florestas e da Conservação da Natureza da Madeira (IFCN) e Governos. O *GesFoGO* promove a cooperação entre organizações com capacidade operacional para a criação de uma rede abrangente de prevenção e gestão de incêndios florestais em tempo real, através de unidades móveis de rápida implantação com sistema auto-georreferenciado, para alcançar uma gestão sustentável em ambientes florestais montanhosos, característicos dos territórios de cooperação. A implementação deste projeto envolveu a criação de uma aplicação web que inclui a modelação da base de dados, a implementação de uma componente servidora (API REST) para acesso aos dados e adição de mecanismos de segurança e ainda, a interface gráfica (GUI) com integração de sistemas de georreferenciamento. Após integração com hardware, desenvolvido pela ULPGC, foram realizados testes no terreno e demonstrações às entidades oficiais que confirmaram o correto funcionamento de todo o sistema e não identificaram nenhuma informação em falta.

Depois foi implementada a API REST para um outro projeto de investigação, o *MiColEC*, que apresenta um novo paradigma logístico que promove a colaboração entre empresas de entregas *last-mile*, através da partilha de meios e recursos.

A colaboração nestes projetos, sobretudo no que diz respeito às APIs REST, contribuíram para a aquisição de conhecimentos e valências nessa temática. Com base nessas competências adquiridas, foi proposta a adição de mecanismos de criação e gestão de APIs REST à plataforma *low-code* DISME (*Dynamic Information System Modeller and Executer*), que está a ser desenvolvida na ARDITI pela equipa do EELab (*Enterprise Engineering Laboratory*). Esta abordagem tem como principal objetivo tornar o desenvolvimento de APIs REST mais rápido e mais fácil, eventualmente sem a necessidade de competências em programação. Mais concretamente, permitir a geração de *endpoints* para acessos simples de leitura, resultados de *queries* complexas e criação de recursos, tirando partido da metodologia DEMO (*Design and Engineering Methodology for Organizations*) e do DISME, através de operações de *drag-and-drop* numa interface gráfica intuitiva.

Alguns dos resultados obtidos durante a realização destes trabalhos constam em artigos científicos submetidos, aceites e publicados, o que não só demonstra os contributos de investigação deste trabalho, como o compromisso com o rigor académico e a participação ativa na divulgação científica.

Keywords: API REST · Georreferenciação · Sistemas de Informação · Engenharia Organizacional · DEMO · *Low-code*

Abstract

In enterprise data management, the development of REST APIs for integration with diverse information systems is crucial. This need often leads to performing repetitive, laborious and time-consuming tasks. The low-code/no-code platforms allow a significant transformation to this landscape by enabling the rapid/automatic generation of APIs for data input and output and execution of operational actions.

The work developed in this thesis begins with the execution of Project *GesFoGO*, in partnership with the *Universidade de Las Palmas de Gran Canaria* (ULPGC) and the collaboration of several other official entities such as *Serviço Nacional de Proteção Civil e Bombeiros de Cabo Verde*, *Instituto das Florestas e da Conservação da Natureza da Madeira* (IFCN) and Governments. The *GesFoGO* project promotes the cooperation between organizations with operational capacity for the creation of a comprehensive real-time fire prevention and management network, through fast deployment mobile units with self-georeferenced system, to achieve a sustainable management in mountainous forest environments, characteristic of the cooperation territories. The development of this project involved creating a web application that includes the modeling of the database, the implementation of a server component (REST API) for data access and addition of security mechanisms and also, the graphical user interface (GUI) with integration of georeferencing systems. After integration with the hardware, developed by the ULPGC, field tests and demonstrations were performed to the official entities that confirmed the correct operation of the entire system and did not identify any missing information.

Then a REST API was implemented for another research project, *MiColEC*, which presents a new logistical paradigm that promotes collaboration between last-mile delivery companies, through the sharing of media and resources.

The collaboration on these projects, especially with regard to REST APIs, contributed to the acquisition of knowledge and abilities in that topic. Based on these acquired skills, the addition of REST API creation and management mechanisms was proposed to the *low-code* platform DISME (*Dynamic Information System Modeler and Executer*), which is being developed in ARDITI by the EELab (*Enterprise Engineering Laboratory*) team. This approach aims to make the development of REST APIs faster and easier, eventually without the need for programming skills. More specifically, allowing the generation of endpoints for simple read access, complex query results and resource creation, taking advantage of the DEMO (*Design and Engineering Methodology for Organizations*) methodology and DISME, through *drag-and-drop* operations in an intuitive graphical user interface.

Some of the results gathered during the development of these projects are in scientific papers that were submitted, accepted and published, which not only demonstrates the investigation contributions of this work, but also the commitment to academic rigor and active participation in scientific promotion.

Keywords: REST API · Georeferencing · Information Systems · Enterprise Engineering · DEMO · Low-code

Agradecimentos

A realização desta tese apenas se tornou possível devido ao trabalho, dedicação, apoio e motivação de várias pessoas, a quem estou extremamente grato.

Em primeiro lugar, agradeço ao meu orientador da tese, Professor David Sardinha Andrade de Aveiro, pelo apoio ao longo de todo o processo de elaboração desta tese. O seu conhecimento, auxílio e disponibilidade foram cruciais para me direcionar e ajudar a superar os desafios que foram surgindo e levar esta tese a bom porto. Além disso, quero agradecer as oportunidades que me proporcionou e a confiança que depositou nas minhas capacidades.

Agradeço também o esforço de toda a equipa do EELab, *Enterprise Engineering Laboratory*, pelo seu trabalho e ajuda inestimáveis.

Reconheço também que este trabalho não teria sido possível sem o suporte da ARDITI, Agência Regional para o Desenvolvimento da Investigação, Tecnologia e Inovação, pelas Bolsas de Investigação que me concedeu durante os últimos 4 anos.

Por fim, estou muito grato à minha família pelo seu apoio incondicional, compreensão e motivação durante todo o meu percurso académico.

A todos, o meu sincero obrigado.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

API Application Programming Interface

BD Base de Dados

CSS Cascading Style Sheets

DEMO Design and Engineering Methodology for Organizations

DISME Dynamic Information System Modeller and Executer

EBNF Extended Backus-Naur Form

FTP File Transfer Protocol

GUI Graphical User Interface

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

JSON JavaScript Object Notation

LCDPs Low-code Development Platforms

LCPs Low-code Platforms

MDE Model-driven Engineering

MVC Model-View-Controller

NPM Node Package Manager

ORM Object-Relational Mapper

PHP Hypertext Preprocessor

RAAG REST API Automatic Generation

REST REpresentational State Transfer

RRAM Rapid REST API Management

SGBD Sistema de Gestão de Bases de Dados

SQL Structured Query Language

SSH Secure Shell Protocol

ULPGC Universidade de Las Palmas de Gran Canaria

URI Uniform Resource Identifier

URL Uniform Resource Locator

VCS Version Control System

1 Introdução

Este capítulo apresenta o enquadramento geral e objetivos das várias tarefas realizadas no âmbito desta tese, bem como a estrutura do documento para uma melhor compreensão geral dos conteúdos relatados.

1.1 Enquadramento

Esta tese começa com o desenvolvimento do projeto *Red GesFoGO*[1] (MAC2/3.5b/227) - Rede abrangente de prevenção e gestão de incêndios florestais por meio de georreferenciamento em observadores móveis, que foi desenvolvido em parceria com a *Universidade de Las Palmas de Gran Canaria* (ULPGC) e foi aprovado pelo Programa de Cooperação Territorial INTERREG V A Espanha-Portugal, MAC 2014-2020. Além disso, envolve várias outras entidades[2], como Serviço Nacional de Proteção Civil e Bombeiros de Cabo Verde, Instituto das Florestas e da Conservação da Natureza da Madeira (IFCN) e Governos.

A partir deste ponto, o projeto '*Red GesFoGO*' será referido pela sua forma abreviada, '*GesFoGO*', por questões de brevidade e facilidade de referência ao longo desta tese. Em média, ocorrem anualmente cerca de 65 000 incêndios na Europa, queimando aproximadamente meio milhão de hectares de terras selvagens e áreas florestais [1]. Esta tragédia reduz drasticamente a biomassa florestal e a biodiversidade, causando danos graves aos ecossistemas [2]. A capacidade de detetar a ocorrência de um incêndio florestal e ter a capacidade de realizar um acompanhamento preciso e em tempo real da sua evolução é vital para organizar, de forma rápida e eficiente, os recursos disponíveis para o controlar e extinguir [2, 3]. O projeto *GesFoGO*, propõe a cooperação entre organizações, instituições e empresas com capacidade operacional e científico-tecnológica para o desenvolvimento de uma rede abrangente de prevenção e gestão de incêndios florestais em tempo real, através de unidades móveis de rápida implantação com sistema auto-georreferenciado, para alcançar uma gestão sustentável em ambientes florestais montanhosos, característicos dos territórios de cooperação.

Inicialmente, a ideia era utilizar a plataforma *low-code Dynamic Information System Modeller and Executer* (DISME) para implementar as funcionalidades do projeto *GesFoGO*, visto que esta permite representar e executar sistemas de informação automaticamente baseado em modelos de processos das organizações. No entanto, após alguma ponderação chegamos à conclusão de que o *GesFoGO* apresentava uma complexidade acrescida devido à utilização de sistemas de georreferenciamento e também à necessidade de ter uma API REST, que ainda teriam de ser acrescentadas ao DISME, o que seria difícil de desenvolver em tempo útil. Por essa razão, foi decidido avançar no projeto *GesFoGO* com uma abordagem *standalone* e, posteriormente, usar a experiência obtida para enriquecer o DISME com os mecanismos de criação e gestão de APIs REST.

A colaboração nestes projetos teve lugar na ARDITI (Agência Regional para o Desenvolvimento da Investigação, Tecnologia e Inovação), mais concretamente na equipa do EELab (*Enterprise Engineering Laboratory*). A colaboração no projeto *GesFoGO* teve início em setembro de 2020 e decorreu até à data do seu término em setembro de 2022. No entanto, como estava integrado na equipa do EELab, por vezes houve a necessidade de colaborar noutros projetos, nomeadamente o projeto *BIO BEX-A* e o projeto *MiCoLEC*.

[1] Red GesFoGO - `https://www.gesfogo.ulpgc.es/index.php/pt/`
[2] Parceiros - `https://www.gesfogo.ulpgc.es/index.php/pt/socios-pt`

O projeto *BIO BEX-A*[3] pretendia fornecer uma estação totalmente digital e controlada centralmente para o tratamento de águas residuais, com base na análise de dados, com o principal objetivo de reduzir a contaminação causada pelas águas residuais resultantes das estações de tratamento de água. Colaborei neste projeto entre novembro de 2021 e abril de 2022, onde fui responsável por desenvolver a aplicação web, mais especificamente a aplicação da metodologia *Design and Engineering Methodology for Organizations* (DEMO) [4] para a modelação de processos organizacionais e identificação de requisitos, a implementação do sistema de gestão de dados operacionais e implementação da sua interface gráfica. A implementação deste projeto resultou num total de 2943 linhas de código, divididas em 89 ficheiros PHP, 13 ficheiros *javascript*, 1 ficheiro CSS, 4 ficheiros JSON e 1 ficheiro HTML.

O projeto *MiCoLEC*[4] (M1420-01-0247-FEDER-000072) - Micro-hubs Colaborativos para a Economia Circular, apresenta um novo paradigma logístico em que um grupo de empresas de entregas *last-mile* colaboram entre si, partilhando meios e recursos de entrega de encomendas, com vista à redução de custos de operação, melhoria da qualidade de serviço e cobertura de mais zonas geográficas sem investimentos iniciais avultados. Além disso, estende o conceito de micro-hub logístico colaborativo para cobrir a promoção e gestão de processos de economia circular, aumentando simultaneamente a produtividade e a sustentabilidade das operações logísticas associadas. Trabalhei neste projeto entre outubro de 2022 e junho de 2023, onde fui responsável por desenvolver a API REST e a respetiva base de dados aplicando a metodologia DEMO. A implementação deste projeto resultou num total de 7957 linhas de código, divididas em 122 ficheiros PHP, 3 ficheiros *javascript* e 2 ficheiros JSON.

Por motivos de redundância, os projetos *BIO BEX-A* e *MiCoLEC* não são aqui reportados. Embora tenham contribuído bastante para continuar a aplicar e adquirir conhecimentos e valências no desenvolvimento de APIs REST, não apresentam muita coisa nova relativamente ao que é explicado a respeito da implementação do projeto *GesFoGO*.

Com base em todo o conhecimento e competências adquiridas com o envolvimento nos projetos mencionados, sobretudo no que diz respeito a APIs REST, a partir de julho de 2023 passei a estar dedicado ao projeto DISME, mais concretamente na integração de mecanismos de criação e gestão de APIs REST.

1.2 Resultados

Ao longo deste documento, várias secções apresentam os resultados e conclusões da investigação realizada sob a forma de quatro artigos científicos submetidos, aceites e publicados. Especificamente, o artigo "*Ontology for a Georeferencing Mobile System for Real Time Detection and Monitoring of Wildfires*" [5], que engloba algum do conteúdo dos Capítulos 2 e 3. A Subsecção 5.1.3 resulta de uma forma resumida do conteúdo presente no artigo "*The DISME low-code platform - from simple diagram creation to system execution*" [6]. Enquanto que, o Capítulo 5 incorpora a informação presente nos artigos "*DEMO Model based Rapid REST API Management in a low code platform*" [7] e "*Rapid REST API Management in a DEMO Based Low Code Platform*" [8], sendo que este último paper é uma versão mais refinada e mais desenvolvida do anterior. A informação mais detalhada acerca de cada um destes artigos científicos encontra-se na Secção 6.2.

[3]BIO BEX-A - `https://www.arditi.pt/pt/projetos-finalizados/projeto-bio-bex-a`
[4]MiCoLEC - `https://www.arditi.pt/pt/projetos-finalizados/projeto-micolec`

A inclusão de capítulos dedicados aos artigos submetidos, aceites e publicados nesta tese não só mostra os contributos de investigação deste trabalho, como demonstra o compromisso com o rigor académico e a participação ativa na divulgação científica.

1.3 Estrutura do Documento

Este documento está organizado em 6 capítulos. No presente Capítulo 1, é fornecida a introdução e contextualização dos projetos/atividades que são abordadas ao longo desta tese. Além disso, são apresentados alguns dos resultados e conclusões da investigação realizada.

No Capítulo 2, são apresentados os requisitos funcionais estabelecidos para o projeto *GesFoGO*, é mostrada e explicada a arquitetura geral do sistema, são explicadas as razões de termos optado por ter uma aplicação web em vez de uma aplicação *Electron*, bem como alguns dos desafios que surgiram durante o processo. Além disso, são apresentadas as aplicações utilizadas durante o processo de desenvolvimento, e também as tecnologias usadas em cada uma das componentes principais: a Base de Dados, o *backend* e o *frontend*.

O Capítulo 3 mostra toda a implementação, de forma detalhada, das três componentes principais da aplicação do *GesFoGO*, nomeadamente a Base de Dados, a API REST (*backend*) e a GUI (*frontend*). Começa com a exposição do modelo de dados, seguida de todo o desenvolvimento relativo à API REST, o acesso aos dados, a segurança e autenticação, a validação/testes ao funcionamento e a documentação. Depois, é explicada a parte do *frontend*, ou seja, a implementação e explicação dos componentes/funcionalidades que compõem a interface gráfica da aplicação.

No Capítulo 4, é esclarecido como foi feita a colocação em produção da aplicação do *GesFoGO* no servidor do EELab, bem como a integração das partes desenvolvidas pela ARDITI e pela ULPGC, incluindo os desafios que surgiram durante esse processo. No final, são ainda expostos os resultados dos testes no terreno e das demonstrações oficiais que se realizaram.

O Capítulo 5 muda o foco para o enriquecimento da plataforma *low-code* DISME com uma nova funcionalidade para a integração de mecanismos de API REST para criação e gestão de *endpoints*. Inicialmente é feita uma contextualização onde é apresentada a motivação para esta nova funcionalidade/componente e é feita uma breve introdução do DISME. Depois, são apresentados os modelos DEMO mais relevantes para este caso específico. Em seguida, é feita a revisão de literatura para analisar a investigação já existente na temática de criação e gestão automática/rápida de APIs REST. Feito isto, é então apresentada a nova abordagem proposta para integração desta funcionalidade no DISME.

Por fim, no Capítulo 6 é feito um balanço global dos projetos e atividades abordadas nesta tese. São enumeradas as publicações de artigos científicos relativos a estes trabalhos de investigação, depois são discutidos alguns trabalhos/tarefas futuras para os projetos relatados e é feito um balanço final com algumas considerações e conclusões obtidas com a realização desta tese.

2 Requisitos, Arquitetura e Tecnologias

Neste capítulo é feita a apresentação dos requisitos funcionais estabelecidos para o projeto *GesFoGO*, a arquitetura que foi projetada e as tecnologias usadas para garantir o cumprimento desses requisitos. Importa referir que quando iniciei a colaboração no *GesFoGO*, uma parte da sua interface gráfica já tinha sido desenvolvida utilizando a *framework Electron*[5] que serve para desenvolver aplicações multi-plataforma usando *Javascript*, HTML e CSS. A ideia inicial foi fazer uma implementação rápida seguindo o estilo RAD - *Rapid Application Development* [9] para rapidamente se testar e mostrar as funcionalidades básicas da aplicação, como uma espécie de protótipo. No entanto, faltava ainda implementar algumas das funcionalidades necessárias, melhorar algumas já existentes e adaptar esta implementação para uma arquitetura baseada na Web, de forma a permitir usar este sistema em diferentes dispositivos (PC, Tablet, Smartphone,...).

Nas secções que se seguem são enumerados os requisitos funcionais da aplicação, é mostrada e explicada a arquitetura geral do sistema, são explicadas as razões de termos optado por uma aplicação Web em vez de uma aplicação *Electron*, bem como alguns dos desafios que surgiram durante o processo. Além disso, são apresentadas as aplicações utilizadas durante o processo de desenvolvimento, bem como as tecnologias usadas em cada uma das componentes principais: a Base de Dados, o *backend* e o *frontend*.

2.1 Requisitos Funcionais

Nesta secção são descritos os requisitos funcionais para o *GesFoGO*, nomeadamente os recursos e capacidades essenciais para o bom funcionamento e eficácia da solução proposta.

A aplicação deve ter uma Interface Gráfica do Utilizador (GUI) composta por 3 áreas de trabalho principais:

- ***MainVisor*** - Plano 2D com aspeto do terreno (ortofoto) e diversas camadas de trabalho.
- ***NodeVisor*** - Dados e ações relativas ao sensor selecionado.
- ***ImageVisor*** - Imagem térmica e/ou visível captada pelo sensor selecionado.

Cada área de trabalho deve ter a capacidade de realizar as diversas operações e mostrar várias informações, conforme referido em seguida.

MainVisor:

- Possibilidade de mostrar/ocultar camadas (visibilidade, avanço do fogo, podendo desenhar possíveis evoluções e contornos do fogo).
- Colocar marcas provisórias com opção de colocar tudo novo ou coletar parte da marca anterior.
- Possibilidade de fazer zoom.
- Os pontos pré-definidos deverão estar sujeitos a rotas de fuga operacionais.
- Deve ser indicado para onde o sensor está a apontar.
- Definir o intervalo de tempo de coleta dos dados mostrados.
- Os dados devem ter carimbo de data/hora para ser sincronizados.

[5] Electron - `https://www.electronjs.org/`

NodeVisor:

- Localização do sensor em coordenadas UTM ou Longitude/Latitude.
- Referência ou indicação gráfica de onde o sensor está a apontar.
- A pessoa/equipa/organização responsável pelo referido sensor.
- O estado da bateria (percentagem).
- O estado das comunicações que estão disponíveis para esse sensor (VHF, 3G, etc).

ImageVisor:

- Escolha entre imagem visível e térmica.

Com base nestas especificações gerais, são desenvolvidos mais detalhadamente os formatos para cada uma das opções que farão parte da GUI, em seguida.

Requisitos Gerais:

- Cada área de trabalho deve possuir a opção de ocupar a página/ecrã inteiro (*fullscreen*).
- Cada área de trabalho deve ter pelo menos um menu ou botão(ões) para opções.

Funcionalidade de Linha do Tempo (*Timeline*):

- Permite estabelecer o período temporal para as amostras a apresentar.
- Permite escolher o intervalo de tempo entre cada amostra que vai ser apresentada.
- Os dados de cada sensor devem ter carimbo de data e hora para serem sincronizados.

Camadas de informação e opções do *MainVisor*:

- Cada família de informações deve estar numa camada independente que pode ser ativada/desativada, o que facilita e melhora a organização e gestão da informação. Estas camadas podem ser, por exemplo, zonas de fogo detetadas pelos sensores, zonas de fogo onde os sensores não conseguem captar informação mas que podem ser obtidas por outros meios (aéreos, operacionais, etc.) e desenhadas manualmente pelos técnicos, entre outras.
- Podem existir camadas resultantes de operações (adição, subtração, gradiente, etc.) das camadas anteriores.
- De acordo com os pontos anteriores as camadas sugeridas são:
  - Linhas de fogo fornecidas pelos sensores.
  - Linhas de fogo introduzidas manualmente pelo operador. (*)
  - Avanço das linhas de fogo inseridas manualmente pelo operador. (*)
  - Linhas de ação em caso de incêndio inseridas manualmente pelo operador. (*)
  - Linhas de visibilidade de cada sensor.
  - Propostas do gestor de localização para cada sensor.
  - Outras linhas inseridas manualmente pelo operador. (*)
  - Gradientes de evolução do fogo. (**)

- Gradientes de temperatura (temperaturas-> comprimento da chama). (***)

– Conforme o instante de tempo que está a ser mostrado, deve ser possível a alteração/adição de cada camada manual.

– O mapa deve ter a possibilidade de fazer zoom de forma cómoda e simples.

– A representação manual poderá ter várias opções: desenho de mão livre, junção de pontos, etc.

(*) A versão final e as opções ficam a critério da ARDITI-ULPGC, levando em consideração os recursos e plano de trabalho do projeto. Por outro lado, sugere-se diferenciar claramente as camadas que estão associadas à informação fornecida pelo sistema (não editáveis) e aquelas que podem ser modificadas manualmente. Propõe-se chamar esta última "camada manual".

(**) O sentido de avanço pode ser sempre introduzido manualmente e tendo em conta a evolução do incêndio. Contudo, propõe-se a possibilidade de incluir informações de velocidade de avanço (distância/tempo) obtidas pontualmente (dois pontos) ou a média. No entanto, a sua implementação não é garantida porque está sujeita à capacidade de obter os valores necessários a partir da informação disponível.

(***) Os gradientes de temperatura requerem informação que não é garantida com canais de informação de banda estreita. No entanto, será analisada a possibilidade de estabelecer níveis intermédios de volume de informação que possam ser enviados através de banda estreita. Posto isto, a sua implementação não é garantida porque está sujeita à capacidade de obter tais valores a partir da informação e dos meios disponíveis.

Informação e opções do *NodeVisor*:

– A localização do sensor deve aparecer em coordenadas UTM ou Longitude/Latitude. Esta localização está ligada a uma referência temporal, uma vez que um sensor pode estar localizado em diferentes pontos ao longo do tempo.

– Conforme o ponto anterior, deve aparecer uma referência temporal ou utilizar a Linha do Tempo (*Timeline*).

– Deve ser mostrada a pessoa/equipa/organização responsável pelo sensor que está a ser mostrado. Esta informação deve ser inserida manualmente.

– Estado do sensor: estado da bateria, comunicações disponíveis (VHF, Banda Larga, ...), alarmes, etc.

– Configuração: versões de hardware e software.

– Deve ser possível configurar o intervalo de tempo de captura de dados. Esta medida resulta em poupança de energia: o operador pode considerar que o progresso numa determinada área é muito lento e decidir aumentar o intervalo de tempo entre as amostras.

– Deve ser possível forçar os ângulos para onde o sensor está a apontar, para delimitar a área de visualização.

– Pode ser configurada uma posição fixa de visualização ou modo varrimento, em que o sensor faz a rotação delimitada por um ângulo mínimo e máximo.

– Coeficientes associados à localização do sensor, caso estejam previamente definidos no "Assistente de Localização". Esses coeficientes indicam a qualidade do ponto onde o sensor está localizado com base em diferentes critérios (coeficiente de adequação).

Informação e opções do *ImageVisor*:

– As imagens mostradas correspondem às capturas do sensor que está selecionado no *NodeVisor*. Esta escolha deve ser feita com um simples clique no sensor presente no *MainVisor*.

– Deve permitir escolher entre imagem visível e térmica (desde que estejam disponíveis).

– Deve haver informações que indiquem para onde o sensor está a apontar, considerando que podem ser necessárias várias imagens para cobrir a área de observação definida.

Funcionalidade do *Assistente de localização*:

– Os locais propostos para instalar o sensor podem estar sujeitos a rotas de fuga operacionais. (Este ponto deverá ficar para uma eventual versão futura, não será abordado neste projeto).

– Mostra uma representação visual da cobertura do sensor na localização escolhida.

– Apresenta propostas de mudança de local do sensor em que é abrangida uma área de captura maior.

Nota: Importa referir que, ao longo do desenvolvimento do projeto foram realizadas várias reuniões técnicas onde foram tomadas decisões de forma colaborativa, que resultaram na adaptação ou não realização de alguns dos requisitos funcionais, seja por motivos operacionais, por falta de tempo ou mesmo por termos chegado à conclusão de que já não faziam sentido.

2.2 Arquitetura do Sistema

Antes de descrever o que foi implementado, interessa mostrar e explicar a arquitetura do sistema como um todo e também da parte que está a ser desenvolvida no âmbito desta tese. Na Figura 1 é apresentada a arquitetura geral do sistema.

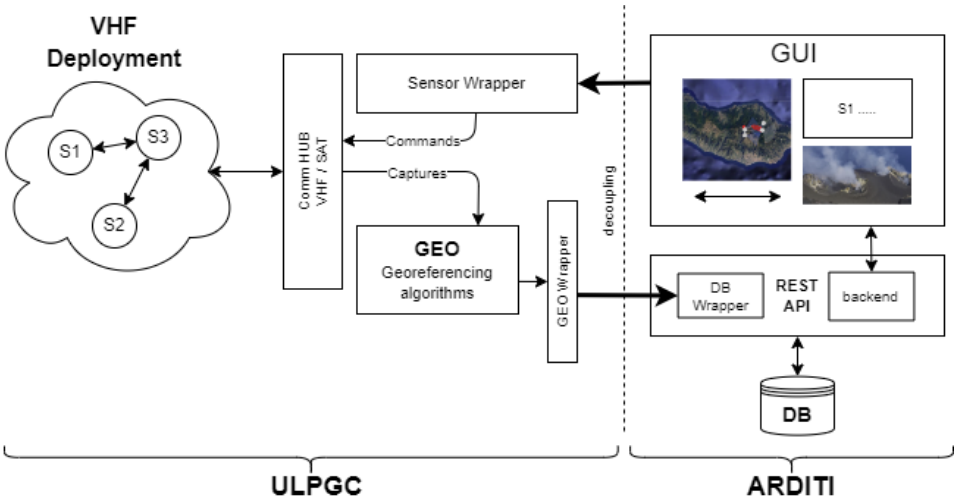


Figura 1. Arquitetura geral do sistema

Como é possível observar na Figura 1, a arquitetura está dividida em duas partes principais. No lado esquerdo da Figura, está a parte desenvolvida pela ULPGC e no lado direito está a parte da ARDITI, que é a parte desenvolvida no âmbito desta tese. Portanto, a parte que é responsabilidade da ARDITI corresponde à base de dados, API REST e interface gráfica da aplicação, onde o utilizador pode visualizar o mapa com os sensores, linhas de fogo e fotografias e também, enviar comandos para alterar algumas configurações dos sensores, como o campo de visão, o ângulo para onde apontam, entre outras. A explicação detalhada da implementação de cada uma destas componentes é feita no Capítulo 3. Quanto à integração das duas partes, da ARDITI e da ULPGC, a explicação mais detalhada é feita na Secção 4.3.

A ULPGC ficou encarregue de desenvolver os sensores propriamente ditos (*hardware*), identificados como S1, S2 e S3 na Figura 1, o *software* de comunicação e controlo dos sensores (bloco *"Comm HUB VHF / SAT"*), o bloco *"Sensor Wrapper"* que é uma interface REST que permite receber e mandar executar comandos nos sensores, o bloco *"GEO Wrapper"* que realiza os pedidos à interface REST *"DB Wrapper"* para envio de capturas para a base de dados e também os algoritmos do bloco *"GEO"* que, através da posição onde os sensores estão instalados e das imagens que captura, consegue determinar com elevada precisão a localização (latitude e longitude) do(s) foco(s) de incêndio. Na Figura 2, é ilustrado um pequeno exemplo, através da imagem da esquerda (e da imagem térmica correspondente) capturadas pelo sensor, são obtidas as coordenadas geográficas da imagem satélite da direita. O pontos onde foi detetado o fogo estão destacados com as setas, sendo que o algoritmo determinou o ponto vermelho da imagem da direita.

Figura 2. Exemplo do funcionamento do algoritmo que determina coordenadas de fogo.

Tendo em conta as escolhas e decisões tomadas para desenvolver a aplicação Web, chegou-se a uma arquitetura para esta parte do sistema, que pode ser observada na Figura 3, em seguida.

Como mostra a Figura 3, temos as componentes cliente e servidor separadas. Esta necessidade surgiu por causa da impossibilidade de aceder diretamente à base de dados através do *browser*, como era possível através da aplicação em *Electron* e é explicada melhor nas secções seguintes. A componente cliente consiste num *"Bundle"* com conteúdo estático que é carregado apenas na primeira vez que a aplicação é pedida, composto pelo ficheiro HTML principal e todos os outros recursos necessários, *javascript*, CSS e imagens, que é servido pela componente servidor. A escolha de utilizar um *Bundler* está justificada na Secção 3.8. A componente servidor, além de servir a aplicação cliente, contém a API REST que também está dividida em duas componentes: *'Backend'* que consite nos *endpoints* que são utilizados para dar suporte à GUI e *'DB Wrapper'* que também

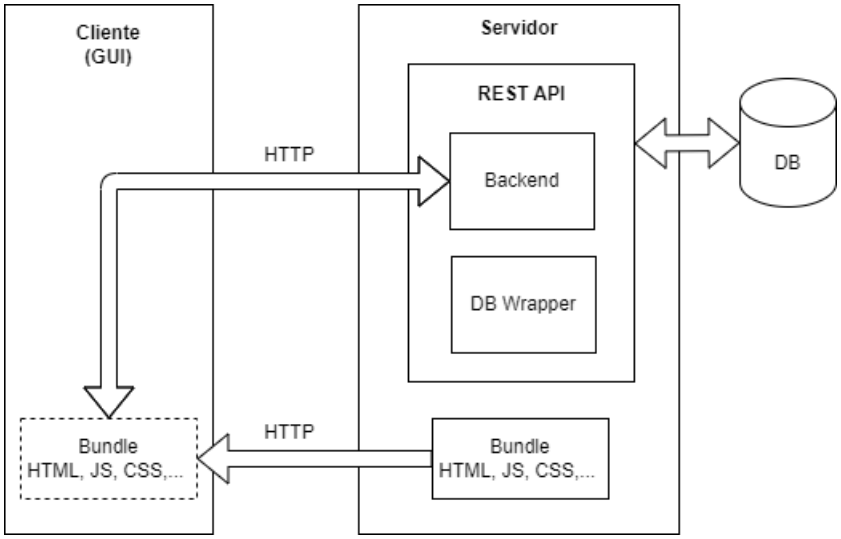


Figura 3. Arquitetura da aplicação web

consiste num conjunto de *endpoints* mas servem para receber os dados provenientes dos sensores (linhas de fogo, imagens e outras informações). Por fim, temos a base de dados *(DB)* onde são armazenadas todas as informações operacionais para o funcionamento da aplicação, bem como as capturas recebidas dos sensores.

2.3 Passagem de Aplicação *Electron* para Aplicação Web

Como já foi abordado, quando iniciei a colaboração no projeto *GesFoGO* a aplicação já estava a ser desenvolvida usando a *framework Electron*, que serve para desenvolver aplicações multi-plataforma com *Javascript*, HTML e CSS. No entanto, nas reuniões técnicas foi decidido fazer a passagem para uma aplicação baseada na web. Como a aplicação já precisava de algumas alterações estruturais, decidimos que seria melhor fazer esta passagem já numa fase inicial.

A opção de ter uma aplicação web em vez de uma aplicação *Electron* deveu-se sobretudo à maior compatibilidade em diferentes plataformas através da utilização de um navegador web, proporcionando uma compatibilidade mais alargada, e também à facilidade de distribuição/disponibilização e atualização, isto é, as aplicações não requerem uma instalação local porque são disponibilizadas de modo centralizado e acedidas através de um navegador web. Em relação às atualizações nas aplicações web, não é necessária uma atualização manual por parte dos utilizadores, basta atualizar a aplicação que está no servidor e no próximo pedido feito o utilizador já tem acesso à nova versão. Além disso, as aplicações web aderem ao modelo de segurança dos navegadores, que têm algumas camadas de segurança adicionais contra certas vulnerabilidades. Apesar das vantagens enumeradas também há algumas desvantagens como, a impossibilidade de utilizar a aplicação em modo *offline*, mesmo que as funcionalidades sejam limitadas, e também podem ter alguma limitação em termos de recursos e desempenho. Tendo em conta as vantagens, desvantagens, os requisitos e prioridades deste projeto, consideramos que a escolha mais adequada é a aplicação web.

O processo de passagem da aplicação em *Electron* para uma aplicação web com componente servidor e componente cliente não requereu muita pesquisa, mas sim uma análise daquilo que já estava feito e de como estava feito. Uma aplicação *Electron* consiste numa espécie de aplicação

web que é executada como uma aplicação *desktop*, ou seja, uma aplicação *Electron* é implementada usando HTML, CSS e *Javascript* e até utiliza *Node.js*[6] e o gestor de módulos NPM[7].

Posto isto, a decisão mais indicada tendo em conta o tempo disponível e também prioridade de aproveitar ao máximo o código já implementado, evitando implementar novo código, por exemplo, numa outra linguagem de programação, foi decidida a utilização de *Node.js*. Desta forma, apenas algum código estrutural é que teve de ser mudado e puderam ser aproveitadas muitas das funções já implementadas, mantendo as funcionalidades da GUI a funcionar, mesmo que com alguns *bugs* facilmente corrigidos. Foram aproveitadas aproximadamente 380 linhas de código, a maior parte em *javascript*, para funções relacionadas a algumas funcionalidades do *Leaflet* na GUI, e ainda, algum HTML.

Com a tomada desta decisão surgiu um problema relacionado ao acesso à base de dados, que resultou na necessidade de separação das componentes cliente e servidor. Os detalhes deste problema, bem como a forma como foi resolvido encontram-se explicados na Subsecção 2.3.1.

2.3.1 O Porquê da API REST

Como já foi referido, a necessidade de ter uma API REST [10] surgiu por causa da impossibilidade de aceder diretamente à base de dados através do navegador web, como era possível através da aplicação em *Electron*. Portanto, a aplicação do *GesFoGO* passou a ter de estar dividida em duas partes: *backend* e *frontend*, como já é referido na Secção 2.2.

O intuito de implementar uma API REST no contexto deste projeto é que esta seja utilizada como interface REST [11] para a gestão dos dados necessários para o funcionamento dos sensores, bem como dos dados recebidos dos mesmos. Dado esta seguir os princípios da arquitetura REST, isto é, expõe a sua informação através de representações sem estado dos seus recursos, permite a interoperabilidade entre diferentes tipos de clientes. Neste caso apenas temos um cliente *web*, mas na necessidade de haver outros tipos de cliente, por exemplo, para dispositivos móveis seria fácil a sua integração no sistema. O único requisito para estes conseguirem comunicar é haver uma conexão HTTPS, escolhida por motivos de segurança e compatibilidade em certos dispositivos.

Para manter o funcionamento da aplicação, sendo possível o acesso aos dados a partir do navegador, implementei um servidor bastante simples em *Node.js*, como medida temporária. Normalmente uma API REST possui um *endpoint* para cada tipo de recurso a que estamos a aceder e utiliza diferentes métodos HTTP de acordo com a operação pretendida. Mas como o código já implementado já estava todo a utilizar *queries* SQL, este servidor simples que fiz usando *Node.js*, tinha apenas um *endpoint* que recebia a *query* SQL no corpo (*body*) do pedido HTTP, executava essa *query* SQL à base de dados e retornava o resultado da *query* no corpo (*body*) da resposta do pedido HTTP. O objetivo desta medida provisória foi apenas manter o funcionamento de tudo o que já estava a funcionar e fazer a migração para a API REST de forma gradual conforme os *endpoints* da API REST ficavam prontos.

Para a aplicação do *frontend* passar a utilizar esta este servidor temporário foi apenas necessário mudar uma função, *accessDatabase()*. Esta é a função que é chamada por todas as funções que acedem a dados e passou de realizar os acessos à base de dados para preparar e realizar pedidos

[6] Node.js - `https://nodejs.org/en`
[7] NPM - `https://www.npmjs.com/`

HTTP à API. Essas alterações no código da função *accessDatabase()* podem ser observadas na Figura 4, em seguida.

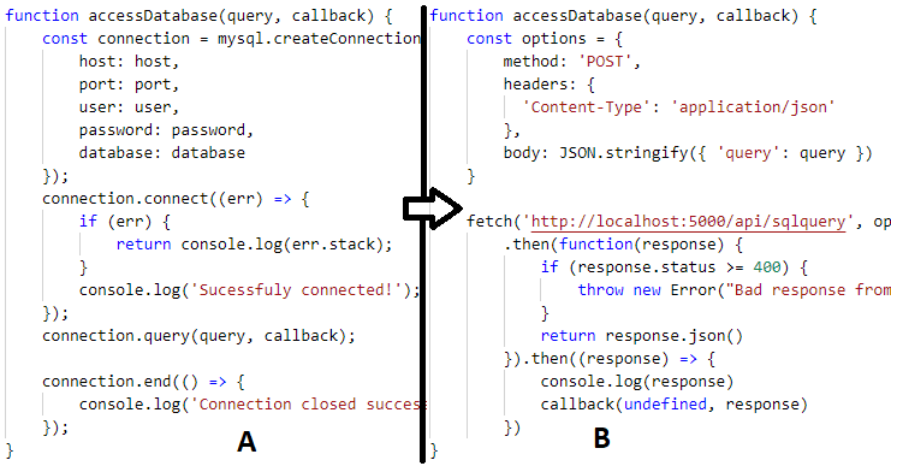


Figura 4. Alteração da função que acede ao dados

A Figura 4 apresenta no lado esquerdo (A) o código para acesso direto à base de dados e no lado direito (B) o código para acesso à API através de HTTP. No entanto, importa salientar que esta medida foi apenas temporária e que já não se encontra no código fonte atual da aplicação do *GesFoGO*.

Posto isto, nas secções que se seguem, vão ser apresentadas todas as ferramentas e tecnologias utilizadas para implementar a Base de Dados, a API REST e a GUI do *GesFoGO*.

2.4 Ambiente de Desenvolvimento

Para todo o desenvolvimento do projeto *GesFoGO* foram utilizadas diversas ferramentas de *software*, que foram essenciais para cumprir os objetivos definidos para este projeto. Desde IDE's *(Integrated Development Environment)*, *software* de controlo de versões, entre outros programas para a criação de documentos auxiliares, diagramas, etc. Em seguida, encontra-se a lista com os programas utilizados, uma breve descrição de cada um e também para que foram usados:

- ***XAMPP***[8] - é um *software open-source*, de fácil instalação e utilização, amplamente utilizado que permite a configuração de um ambiente de desenvolvimento web local, combinando **Apache**[9] como servidor web, **MySQL** como base de dados e **PHP** como linguagem de programação. Utilizado para a base de dados e para executar o servidor da API REST.
- ***PhpStorm***[10] - é um **IDE** específico para desenvolvimento de **PHP**, desenvolvido pela *JetBrains*[11]. Fornece um conjunto de recursos para facilitar o desenvolvimento de **PHP**, como a assistência/sugestão/conclusão de código, correções rápidas, formatação automática, recursos de depuração e suporte para várias *frameworks* **PHP**, nomeadamente ***Laravel***[12], que é usada neste projeto para implementar a API REST.

[8] XAMPP - `https://www.apachefriends.org/`
[9] Apache - `https://httpd.apache.org/`
[10] PhpStorm - `https://www.jetbrains.com/phpstorm/`
[11] JetBrains - `https://www.jetbrains.com/`
[12] Laravel - `https://laravel.com/`

- ***Visual Studio Code***[13] - é um editor de código *open-source* desenvolvido pela *Microsoft*[14]. É uma ferramenta leve, mas poderosa, que suporta várias linguagens de programação e oferece uma vasta gama de recursos para desenvolvimento de software. Fornece recursos como destaque de sintaxe, suporte para depuração, assistência/sugestão/conclusão de código, integração *software* de controlo de versões e permite instalar extensões para funcionalidades adicionais. Utilizado para o desenvolvimento do *frontend* (GUI) em *Node.js* e também para edição dos ficheiros SQL da base de dados.

- ***GitHub***[15] - é uma plataforma baseada na web para controlo de versões e colaboração no desenvolvimento de *software*. Fornece um repositório central onde os programadores podem armazenar os seus projetos e colaborar com outras pessoas. Utiliza o ***Git***[16], um Sistema de Controlo de Versões (VCS) distribuído, para gerir e rastrear alterações no código-fonte durante o processo de desenvolvimento.

- ***GitKraken***[17] - é um cliente ***Git*** com uma interface gráfica que simplifica e melhora a experiência de trabalhar com controlo de versões ***Git***. Fornece uma interface intuitiva e visualmente apelativa para a gestão de repositórios, controlo de alterações no código-fonte, gestão de conflitos, entre outras operações. Além disso, integra-se facilmente com os serviços mais populares de hospedagem de repositórios, como ***GitHub*** (usado neste caso), *GitLab* ou *Bitbucket*.

- ***Postman***[18] - é uma ferramenta de desenvolvimento e teste de API's que simplifica o processo de estruturar, testar e documentar API's. Fornece uma interface intuitiva para enviar pedidos HTTP a um servidor, receber as respostas e inspecionar os resultados. Utilizado para testar cada um dos *endpoints* da API REST.

- ***Diagrams.net***[19] - é uma ferramenta gratuita que permite a elaboração de diagramas. Fornece uma plataforma baseada na web para a criação de uma variedade de diagramas, fluxogramas, diagramas de rede, diagramas de fluxo e muito mais. Neste projeto, foi utilizado para a criação dos diagramas da base de dados e também para os diagramas de arquitetura de todo o sistema.

- ***Google Docs***[20] - é uma aplicação baseada na web desenvolvida pela *Google* que permite criar, editar e armazenar documentos online. Além disso, permite partilhar os documentos com segurança, trabalhar de forma colaborativa em simultâneo e possui o modo de "sugestão"que permite deixar ou obter *feedback* sem alterar o conteúdo original dos documentos. Foi utilizado neste projeto para escrita de relatórios específicos do projeto, para documentos de dúvidas para esclarecer nas reuniões, para apontamentos, para artigos científicos, entre outros.

- ***Google Slides***[21] - é uma aplicação baseada na web desenvolvida pela *Google* que permite criar, editar e armazenar apresentações online. Possui as mesmas características gerais que o *Google Docs*. Foi utilizado neste projeto para criar algumas apresentações necessárias para demonstrações que foram solicitadas durante o desenvolvimento do projeto.

---

[13] Visual Studio Code - `https://code.visualstudio.com/`
[14] Microsoft - `https://www.microsoft.com/pt-pt`
[15] GitHub - `https://github.com/`
[16] Git - `https://git-scm.com/`
[17] GitKraken - `https://www.gitkraken.com/`
[18] Postman - `https://www.postman.com/`
[19] Diagrams.net - `https://app.diagrams.net/`
[20] Google Docs - `https://docs.google.com/document/u/0/`
[21] Google Slides - `https://docs.google.com/presentation/u/0/`

2.5 Tecnologias Utilizadas

Para a implementação do projeto *GesFoGO* foi necessária a utilização de diversas tecnologias, nomeadamente para a base de dados, API REST e GUI. A escolha das tecnologias mais adequadas para cada contexto é crucial para o bom funcionamento das soluções desenvolvidas. Nas 3 subsecções que se seguem são apresentadas as tecnologias selecionadas para cada um dos componentes do *GesFoGO*: Base de Dados, Servidor - API REST e Cliente - GUI.

2.5.1 Base de Dados

No caso do armazenamento persistente do *GesFoGO*, foi utilizado o ***MySQL***, um Sistema de Gestão de Bases de Dados (SGBD), amplamente usado para a gestão e organização de dados estruturados. É utilizado em diversos tipos de aplicações devido à sua fiabilidade, desempenho e facilidade de uso. Neste projeto, a escolha do MySQL como SGBD, já estava feita quando iniciei a minha colaboração. O *XAMPP*, mencionado na Secção 2.4, já inclui o ***phpMyAdmin***, uma interface gráfica para a gestão de bases de dados, que evita a necessidade de utilizar a linha de comandos para executar a maioria das operações mais comuns.

2.5.2 Servidor - API REST

Relativamente à componente servidor, a API REST, foi escolhida a *framework* ***Laravel***, utilizada para desenvolvimento de aplicações web em ***PHP***. Esta escolha resulta de inicialmente estar previsto utilizar o DISME [6] como base do projeto, uma plataforma *low-code* que está a ser desenvolvida pelo EELab, cujo *backend* está implementado em *Laravel*, no entanto, chegou-se à conclusão de que seria melhor fazer uma aplicação *standalone*. A *framework* ***Laravel*** segue o padrão de desenho *Model-View-Controller* (MVC) e as suas funcionalidades mais úteis para este caso específico, a API REST do *GesFoGO*, são o *Object-Relational Mapping* (ORM) ***Eloquent*** que facilita a interação com a BD, o ***roteamento*** que permite a definição de *endpoints*, os mecanismos de ***autenticação*** para controlo de acessos e também uma vasta **documentação** técnica.

2.5.3 Cliente - GUI

Por fim, no que diz respeito à implementação do *frontend* do *GesFoGO*, escolhemos utilizar o ***Node.js***. Esta decisão surge devido à possibilidade de reaproveitar grande parte das funções que já estavam implementadas, apenas realizando algumas alterações ao código estrutural, como é explicado na Secção 2.3, e também porque já tinha experiência prévia. Contudo, o *Node.js* é um ambiente de execução de *Javascript* com várias vantagens para este caso, principalmente o *Node Package Manager* (NPM) que permite a descoberta, instalação e gestão de módulos *javascript*. Além disso, foi usada a *framework* ***Bootstrap***[22], que possui um conjunto de componentes e ferramentas *HTML*, *CSS* e *Javascript* pré-concebidos e personalizáveis, como barras de navegação, formulários, botões e muito mais, que potenciam a interface gráfica do utilizador, tornando-a mais consistente e apelativa.

[22]Bootstrap - `https://getbootstrap.com/`

3 Implementação

Neste capítulo é explicada toda a implementação das três componentes principais da aplicação do *GesFoGO*, nomeadamente a Base de Dados, a API REST (*backend*) e a GUI (*frontend*). Começa com a exposição do modelo de dados, seguida de todo o desenvolvimento relativo à API REST, o acesso aos dados, a segurança e autenticação, a validação/testes ao funcionamento e a documentação. Depois é explicada a parte do *frontend*, ou seja, a implementação e explicação dos componentes/funcionalidades que compõem a interface gráfica da aplicação. A implementação do projeto *GesFoGO* resultou num total de 7111 linhas de código, divididas em 107 ficheiros PHP, 10 ficheiros *javascript*, 4 ficheiros JSON, 2 ficheiros *Markdown*, 1 ficheiro HTML e 1 ficheiro CSS.

3.1 Modelo de dados

Nesta secção é mostrada e explicada a estrutura escolhida para a base de dados, bem como o papel de cada elemento para a solução implementada. Esta estrutura foi elaborada com base na análise dos requisitos funcionais e na identificação das necessidades de todo o sistema, refletindo o resultado de várias melhorias contínuas, como a adição de mais informações, a remoção de informação que já não fazia sentido ou a renomeação de alguns conceitos. Estas melhorias, aplicadas durante toda a implementação do projeto *GesFoGO*, tiveram em consideração o *feedback* transmitido pelos vários parceiros do projeto, nomeadamente os especialistas no combate a incêndios florestais.

Tendo em conta toda a informação que identificamos como necessária e todas as iterações do diagrama, foi elaborado o Modelo Entidade-Associação apresentado na Figura 5. Em seguida, são explicados os atributos, as entidades e as associações do modelo criado.

A entidade **Região** (*Region*) refere-se a uma área geográfica específica onde o *GesFoGO* será utilizado, sendo os atributos relevantes o nome da referida região (*name*) e o relevo (*relief*), ou seja, o tipo de terreno nessa região, por exemplo montanha, planície, entre outras. Uma **Região** representa uma zona que não será inferior a uma ilha, ou seja, no caso da Ilha da Madeira esta é considerada uma única região.

Uma **Campanha** (*Campaign*) é uma atividade de prevenção de incêndios organizada de um determinado tipo (*type*) e com um nome (*name*), num período de tempo determinado através de uma data de início (*start_ date*) e uma data de fim (*end_ date*), que se aplica a uma **Região** específica.

O Sistema do *GesFoGO* tem como uma das suas principais ferramentas os **Sensores** (*Sensor*), que possuem a capacidade de obter imagens visíveis e térmicas do terreno. Cada um destes sensores possui um conjunto de propriedades intrínsecas ao equipamento, nomeadamente o número de série (*serial_ number*), a versão de software (*software_ version*) e hardware (*hardware_ version*), a especificação do comprimento da lente da câmara (*camera_ lens_ length*), o campo de visão esperado da lente (*lens_ fov*) e a capacidade da bateria (*battery_ capacity*).

A entidade **Instalação de Sensor** (*Sensor_ Deploy*) diz respeito à utilização dos **Sensores** mencionados em **Campanhas** específicas, instalados numa determinada **Localização** (*Location*) da **Região**. Estes *deploys* ocorrem dentro de um período específico, que pode ser o mesmo da **Campanha** em si, portanto, precisam de registar o instante de tempo de início (*start_ timestamp*) e de fim (*end_ timestamp*). Outros atributos relevantes incluem: as coordenadas geográficas latitude (*latitude*) e longitude (*longitude*) onde o sensor está instalado (bem como, uma aproximação

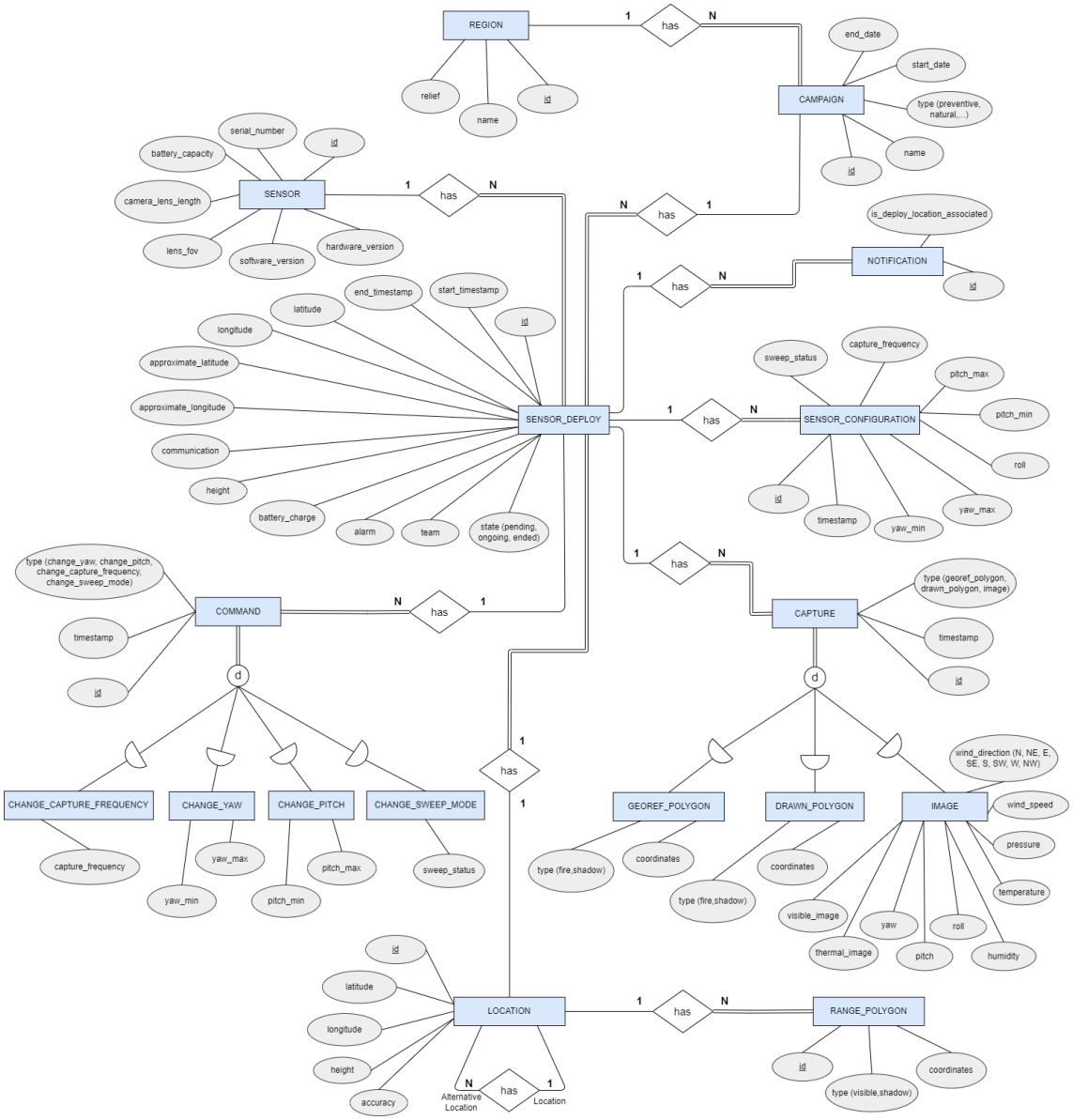


Figura 5. Modelo Entidade-Associação da Base de Dados do *GesFoGO*

menos exata dessa latitude (*approximate_ latitude*) e longitude (*approximate_ longitude*)); a altura (*height*), em metros, a que o sensor está instalado; o tipo de comunicação (*communication*) que pode ser por exemplo, Banda Larga, Banda Estreita, etc; o código de identificação de alarme (*alarm*) ou erro no sensor que possa surgir (não está a ser utilizado); a equipa (*team*) responsável pela instalação e supervisão desse *deploy*; o nível atual da bateria (*battery_ charge*) em percentagem; e o estado (*state*) do *deploy*, que pode ser pendente (*PENDING*) quando foi criado um deploy na plataforma mas o sensor físico ainda não está instalado, em curso (*ONGOING*) quando o sensor está instalado e a funcionar normalmente, ou terminado (*ENDED*) quando o *deploy* acaba. A razão para ter as coordenadas geográficas aproximadas era ficar com o registo das coordenadas que foram sugeridas pela funcionalidade que sugere a melhor localização para a instalação do sensor, e as coordenadas (latitude e longitude) normais para a localização exata do sensor, no entanto, esta funcionalidade acabou por não ser feita.

Cada **Instalação de Sensor** também precisa de ter associada uma **Configuração** (*Sensor_ Configuration*) onde a parametrização para aquele **Sensor** específico é definida, embora, durante o período do *deploy*, esta configuração possa ser alterada várias vezes. Os sensores têm também a possibilidade de estar em modo de varrimento ou modo fixo, determinado pelo atributo estado de varrimento (*sweep_ status*), um booleano que determinada o modo selecionado atualmente, modo fixo (*false*) ou modo varrimento (*true*). No modo varrimento o sensor pode variar entre um limite mínimo (*pitch_ min*) e máximo (*pitch_ max*) de inclinação (eixo lateral/transversal) e um limite mínimo (*yaw_ min*) e máximo (*yaw_ max*) de orientação (eixo vertical), ambos em graus. No caso do sensor estar no modo fixo, os valores dos limites mínimo e máximo são iguais. Os valores de *pitch_ min* e *pitch_ max* só podem variar entre -20 e 20 graus, para evitar que o sensor fique apontado para o chão ou para o céu; já os valores de *yaw_ min* e *yaw_ max* podem variar entre 0 e 360 graus. Além destes, há também a definição do *roll* (eixo longitudinal), que de momento não está a ser usada. Na Figura 6 é apresentada uma imagem que ilustra de forma mais clara o que cada uma destas 3 configurações representa. A **Configuração** do sensor inclui o registo do instante temporal (*timestamp*) em que ocorreu a última alteração e a frequência de captura (*capture_frequency*) de polígonos e imagens.

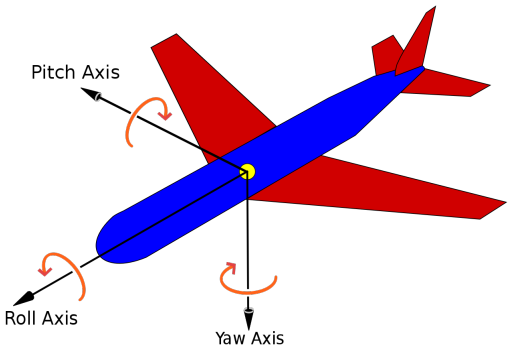


Figura 6. Principais eixos de rotação [12]

Embora o sensor tenha uma **Configuração** inicial, é possível enviar **Comandos** (*Commands*) específicos para serem aplicadas alterações a cada sensor. A entidade *Command* está sempre associada a uma **Instalação de Sensor**, e é uma generalização que agrega os atributos comuns a todos

os comandos, ou seja, o registo do instante temporal (*timestamp*) em que o comando foi solicitado e o tipo (*type*) de comando, que pode ter o valor *'change_ capture_ frequency'*, *'change_ yaw'*, *'change_ pitch'* ou *'change_ sweep_ mode'*, conforme a especialização. As 4 especializações da entidade **Comando** são enumeradas e detalhadas em seguida.

O **Comando** para alterar a **Frequência de Captura** (*Change_ Capture_ Frequency*) do sensor requer apenas o novo valor para frequência de captura (*capture_ frequency*) em minutos. Para atualizar a **Orientação** (*Change_ Yaw*) do sensor são necessários os limites máximo (*yaw_ max*) e mínimo (*yaw_ min*) que podem variar entre 0 e 360 graus. Para atualizar a **Inclinação** (*Change_ Pitch*) do sensor são necessários os limites máximo (*pitch_ max*) e mínimo (*pitch_ min*) que podem variar entre -20 e 20 graus. Por fim, a mudança do **Modo de Varrimento** (*Change_ Sweep_ Mode*) usa um booleano que determina o novo modo como fixo (*false*) ou varrimento (*true*).

Quando está a decorrer uma **Instalação de Sensor** e dependendo da frequência de captura definida, o sensor enviará múltiplas **Capturas** (*Captures*) de imagem e/ou polígonos (também designados por linhas de fogo e linhas de sombra). A entidade *Capture* está sempre associada a uma **Instalação de Sensor**, e é uma generalização que agrega os atributos comuns a todas as capturas, ou seja, o registo do instante temporal (*timestamp*) em que a captura foi feita e o tipo (*type*) de captura, que pode assumir um dos seguintes valores *'georef_ polygon'*, *'drawn_ polygon'* ou *'image'*, conforme a especialização. As 3 especializações da entidade **Captura** são enumeradas e detalhadas em seguida.

As **Capturas** do tipo **Polígono *Georef*** (*'Georef_ Polygon'*) correspondem a polígonos ou linhas de fogo, isto é, polígonos que delimitam a(s) zona(s) onde foi detetado um incêndio, gerados pelos algoritmos de georreferenciação, desenvolvidos pela ULPGC. Já as **Capturas** do tipo **Polígono Desenhado** (*'Drawn_ Polygon'*) correspondem a polígonos ou linhas de fogo, desenhados por um operador/utilizador através da GUI da aplicação. Ambos possuem os mesmos atributos, nomeadamente as coordenadas (*coordinates*) que correspondem a um conjunto de pontos (coordenadas geográficas) que constituem cada polígono (por exemplo, "*POLYGON((-16.918975300 32.742479946,-16.918774 32.741804,...))*"), usando o tipo "*Polygon*"do SQL [13]; e também o atributo tipo (*type*) que pode ser fogo ('*fire*') ou sombra ('*shadow*'), respetivamente, linhas onde foi detetado fogo ou linhas onde os sensores não conseguem ver (por exemplo, devido a uma montanha) mas com base na experiência dos operadores sabem que há fogo nessa zona, considerando a progressão do fogo, as condições meteorológicas locais, o conhecimento do terreno e outras observações diretas (por pilotos de helicóptero, por exemplo). Na Figura 7, é mostrado um exemplo destes polígonos no mapa da GUI, as diferentes cores são usadas para distinguir linhas de fogo, linhas de sombra e linhas desenhadas e vai ser explicado mais à frente, na Secção 3.10.

As **Capturas** do tipo **Imagem** (*Image*) possuem uma imagem visível (visible_image) e uma imagem térmica (thermal_image). 'Imagem visível' é a designação que damos a uma fotografia capturada pelos sensores. Além das imagens, é também registado o valor das configurações do sensor no momento da captura, nomeadamente a orientação (*yaw*), inclinação (*pitch*) e *roll*. São ainda registados alguns dados meteorológicos: a temperatura (*temperature*), a pressão atmosférica (*pressure*), a humidade (*humidity*), a velocidade do vento (*wind_ speed*) e a direção do vento (*wind_ direction*). A direção do vento é representada através dos pontos cardeais e pontos colaterais, ou seja, Norte (N), Sul (S), Este (E), Oeste (O), nordeste (NE), sudeste (SE), noroeste (NO) e sudoeste (SO).

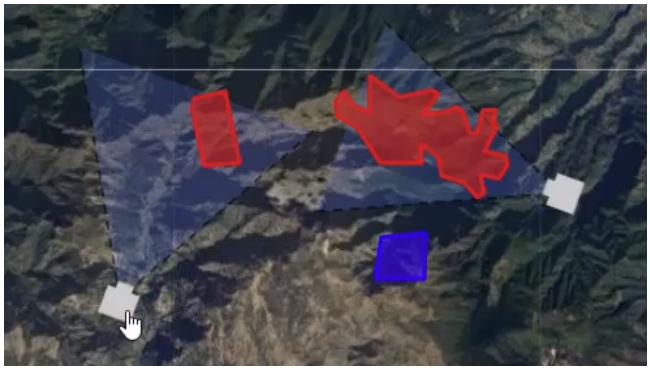


Figura 7. Exemplo de algumas linhas de fogo e de sombra

As entidades que se seguem, apesar de fazerem parte da base de dados e até terem as respetivas operações implementadas como *endpoints* da API REST, acabaram por não ser usados na solução final, por diversas razões que serão explicadas mais à frente.

A entidade **Localização** (*Location*), mencionado anteriormente em **Instalação do Sensor**, serve para dar suporte a uma funcionalidade que, a partir da localização atual da instalação de um sensor, gera/determina algumas localizações alternativas que otimizam a sua área de cobertura. Possui os seguintes atributos: latitude (*latitude*), longitude (*longitude*), altura (*height*) e precisão (accuracy) que é uma classificação de quão boa é essa localização numa escala de 0 a 10.

Cada **Localização** pode ter associados **Polígonos de Cobertura** (*Range_Polygon*), que seguem as mesmas características que os polígonos mencionados na parte das **Capturas**, e delimita a área visível total que o sensor pode capturar, considerando a topografia circundante, como montanhas que podem reduzir a visibilidade. Tem como atributos as coordenadas (*coordinates*) que correspondem a um conjunto de pontos (coordenadas geográficas) que constituem cada polígono e um tipo (*type*) que pode ser fogo ('*fire*') ou sombra ('*shadow*').

Por fim, a entidade **Notificação** (*Notification*) que fica sempre associado a uma **Instalação de Sensor**, serve para reportar que um Sensor foi instalado numa das localizações pré-determinadas (ou não). Tem como atributo um booleano (*is_deploy_location_associated*) que no caso de ter associada uma localização fica a *true*, caso contrário fica a *false*.

3.2 Estrutura do Código da API REST

Num projeto *Laravel*, neste caso específico uma API REST, a estrutura de diretorias e a própria implementação seguem um padrão definido que separa as componentes, facilita a organização e promove a escalabilidade [14]. Em seguida, essa estrutura é descrita explicando o propósito de cada diretoria e/ou ficheiro para o projeto em questão.

- **./app/** - pasta que contém o código principal da aplicação, incluindo modelos, controladores, *middlewares*, entre outras.
  - **./app/Http/Controllers/** - diretoria que contem as classes dos controladores, que possuem as funções que tratam dos pedidos recebidos, processam a informação e retornam respostas. Funcionam como uma ponte entre os *endpoints* definidos e a lógica implementada na aplicação.

    - **./app/Http/Middleware/** - diretoria que contém as classes que definem *middlewares* HTTP, que podem intercetar e processar pedidos HTTP antes destes serem processados pelos controladores. Os *middlewares* podem ser usados para autenticação, autorização, registo, entre outros.

    - **./app/Models/** - pasta que contém os modelos que representam as tabelas da base de dados, responsáveis pela interação com a BD. Cada modelo corresponde a uma tabela e permite definir relações com outros modelos.

- **./config/** - esta diretoria inclui vários ficheiros de configuração para diversos componentes do *Laravel*, da base de dados, da *cache*, da autenticação, etc. No caso do *GesFoGO*, os ficheiros relevantes são: *app.php*, *cors.php*, *sanctum.php* e *scribe.php*.

- **./database/** - esta pasta agrega ficheiros relacionados à base de dados, isto é, Migrações, *Seeders* e *Factories*.

    - **./database/migrations/** - diretoria para as migrações, usadas para definir e modificar o esquema da base de dados e tem a vantagem de permitir o registo das alterações num sistema de controlo de versões, garantindo a sua consistência.

- **./public/** - diretoria onde são armazenados os *assets* públicos da aplicação, como imagens, javascript, CSS, HTML, etc.

    - **./public/docs/** - diretoria onde ficam armazenados todos os ficheiros relativos à documentação da API, obtidos através do gerador da documentção, explicado em mais detalhe na Secção 3.6.

- **./routes/** - esta diretoria possui as definições de *endpoints* que mapeiam os pedidos HTTP recebidos às ações do controlador apropriado.

    - **./routes/api.php** - ficheiro onde são estabelecidas as definições de roteamento para cada *endpoint* da API.

- **./vendor/** - diretoria onde ficam armazenadas as dependências instaladas através do gestor de pacotes do *Laravel*, o *Composer*.

- **./.env** - ficheiro onde são definidas algumas configurações de ambiente específicas para a aplicação. Contém informações acerca da conexão à base de dados, de *API Key's*, entre outras.

Além destas diretorias e ficheiros mencionados, há mais alguns que já fazem parte da aplicação base do *Laravel* e que não foram alterados, portanto não os mencionei.

Antes de passar à implementação propriamente dita dos *Endpoints* da API, importa detalhar alguns procedimentos que foram feitos numa fase inicial para melhorar ou facilitar o processo de implementação da API REST. Em primeiro lugar, nalguns dos modelos (*Models*), foram definidas algumas relações entre tabelas tirando partido do *object-relational mapper* (ORM) *Eloquent* [15], o que facilita a interação com tabelas da base de dados relacionadas entre si. No caso, foram usadas as funções *hasOne()* (Um-Para-Um) e *hasMany()* (Um-Para-Muitos) [15] nos modelos '*Sensor*' e '*SensorDeploy*'. No excerto de código da Figura 8 é mostrado um exemplo, no Modelo '*SensorDeploy*' a função *sensor()* permite obter o objeto *Sensor* que está associado ao '*SensorDeploy*' em questão.

```php
public function sensor()
{
    return $this->hasOne( related: Sensor::class, foreignKey: 'id', localKey: 'sensor_id');
}
```

Figura 8. Excerto de código que exemplifica a definição de relações entre tabelas

Foi também criada uma *Landing-page* para a API, não era algo necessário mas acredito ser uma mais valia sobretudo para quem precisa de utilizar a API. No caso é uma página bastante simples, apenas com o logótipo do projeto, uma frase de boas-vindas, a versão do Laravel (e PHP) e também uma hiperligação para a documentação da API. Na Figura 9 é mostrada uma captura de ecrã da página.

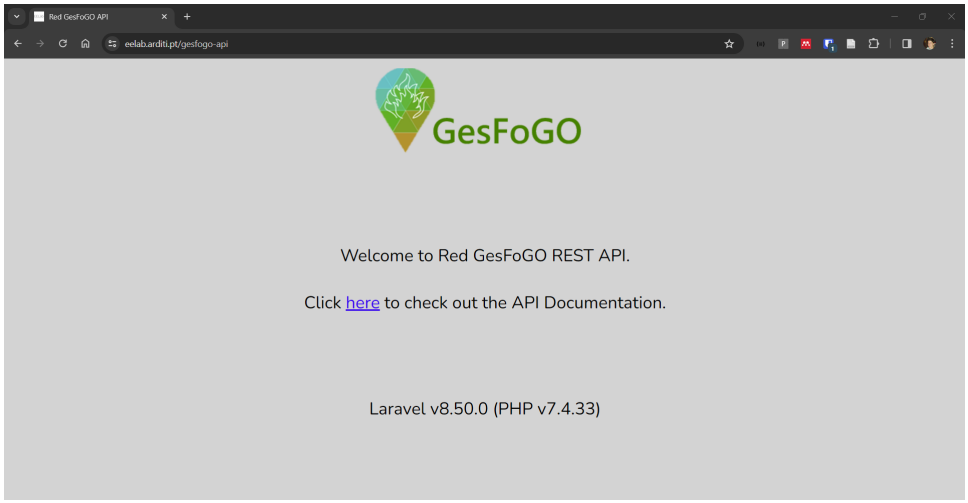


Figura 9. Captura de ecrã da *Landing-page* da API

Para criar esta página adaptei o ficheiro Blade (*welcome.blade.php*) padrão do *Laravel*, localizado na pasta *./resources/views/* e defini que o *endpoint* base da API ('/') retorna essa *view*.

Por fim, foram criadas migrações que adicionam o atributo designado por '*timestamps*' em cada uma das tabelas da base de dados. Este atributo traduz-se na adição dos atributos *created_at* e *updated_at*, que são geridos e atualizados automaticamente pelo *Laravel*, isto é, o *created_at* é definido no momento em que cada tuplo é criado e o *updated_at* é atualizado para o instante de tempo em que é feita a alteração ao tuplo em questão.

3.3 Autenticação e Segurança dos Dados

Para estabelecer a segurança dos dados, foram implementadas restrições de acesso em todos os *endpoints* da API REST, tendo em consideração a forma como as informações sensíveis são armazenadas, como passwords. Como já foi referido na Secção 2.2, os *endpoints* estão divididos em dois grupos principais: '*Backend*' e '*DB Wrapper*', designados como '*GUI Endpoints*' e '*Info Reception Endpoints*' na documentação, respetivamente. Posto isto, foi decidido que seria melhor ter um tipo de autenticação diferente para cada, visto que são utilizados de forma distinta. '*GUI Endpoints*' agrega as rotas da API que são usadas para dar suporte às funcionalidades da GUI, já os '*Info Reception Endpoints*' servem para receber informações vindas dos sensores.

Para os '*GUI Endpoints*' foi utilizada a autenticação através de **Login** do utilizador (*email* + *password*), em que o utilizador se autentica usando as suas credenciais e recebe um *token* que é enviado no cabeçalho dos pedidos subsequentes. Para os '*Info Reception Endpoints*' foi utilizada a autenticação através de **API Key**, na qual é solicitada uma chave ao administrador do sistema e esta chave deve posteriormente ser enviada em cada pedido efetuado.

A escolha desta abordagem híbrida usando os dois tipos de autenticação *login* e *API Key* na API REST do *GesFoGO* deve-se a vários fatores, uma vez que cada método tem o seu próprio conjunto de vantagens e considerações. No caso, a autenticação através de ***Login*** é utilizada para os *endpoints* que dão suporte à GUI, porque esta é centrada no utilizador, isto é, cada utilizador/operador que utiliza a aplicação deve ter a sua conta de utilizador, o que melhora a usabilidade. Nesta autenticação os utilizadores usam um *endpoint* (neste caso, */api/login*) para se autenticar através das suas credenciais (*email* + *password*) e recebem um *token* de sessão que deve ser enviado no cabeçalho dos pedidos seguintes e é válido até ser terminada a sessão (*/api/logout*). A autenticação através de ***API Key*** é utilizada para os *endpoints* para receção de informação resultante dos sensores e dos algoritmos de georreferenciação, porque neste caso o cliente é outro servidor, ou seja, a comunicação é máquina-para-máquina. Desta forma o processo de autenticação é mais simples, removendo a necessidade da aplicação cliente se preocupar com o *login* e *logout*, reduzindo a sua complexidade.

Uma vez justificadas as escolhas, a explicação de como foi realizada a implementação é apresentada nas subsecções que se seguem.

3.3.1 Autenticação com *Login*

Para a autenticação através de *Login* foi escolhido utilizar o módulo *Laravel Sanctum* [16], porque é o módulo oficial do *Laravel* para autenticação, portanto a sua integração é bastante boa e possui uma vasta documentação e suporte.

No entanto, adicionar esta autenticação centrada no utilizador requer algumas tabelas adicionais na base de dados para a gestão de utilizadores ('*users*') e dos respetivos *tokens*. A aplicação base do *Laravel* já vem com 3 migrações para criar essas tabelas: '*users*', '*password_resets*' e '*personal_access_tokens*'. O procedimento para instalar o *Laravel Sanctum* é o seguinte:

1. Instalar o módulo em si, através do comando *'composer require laravel/sanctum'*.
2. Executar '*php artisan vendor:publish –provider="Laravel\Sanctum\SanctumServiceProvider"*', que adiciona o ficheiro de configuração *sanctum.php* na pasta *./config/* e o ficheiro da migração para criar a tabela dos *personal_access_tokens* (caso ainda não existam).
3. Executar o comando *'php artisan migrate'* para criar as tabelas necessárias na base de dados.
4. Adicionar o *sanctum* ao grupo de *middleware 'api'*, '\Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class', no ficheiro *./app/Http/Kernel.php*.
5. Adicionar o *Trait* '*Laravel\Sanctum\HasApiTokens*' ao modelo do *User*.

Uma vez instalado, foi preciso criar os *endpoints* necessários para dar suporte à autenticação, nomeadamente para registo (*register*), *login* e *logout*. Primeiramente, foi criado um controlador para a autenticação, denominado *AuthController*. Em seguida, explico algumas considerações acerca da implementação de cada uma das funções associadas a cada um destes *endpoints*.

- ***POST /api/register*** - este é o *endpoint* para registo dos utilizadores, ou seja, permite a criação de novos utilizadores. Requer o envio de informações pessoais, como nome, email, *username* e password (e a confirmação da mesma). Com base nessas informações são criados os novos '*User*' e o respetivo *token*, que são retornados como resposta ao pedido. Importa ainda referir que é usada a função *bcrypt()*, da *Facade* '*Hash*' do *Laravel*, que aplica um algoritmo de encriptação à *password* para que esta não seja armazenada em claro na BD, aumentando a segurança.
- ***POST /api/login*** - este é o *endpoint* para os utilizadores já registados efetuarem o início de sessão, onde devem ser enviados o email e a *password* corretas. Em primeiro lugar, é verificado se o email recebido consta na BD e depois se a *password* está correta. A verificação da *password* é feita recorrendo à função *Hash::check()* que recebe como argumentos a *password* enviada pelo utilizador e a *password* encriptada armazenada na BD e verifica se correspondem. Retorna um booleano a *true* se a password estiver correta ou a *false* se estiver errada. Se estiver errada é enviada uma resposta com o código de *status* HTTP 401 (*Unauthorized*). No caso de estar correta, é criado um novo *token* e como resposta é enviado o objeto '*User*' em questão e o *token*.
- ***POST /api/logout*** - este é o *endpoint* para o utilizador que tem sessão iniciada poder terminá-la. Como é lógico, requer autenticação, ou seja, tem de ser enviado o *token* da sessão atual. A execução deste *endpoint* envolve apenas a remoção desse *token* da BD.

Para os pedidos passarem na autenticação com sucesso, devem incluir no cabeçalho (*header*) "*Authorization*"com o valor "*Bearer <TOKEN>*". Por exemplo, *Authorization: Bearer 21|lmUtB2 WQucfJlZGFYFWFgnHYXk2wczg5Q*. O valor do *TOKEN* é o recebido nos *endpoints* de *login* ou *register*.

3.3.2 Autenticação com *API Key*

Para a autenticação através de *API Key* foi escolhido utilizar o módulo *ejarnutowski/laravel-api-key* [17].

À semelhança do módulo *Laravel Sanctum*, adicionar esta autenticação através de *API Key* requer algumas tabelas adicionais na base de dados para a gestão dessas chaves de acesso. O procedimento para instalar o módulo é o seguinte:

1. Instalar o módulo em si, através do comando *'composer require ejarnutowski/laravel-api-key'*. Ao instalar o módulo, são acrescentadas 3 novas migrações na respetiva diretoria, para a criação de 3 tabelas: '*api_keys*', '*api_key_access_events*' e '*api_key_admin_events*'.
2. No ficheiro *./config/app.php*, adicionar o provedor de serviço do *LaravelAPIKey*, *'Ejarnutowski\LaravelApiKey\Providers\ApiKeyServiceProvider::class'* ao final do *array* '*providers*'.
3. Executar '*php artisan vendor:publish*', que publica os ficheiros de migração.
4. Executar o comando *'php artisan migrate'* para criar as tabelas necessárias na base de dados.

Depois de instalado, já temos à disposição um conjunto de comandos para gerir as chaves da API. Os comandos são os seguintes:

- *'php artisan apikey:generate <NAME>'* - para gerar uma nova *API Key*, atribuindo um nome (*NAME*).

- *'php artisan apikey:deactivate <NAME>'* - para desativar a *API Key*, indicando o nome (*NAME*) da chave que se pretende desativar.
- *'php artisan apikey:activate <NAME>'* - para ativar a *API Key*, indicando o nome (*NAME*) da chave que se pretende ativar.
- *'php artisan apikey:delete <NAME>'* - para eliminar a *API Key*, indicando o nome (*NAME*) da chave que se pretende eliminar. Para manter o registo das chaves, apenas é feito *soft-delete*.
- *'php artisan apikey:list'* - para listar todas as *API Keys*. Usando a flag *-D* ou *–deleted* também são incluídas as chaves que foram apagadas.

Para os pedidos passarem na autenticação com sucesso, devem incluir um cabeçalho (*header*) cuja chave é "*X-Authorization*"e o valor é a "*API Key*"propriamente dita. Por exemplo, *X-Authorization: KuKMQbgZPv0PRC6GqCMlDQ7msVY75FrQ*.

3.4 *Endpoints* da API REST

Nesta secção, são explorados os *endpoints* desenvolvidos para formar a API REST do projeto *GesFoGO*. Um *endpoint* de uma API consiste num URL (*Uniform Resource Locator*) ou URI (*Uniform Resource Identifier*) específico que uma aplicação expõe para permitir a comunicação com outros componentes de software (clientes) e representa uma função, recurso ou serviço específico que a API disponibiliza através de pedidos HTTP. [18]

Cada *endpoint* é constituído por [19]:

- **URL/URI** - endereço web que especifica a localização do recurso ou operação. Por exemplo, `https://eelab.arditi.pt/gesfogo-api/sensors` podia representar um *endpoint* para aceder a informações relacionadas ao sensor.
- **Método HTTP** [20] - os *endpoints* têm sempre associado um método específico, de acordo com o tipo de operação a ser executada. Os métodos HTTP mais comuns e utilizados neste caso são:
  - ***GET*** - utilizado para obter dados de um recurso específico. É um método '*safe*', porque não altera o estado do servidor, portanto é utilizado para operações *read-only* e as respostas a este tipo de pedidos podem ser armazenadas em *cache*.
  - ***PUT*** - utilizado para enviar dados para o servidor para criar ou atualizar um recurso. É um método 'idempotente', ou seja, executar um determinado *endpoint* do tipo *PUT* **1** ou **n** vezes produz o mesmo resultado. Por exemplo, num pedido que faz alterar o nome da equipa responsável por um sensor, mesmo que seja executado várias vezes o valor desse atributo vai ser igual.
  - ***POST*** - utilizado para enviar dados para o servidor para criar ou atualizar um recurso. Mas, ao contrário do *PUT*, **não** é 'idempotente', portanto se um mesmo pedido for executado mais do que uma vez pode deixar o servidor inconsistente.
  - ***DELETE*** - utilizado para eliminar um recurso especificado. É um método 'idempotente'.
- **Parâmetros (*Parameters*))** - os *endpoints* podem aceitar parâmetros com informações adicionais como parte do URL do pedido ou como *query*. Por exemplo, o identificador (*id*) de um sensor (*/gesfogo-api/sensors/845*) ou para indicar o intervalo de tempo para o qual pretendemos

obter as imagens (*/gesfogo-api/images?initial_ date=2022-09-12 20:31:00&final_ date=2022-09-13 10:52:00*)

- **Cabeçalho (*Header*)** - contém informações adicionais sobre o pedido, como *tokens* para a autenticação, identificação do tipo de conteúdo (*Content Type*) ou escolha do idioma, fornecendo contexto ao servidor.
- **Corpo (*Body*)** - contém o *payload* do pedido, usado para as operações que criam ou atualizam recursos (*PUT* e *POST*), geralmente no formato JSON.

Na sequência de um pedido HTTP e da sua execução, o servidor envia uma resposta constituída por:

- **Corpo (*Body*) da Resposta** - contém os dados retornados pelo servidor. Quando é bem-sucedido, pode ser o recurso que foi solicitado ou a confirmação da operação. Em caso de erro, pode incluir uma mensagem de erro ou detalhes adicionais.
- **Código de *Status* da Resposta** [21] - código que indica o resultado da operação. Os códigos mais comuns são os seguintes:
  - **200 *Ok*** - o pedido foi bem sucedido. Este código pode ter significados diferentes conforme o método HTTP.
  - **201 *Created*** - o pedido foi bem sucedido e, como resultado, um novo recurso foi criado. Normalmente esta resposta é enviada em pedidos *POST*.
  - **400 *Bad Request*** - o servidor não pode processar o pedido devido a algo que é reconhecido como um erro do cliente (por exemplo, sintaxe do pedido mal formada, roteamento errado...).
  - **401 *Unauthorized*** - o servidor não pode processar o pedido porque o cliente não está autenticado.
  - **404 *Not Found*** - o servidor não consegue encontrar o recurso solicitado. Pode significar que o URL não foi reconhecido ou que o recurso identificado não existe.
  - **500 *Internal Server Error*** - O servidor deparou-se com uma situação com a qual não sabe o que fazer.

Na Secção 3.3 já foram explicadas as implementações para a autenticação. No entanto, falta referir como essa autenticação é aplicada aos *endpoints*. Como já foi mencionado, há dois grupos de *endpoints* e cada um tem um tipo de autenticação diferente, os '*GUI Endpoints*' utilizam a autenticação através de *login* do utilizador com o módulo *Laravel Sanctum* e os '*Info Reception Endpoints*' utilizam autenticação por *API Key*. A atribuição do tipo correto de autenticação a cada *endpoint* foi feita no ficheiro *./routes/api.php* onde são declarados todos os *endpoints*, através da divisão por grupos de rotas (*Route Groups* [22]) do *Laravel*, que permitem agrupar várias rotas para aplicar *middlewares* ou outras configurações a todas as rotas desse grupo. Em seguida é mostrada uma captura de ecrã, Figura 10, com essa configuração dos *middlewares* da autenticação e alguns exemplos de definição de *endpoints*. Na linha 49, é declarado um grupo de rotas com a configuração do *middleware* '*auth:sanctum*' onde são declarados os '*GUI Endpoints*' e na linha 92, é declarado o outro grupo de rotas com a configuração do *middleware* '*auth.apikey*' onde são declarados os '*Info Reception Endpoints*'.

```
47  //Routes to be used by GUI (Sanctum Authentication)
48
49  Route::group(['middleware' => ['auth:sanctum']], function () {
50      Route::get( uri: '/sensors', [SensorController::class, 'index']);
51      Route::get( uri: '/sensors/{id}', [SensorController::class, 'show']);
52      /** ...*/
88  });
89
90  //Routes to be used by ULPGC (API Key Authentication)
91
92  Route::group(['middleware' => ['auth.apikey']], function () {
93      Route::post( uri: '/sensor/{sn}/georef-polygons', [GeorefPolygonController::class, 'store']);
94      Route::post( uri: '/sensor/{sn}/images', [ImageController::class, 'store']);
95      /** ...*/
109 });
```

Figura 10. Configuração dos *Middlewares* para autenticação

Além de mostrar a configuração da autenticação para os *endpoints*, também é mostrado na Figura 10 como cada rota (*Route*) é definida. Usando o *endpoint* '*Get Sensor By Id*' (linha 51) como exemplo, começamos por invocar a *facade 'Route'* e escolher o método HTTP correto, neste caso é um *GET*. Em seguida indicamos o URI, que pode ter parâmetros, neste caso '*id*', cujo seu valor é passado como argumento à função *show()*. Por fim, é feito o mapeamento deste URI para a função *show()* do controlador '*SensorController*'.

No total, a API REST do *GesFoGO* conta com 34 *endpoints*, que estão listados na Figura 11. Para cada *endpoint* é indicado um número identificador para facilitar referências ao mesmo ao longo deste relatório, seguido do método HTTP, o URI e o nome que lhe foi atribuído. Os *endpoints* **1** ao **3** são relativos à autenticação (por *login* de utilizador) e já foram explicados na Subsecção 3.3.1, do **4** ao **27** são os '*GUI Endpoints*' e do **28** ao **34** são os '*Info Reception Endpoints*'.

Alguns dos *endpoints* têm uma implementação bastante simples, isto porque correspondem a operações simples de CRUD (*Create*, *Read*, *Update* e *Delete*), cujos modelos do *Laravel* têm como parte do ORM *Eloquent*. Os *endpoints* 4, 5, 7, 14 e 15 enquadram-se neste caso, já que são operações diretas à BD e envolvem uma única tabela.

Há ainda alguns *endpoints*, 6, 8, 9, 17, 19, 22 e 23, que também são simples mas que requerem algumas operações a mais, porque a obtenção dos dados envolve alguma filtragem, obtenção de informação de mais do que uma tabela e/ou métodos específicos do SQL para lidar com certos tipos de dados, por exemplo, polígonos de georreferenciação. As rotas 9 (*Update Team*) e 23 (*Update Notification*) correspondem a pedidos do tipo PUT, ou seja, para atualizar o valor de determinados atributos, respetivamente, atualizar o nome da equipa responsável pela instalação do sensor especificada pelo *id* e para marcar a notificação especificada com o *id* como resolvida. As rotas 6, 8 e 19 são pedidos de leitura mas que retornam informação de mais do que uma tabela, usando as relações já referidas na Secção 3.2, na Figura 8, para fazer *Eager Loading* [23], através do método *with()* do *Eloquent*, que os acrescenta como objetos *nested* à resposta do pedido. Na Figura 12 está um exemplo da rota 8 (*Get Sensor Configurations*), em que é retornada a informação da instalação de sensor (da tabela *sensor_ deploy*) especificado pelo *id* e acrescenta a informação de *'sensor'* (da tabela *sensor*) e de *'configurations'* (da tabela *sensor_ configuration*). A resposta resultante consiste num objeto JSON com os atributos da própria instalação do sensor com um objeto *nested* chamado *'sensor'* com a informação do sensor e outro chamado *'configurations'* que consiste num *array* com as configurações que tem associadas.

```
1 - POST /gesfogo-api/register - Register New User
2 - POST /gesfogo-api/login - Login User
3 - POST /gesfogo-api/logout - Logout User

4 - GET /gesfogo-api/regions - Get Regions
5 - POST /gesfogo-api/regions - Create Region
6 - GET /gesfogo-api/sensors - Get Sensors
7 - GET /gesfogo-api/sensors/{id} - Get Sensor By Id
8 - GET /gesfogo-api/sensor-deploy/{id}/configuration - Get Sensor Configurations
9 - PUT /gesfogo-api/sensor-deploy/{id}/team - Update Team
10 - GET /gesfogo-api/sensor-deploy/{id}/images - Get Images
11 - GET /gesfogo-api/sensor-deploy/{id}/georef-polygons - Get Georef Polygons
12 - GET /gesfogo-api/sensor-deploy/{id}/drawn-polygons - Get Drawn Polygons
13 - POST /gesfogo-api/sensor-deploy/{id}/drawn-polygons - Create Drawn Polygon(s)
14 - DELETE /gesfogo-api/sensor-deploy/{id}/drawn-polygons/{polygon_id} - Delete Drawn Polygon
15 - GET /gesfogo-api/locations - Get Locations
16 - POST /gesfogo-api/locations - Create Location
17 - GET /gesfogo-api/locations/{id} - Get Location By Id
18 - POST /gesfogo-api/locations/{id}/range-polygons - Create Range Polygon(s)
19 - GET /gesfogo-api/locations/{id}/alternative-locations - Get Alternative Locations
20 - POST /gesfogo-api/locations/{id}/alternative-locations - Create Alternative Location
21 - GET /gesfogo-api/polygons - Get Polygons
22 - GET /gesfogo-api/notifications - Get Notifications
23 - PUT /gesfogo-api/notifications/{id} - Update Notification
24 - POST /gesfogo-api/sensor-deploy/{id}/capture-frequency-command - Capture Frequency Command
25 - POST /gesfogo-api/sensor-deploy/{id}/pitch-command - Pitch Command
26 - POST /gesfogo-api/sensor-deploy/{id}/sweep-mode-command - Sweep Mode Command
27 - POST /gesfogo-api/sensor-deploy/{id}/yaw-command - Yaw Command

28 - POST /gesfogo-api/sensor/{sn}/georef-polygons - Create Georef Polygon(s)
29 - POST /gesfogo-api/sensor/{sn}/images - Create Image(s)
30 - POST /gesfogo-api/sensor/{sn}/updates - Update Sensor Deploy
31 - POST /gesfogo-api/update-capture-frequency/{sn} - Update Capture Frequency
32 - POST /gesfogo-api/update-pitch/{sn} - Update Pitch
33 - POST /gesfogo-api/update-sweep-mode/{sn} - Update Sweep Mode
34 - POST /gesfogo-api/update-yaw/{sn} - Update Yaw
```

Figura 11. Lista dos *Endpoints* que constituem a API do *GesFoGO*

```
74      public function show($id)
75      {
76          return SensorDeploy::where('id', $id)->with('sensor', 'configurations')->get()->first();
77      }
```

Figura 12. Excerto de código da implementação da rota *Get Sensor Configurations*

No caso das rotas 11, 12, 17, 19 e 22 estão implementadas *queries* que têm alguma complexidade acrescida, desde filtros usando a cláusula *WHERE*, através do método *where()* do *Eloquent*, até à cláusula *INNER JOIN* usando o método *join()* do *Eloquent*. Na Figura 13 é mostrado o código implementado para o *endpoint* 17 (*Get Location By Id*) que retorna a informação da localização especificada pelo *id*, juntamente com a lista dos Polígonos de Cobertura (*Range Polygons*) associados. Este exemplo inclui o *INNER JOIN* das tabelas '*location*' e '*range_polygon*' quando '*location.id*' é igual ao valor de '*range_polygon.location_id*', e no final ainda é usada a cláusula *WHERE* para obter apenas os polígonos associados à localização em questão. De modo a obter os polígonos no formato de texto adequado para utilizar na GUI, foi necessário utilizar a função *ST_AsText()* do SQL, no entanto, como essa função não faz parte do *Eloquent* foi necessário utilizar o método *DB::raw()* [24] (linha 176) para injetar código/expressões SQL personalizadas diretamente na *query*, continuando a usufruir da funcionalidade *query builder* do *Laravel*.

```
170     public function show($id)
171     {
172         $location = Location::find($id);
173
174         $location['range_polygons'] = new Collection(
175             Location::join('range_polygon', 'location.id', '=', 'range_polygon.location_id')
176                 ->select(['range_polygon.*', DB::raw( value: 'ST_AsText(range_polygon.coordinates) as coordinates')])
177                 ->where('location_id', $id)->get()
178         );
179
180         return $location;
181     }
```

Figura 13. Excerto de código da implementação da rota *Get Location By Id*

Os *endpoints* 10 (*Get Images*) e 21 (*Get Polygons*) servem para obter imagens e polígonos (linhas de fogo e de sombra) independentemente de qual o sensor que está associado e são utilizados para dar suporte à funcionalidade de *Timeline*, que será melhor explicada mais à frente na Secção 3.11. Para isso, recebem como parâmetros uma data inicial (*initial_date*) e uma data final (*final_date*) para retornar apenas as amostras cuja captura foi realizada entre a data inicial e final. Nestes dois *endpoints* foi necessário usar o método *DB::raw()* para escrever toda a *query* SQL "por extenso", ou seja, sem usar os métodos do *Eloquent*, porque dessa forma não estava a ser possível obter os dados como pretendia. Na Figura 14, mostrada em seguida, está o código da implementação do *endpoint* 10 (*Get Images*). Na *query* são selecionados os atributos a incluir na resposta, sendo que no caso das imagens (visíveis e térmicas) foi necessário usar a função *TO_BASE64()*, que converte a string armazenada na BD para uma string em base 64. É feito o *INNER JOIN* para juntar as informações das tabelas '*capture*' e '*image*', e através da cláusula *BETWEEN* do SQL é feita a filtragem das amostras relevantes com base no intervalo de tempo. Por fim, é feita a ordenação por ordem crescente do *id*. A implementação da rota 21 é muito semelhante, mas são feitas duas *queries*, uma para os polígonos de georreferenciação (*georef_polygon*) e uma para os polígonos desenhados (*drawn_polygon*).

As rotas 13, 16, 18 e 20 servem para a criação de recursos e têm várias características comuns no que diz respeito à sua implementação, nomeadamente o facto de envolver a criação de tuplos em mais do que uma tabela e receber um *array* de objetos como parâmetro no corpo (*body*) do pedido. Estes aspetos aumentam a complexidade da implementação destes *endpoints*. Para exemplificar a implementação destes *endpoints*, vai ser usada a rota 13 (*Create Drawn Polygon(s)*) que permite

```php
public function index(Request $request, $id)
{
    $initial_date = $request->query( key: 'initial_date');
    $final_date = $request->query( key: 'final_date');

    return new Collection(
        DB::select(DB::raw( value: 'SELECT image.id, capture_id, sensor_deploy_id, timestamp, temperature, pressure, humidity, wind_direction, wind_speed,
        yaw, pitch, roll, TO_BASE64(visible_image) as visible_image, TO_BASE64(thermal_image) as thermal_image, image.created_at, image.updated_at
        FROM capture INNER JOIN image ON capture.id = image.capture_id
        WHERE timestamp BETWEEN \'. $initial_date . '\' AND \'' . $final_date . '\' AND sensor_deploy_id = ' . $id . ' order by image.id asc'))
    );
}
```

Figura 14. Excerto de código da implementação da rota *Get Images*

a criação dos polígonos desenhados por operadores do sistema através da GUI, apresentada na Figura 15.

```php
public function store(Request $request, $id)
{
    $request->validate([
        'timestamp' => 'required|string',
        'polygons' => 'required|array',
        'polygons.*.type' => 'required|in:fire,shadow',
        'polygons.*.coordinates' => 'required|string'
    ]);

    try {
        DB::beginTransaction();
        $capture = Capture::create([
            'sensor_deploy_id' => $id,
            'timestamp' => $request->get( key: 'timestamp'),
            'type' => 'drawn_polygon'
        ]);
        $polygons = [];
        $polygonArray = $request->get( key: 'polygons');

        for ($x = 0; $x < count($polygonArray); $x++) {
            $polygon = $request->get( key: 'polygons')[$x];
            $polygonCoordinates = $polygon['coordinates'];
            $item = DrawnPolygon::create([
                'capture_id' => $capture->id,
                'type' => $polygon['type'],
                'coordinates' => DB::raw( value: 'ST_PolyFromText(\'' . $polygonCoordinates . '\')')
            ]);
            $item['coordinates'] = $polygonCoordinates;
            array_push( &array: $polygons, $item);
        }
        $capture['polygons'] = $polygons;

        DB::commit();
        return $capture;
    } catch (\Throwable $th) {
        DB::rollBack();
        return response()->json(['message' => $th->getMessage()], status: 500);
    }
}
```

Figura 15. Excerto de código da implementação da rota *Create Drawn Polygon(s)*

Primeiramente, é feita a validação dos argumentos recebidos no corpo do pedido, usando a função *validate()* [25] disponibilizada pela classe *Illuminate\Http\Request* do *Laravel*, que recebe as regras de validação para cada atributo. Se a validação for bem sucedida, a execução continua normalmente, caso contrário, é lançada uma exceção (*ValidationException*) que envia automaticamente uma resposta adequada ao utilizador. No caso, todos os parâmetros são obrigatórios, portanto, a regra *'required'* está presente em todos, *'timestamp'* e *'coordinates'* são do tipo '*string*',

'*polygons*' é um *array* e *'type'* é um *enum* que pode ter os valores '*fire*' ou '*shadow*'. O parâmetro '*polygons*' é um *array* de objetos, em que cada objeto possui os atributos *type* e *coordinates*, por isso são declarados como '*polygons.*.type*' e '*polygons.*.coordinates*', respetivamente.

Em segundo lugar, como são feitas várias operações que provocam alterações a mais do que uma tabela, para manter a consistência e integridade dos dados foi necessário implementar transações. Uma transação permite garantir a atomicidade de um conjunto de operações, isto é, este conjunto de operações funciona como uma única operação, se ocorrer algum erro durante a execução de alguma das operações, é feito o *rollback* para não alterar o estado do servidor. Nesse sentido, o código referente a estas operações é executado dentro de um *try-catch*, para poder determinar o que acontece se ocorrer algum erro. A transação é definida recorrendo a 3 métodos disponibilizados pela *facade* 'DB' do *Laravel* [26]. Para iniciar a transação é chamado o método *DB::beginTransaction()* e no final é chamado o método *DB::commit()* para confirmar as alterações. Dentro do bloco *catch* é chamado o método *DB::rollBack()* que desfaz as alterações que já tinham ocorrido antes do erro.

Quanto às operações em si, primeiro é criada uma instância de '*Capture*', à qual todos os polígonos desenhados vão estar associados. Uma vez que é recebido um *array* de polígonos, é feito um ciclo *for* para a criação dessas instâncias que, após criadas, são inseridas num *array* auxiliar (*$polygons*) para ser agregado em *$capture* e ser retornado como resposta. Importa ainda mencionar que, para armazenar os polígonos corretamente na BD com o tipo '*Polygon*' [13], foi necessário recorrer à função *ST_PolyFromText* do SQL para converter a string que é recebida com um formato específico ("*POLYGON((-16.91875300 32.74249946,-16.918774 32.741804,...))*") para o tipo certo.

No caso do envio de comandos para os sensores, são utilizados os *endpoints* 24, 25, 26 e 27 que servem para solicitar as alterações das configurações dos sensores, a frequência de captura, a inclinação (*pitch*), o modo de varrimento e a orientação (*yaw*), respetivamente. Estas rotas são responsáveis por fazer o registo da solicitação desses comandos e o envio do comando propriamente dito para o componente *'Sensor Wrapper'* (Figura 1), desenvolvido pela ULPGC, que é uma API REST que recebe os comandos e os envia para os sensores.

A implementação destes *endpoints* é muito semelhante, visto que realizam a mesma sequência de operações mas com parâmetros e tabelas distintas. Para exemplificar, vai ser usado o *endpoint* 24 (*Capture Frequency Command*), na Figura 16. Inicialmente é verificado em que estado se encontra a instalação do sensor, porque apenas é possível realizar as configurações se estiver no estado '*ONGOING*', depois é feita a verificação dos parâmetros recebidos, que neste caso é um *integer*, obrigatório (*required*) e tem como valor mínimo 1 minuto. Depois disso, foi necessário recorrer à utilização de transações para manter a consistência e integridade do sistema durante a execução das operações. A transação começa por criar uma instância de '*Command*' e uma instância de '*ChangeCaptureFrequency*'. Em seguida, é usada a *facade* HTTP [27] do *Laravel*, para fazer o pedido à API da ULPGC que recebe os comandos para os sensores. Os pedidos ao servidor da ULPGC requerem autenticação através de um *token* (*APP_ULPGC_API_TOKEN*) que nos foi fornecido e que fica guardado no ficheiro *.env*, juntamente com o *host* do servidor (*APP_ULPGC_API_HOST*). A autenticação é feita através do envio do cabeçalho '*Authorization*', cujo valor é "*TOKEN <APP_ULPGC_API_TOKEN>*". Depois é construído o URI, que é específico para cada tipo de comando, onde é adicionado o número de série ($serial_number) e por fim, são enviados os parâmetros, neste caso, o novo valor para a frequência de captura (multiplicada por 60, porque a API da ULPGC recebe esse parâmetro em segundos). Para finalizar, é

verificado o *status code* desse pedido, que se não for bem sucedido lança uma exceção que provoca o *rollback*, para o servidor não ficar com informação inconsistente.

```php
public function store(Request $request, $id)
{
    $deploy = SensorDeploy::find($id);

    if ($deploy->state === 'ENDED' || $deploy->state === 'PENDING') {
        return response()->json(['message' => 'A command can only be requested if deploy state is \'ONGOING\'.'], status: 400);
    }

    $request->validate(['capture_frequency' => 'required|integer|min:1']);
    $sensor = Sensor::find($deploy->sensor_id);
    $serial_number = $sensor->serial_number;

    try {
        DB::beginTransaction();
        $command = Command::create([
            'sensor_deploy_id' => (int) $id,
            'timestamp' => now()->toDateTimeString( unitPrecision: 'second'),
            'type' => 'change_capture_frequency'
        ]);

        $command['capture_frequency'] = ChangeCaptureFrequency::create([
            'command_id' => $command->id,
            'capture_frequency' => $request->get( key: 'capture_frequency')
        ]);

        $response = Http::withHeaders([
            'Authorization' => 'TOKEN  ' . env( key: 'APP_ULPGC_API_TOKEN')
        ])->post( url: env( key: 'APP_ULPGC_API_HOST') . '/sensor-controller/update-capture-frequency/' . $serial_number, [
            'capture_frequency' => $request->get( key: 'capture_frequency') * 60]);

        if ($response->status() !== 201) { throw new \Exception( message: "Command didn't work"); }

        DB::commit();
        return $command;
    } catch (\Throwable $th) {
        DB::rollBack();
        return response()->json(['message' => 'Internal Server Error'], status: 500);
    }
}
```

Figura 16. Excerto de código da implementação da rota *Capture Frequency Command*

Os *endpoints* 28 a 34, para receção de informação proveniente dos sensores, utilizam como identificador o número de série (*{sn}*) do mesmo em vez dos identificadores internos como nos restantes *endpoints*. Devido a isso, a primeira coisa a fazer na implementação destas rotas é validar os argumentos recebidos, incluindo o número de série do sensor, ou seja, é necessário verificar se existe algum sensor com aquele número de série e, consequentemente, a instalação mais recente desse sensor (’*currentDeploy*’). No caso de não ser encontrado é enviada uma resposta conforme. Na Figura 17 é apresentado um exemplo dessas validações para o *endpoint* 29 (*Create Image(s)*), que recebe várias informações meteorológicas, localização geográfica, carga da bateria e as imagens capturadas juntamente com as configurações do sensor naquele momento.

Relativamente às operações específicas dos *endpoints* 28 (*Create Georef Polygon(s)*) e 29 (*Create Image(s)*), seguem uma estrutura bastante similar à da rota 13 (*Create Drawn Polygon(s)*), o que muda são as tabelas da BD envolvidas e os respetivos atributos. No final destes *endpoints* é feita a atualização/confirmação dos valores de *latitude*, *longitude* e *battery_charge*, que são enviados juntamente com as capturas para manter estes valores o mais atuais possível.

```php
public function store(Request $request, $sn)
{
    $request->validate([
        'temperature' => 'required|integer',
        'pressure' => 'required|integer',
        'humidity' => 'required|integer|min:0|max:100',
        'wind_direction' => 'required|in:N,NE,E,SE,S,SW,W,NW',
        'wind_speed' => 'required|numeric',
        'latitude' => 'required|string',
        'longitude' => 'required|string',
        'battery_charge' => 'required|integer|min:0|max:100',
        'images' => 'required|array',
        'images.*.visible_image' => 'required|string',
        'images.*.thermal_image' => 'required|string',
        'images.*.yaw' => 'required|integer|min:0|max:360',
        'images.*.pitch' => 'required|integer|min:-20|max:20',
        'images.*.roll' => 'required|integer|min:0|max:360'
    ]);

    $currentDeploy = Sensor::where('serial_number', $sn)->with('currentDeploy')->get()->first();

    if ($currentDeploy === null) {
        return response()->json(['message' => 'There is no Sensor Deploy for that sensor Serial Number.'], status: 404);
    }
```

Figura 17. Excerto de código das validações da rota *Create Image(s)*

O *endpoint* 30 (*Update Sensor Deploy*) é o mais complexo de todos, porque envolve toda a lógica de gestão das Instalações de Sensores. É usado para receber atualizações regulares relativas ao sensor determinado pelo número número de série (*{sn}*) recebido no URI. Tem como objetivo confirmar que o sensor está configurado e a funcionar corretamente. De acordo com o estado atual da instalação ('*PENDING*', '*ONGOING*' ou '*ENDED*') o comportamento é diferente. A implementação deste *endpoint* encontra-se ilustrada na Figura 18 e será explicada detalhadamente, em seguida.

Se estiver no estado '***ONGOING***' pode ocorrer um destes cenários: o sensor está na mesma localização e apenas atualiza o estado da bateria ou o sensor mudou de localização e é criada uma nova instalação (*deploy*) do sensor e a respetiva configuração. Para determinar qual desses cenários é o atual é necessário comparar as coordenadas que estão atualmente associadas ao *deploy* do sensor em questão, com as coordenadas acabadas de receber (linhas 211 e 212). Essa verificação é feita recorrendo a um *if*, e as coordenadas são comparadas utilizando a função *abs()* do PHP, que retorna o valor absoluto do número recebido como parâmetro, que no caso é o resultado da diferença entre a latitude atual e a recebida e a diferença entre longitude atual e a recebida. Depois estes valores são comparados com 0.0000, que na prática significa que se o número das unidades for igual e as primeiras 4 casas decimais das coordenadas forem iguais consideramos que o sensor está na mesma localização. A razão para a comparação ser feita até à 4ª casa decimal é que já nos dá uma precisão de 11,1 metros [28], que foi o considerado razoável para se assumir como sendo a mesma localização, e consequentemente o mesmo *deploy*. Feita esta comparação, se a localização for considerada igual apenas é atualizado o nível da bateria, caso contrário é realizada uma sequência de operações para terminar o *deploy* atual (linhas 221-223), criar uma nova instância de '*SensorDeploy*' e uma instância de '*SensorConfiguration*' que são preenchidas usando as informação recebidas no pedido e algumas informações copiadas das instâncias de '*SensorDeploy*' e '*SensorConfiguration*' antigas, respetivamente, '*$deploy*' e '*$oldConfiguration*'. Na Figura 18, o código da criação destas duas instâncias está colapsado para evitar que a imagem ficasse demasiado grande (linhas 225 e 241).

Se estiver no estado '***PENDING***' e receber um pedido, o *deploy* passa para o estado '*ONGOING*' e os valores dos seus atributos são atualizados com as informações recebidas no pedido

```
208         $deploy = SensorDeploy::find($deploy_id);
209
210         if (strcmp($deploy->state, string2: 'ONGOING') === 0) {
211             if (abs( num: $deploy->latitude - floatval($request->get( key: 'lat'))) === 0.0000
212                 && abs( num: $deploy->longitude - floatval($request->get( key: 'lon'))) === 0.0000){
213                 $deploy->battery_charge = (int) $request->get( key: 'battery');
214                 $deploy->save();
215                 return $deploy;
216             }
217             try {
218                 DB::beginTransaction();
219                 $oldConfiguration = SensorConfiguration::where('sensor_deploy_id', $deploy_id)->orderBy('timestamp', 'desc')->first();
220
221                 $deploy->end_timestamp = now()->toDateTimeString( unitPrecision: 'second');
222                 $deploy->state = 'ENDED';
223                 $deploy->save();
224
225                 $newDeploy = SensorDeploy::create([...]);
241                 $newConfig = SensorConfiguration::create([...]);
252
253                 DB::commit();
254                 return $newDeploy;
255             } catch (\Throwable $th) {
256                 DB::rollBack();
257                 return response()->json(['message' => 'Internal Server Error'], status: 500);
258             }
259         } else if (strcmp($deploy->state, string2: 'PENDING') === 0) {
260             $deploy->latitude = $request->get( key: 'lat');
261             $deploy->longitude = $request->get( key: 'lon');
262             $deploy->approximate_latitude = $request->get( key: 'lat');
263             $deploy->approximate_longitude = $request->get( key: 'lon');
264             $deploy->height = $request->get( key: 'alt');
265             $deploy->start_timestamp = now()->toDateTimeString( unitPrecision: 'second');
266             $deploy->battery_charge = (int) $request->get( key: 'battery');
267             $deploy->state = 'ONGOING';
268             $deploy->save();
269             return $deploy;
270         } else if (strcmp($deploy->state, string2: 'ENDED') === 0) {
271             return response()->json(['message' => 'The current latest deploy for that sensor is in \'ENDED\' state, it can\'t be changed.'], status: 400);
272         }
```

Figura 18. Excerto de código da implementação da rota *Update Sensor Deploy*

(linhas 259-269). Por fim, se estiver no estado '***ENDED***' e receber um pedido para este *endpoint* é retornada uma resposta de erro porque não é possível atualizar um *deploy* que já terminou.

Para concluir, os *endpoints* 31 a 34 servem como uma confirmação por parte dos sensores de que alguma das configurações foi concretizada. Isto é, quando um utilizador solicita a alteração de alguma configuração do sensor, são usados os *endpoints* 24 a 27 para fazer o registo dos comandos, mas isso não altera a configuração em si. Os *endpoints* 31 a 34 é que fazem a atualização das configurações dos sensores por iniciativa dos sensores. Para exemplificar estas operações, na Figura 19 é mostrada a implementação da rota 34 (*Update Yaw*), que faz a atualização da configuração de orientação segundo o eixo vertical (*yaw*) do sensor com o número de série indicado. Começa por validar os parâmetros recebidos, nomeadamente os valores mínimo (yaw_min) e máximo (yaw_max) de *yaw*, que devem ser um número entre 0 e 360 graus, e também verifica se o número de série recebido ($\{sn\}$) corresponde a algum sensor e ao respetivo *deploy*. Depois é obtida a configuração antiga daquele sensor para usar a sua informação. Como os parâmetros são opcionais, foi necessário usar um operador ternário para usar o valor antigo da configuração caso esta não tenha sido recebida (linhas 246 e 247). Depois é criada uma nova instância de *SensorConfiguration*, que é preenchida agregando as informações recebidas e a configuração anterior. A razão de ser criada uma nova instância é manter um registo das configurações na base de dados para, por exemplo, saber como mostrar estas informações na GUI, de acordo com o instante temporal que está a ser mostrado. Por fim, é retornada a nova configuração.

```php
public function updateYaw(Request $request, $sn)
{
    $request->validate([
        'yaw_min' => 'numeric|min:0|max:360',
        'yaw_max' => 'numeric|min:0|max:360'
    ]);

    $currentDeploy = Sensor::where('serial_number', $sn)->with('currentDeploy')->get()->first();

    if ($currentDeploy === null) {
        return response()->json(['message' => 'There is no Sensor Deploy for that sensor Serial Number.'], status: 404);
    }

    $old_configuration = $currentDeploy->currentDeploy[0]->latest_configuration;

    $yaw_min = ($request->get( key: 'yaw_min') === null) ? $old_configuration->yaw_min : $request->get( key: 'yaw_min');
    $yaw_max = ($request->get( key: 'yaw_max') === null) ? $old_configuration->yaw_max : $request->get( key: 'yaw_max');

    $configuration = SensorConfiguration::create([
        'sensor_deploy_id' => $old_configuration->sensor_deploy_id,
        'timestamp' => now()->toDateTimeString( unitPrecision: 'second'),
        'yaw_min' => $yaw_min,
        'yaw_max' => $yaw_max,
        'pitch_min' => $old_configuration->pitch_min,
        'pitch_max' => $old_configuration->pitch_max,
        'roll' => $old_configuration->roll,
        'capture_frequency' => $old_configuration->capture_frequency,
        'sweep_status' => $old_configuration->sweep_status
    ]);

    return $configuration;
}
```

Figura 19. Excerto de código da implementação da rota *Update Yaw*

3.5 Teste/Validação dos *Endpoints* da API

Nesta secção, é abordada a forma como foram feitos os testes ao funcionamento, segurança e comunicação da API REST com os restantes componentes do sistema, de forma contínua, durante todo o desenvolvimento do projeto. Para o teste da API propriamente dita e consequentemente da sua comunicação com a Base de Dados, foi utilizado o programa *Postman*, uma ferramenta para teste dos *endpoints* da API, que fornece uma interface intuitiva para enviar pedidos HTTP, receber as respostas, inspecionar os resultados, entre outras funções.

Logo numa fase inicial do desenvolvimento da API, foram adicionados dados fictícios na BD com a finalidade de testar o correto funcionamento de cada rota. Depois disso, usando o *Postman*, comecei por criar uma coleção de *endpoints* para este projeto, chamada '*GesFoGO*'. Em seguida, criei um ambiente ('*environment*') do *Postman* que permite criar algumas variáveis, às quais podemos atribuir valores de forma dinâmica e usar essas variáveis nos vários *Endpoints*. Isto facilita bastante porque evita ter esses valores *hard-coded*, que normalmente mudam durante o período de desenvolvimento. Na Figura 20 é mostrada uma captura de ecrã do *Postman*, onde é possível ver a coleção 'GesFoGO' com os respetivos *endpoints* no lado esquerdo e no lado direito o 'ambiente' criado com as variáveis '*baseUri*'(URI base do servidor da API), '*baseUriULPGC*' (URI base do servidor da ULPGC), '*API_KEY*' (chave para autenticação com *API Key*) e '*session_token*' (*token* para autenticação com *login* de utilizador).

Ainda na Figura 20, é possível ver o exemplo do *endpoint* '*Get Images*' que, à semelhança de todos os outros, utiliza a variável '*baseUri*' antes do URI específico daquela rota. Este *endpoint* tem ainda outra particularidade, porque pode receber parâmetros de '*query*' (*initial_date* e *final_date*), que são definidos no separador '*Params*'.

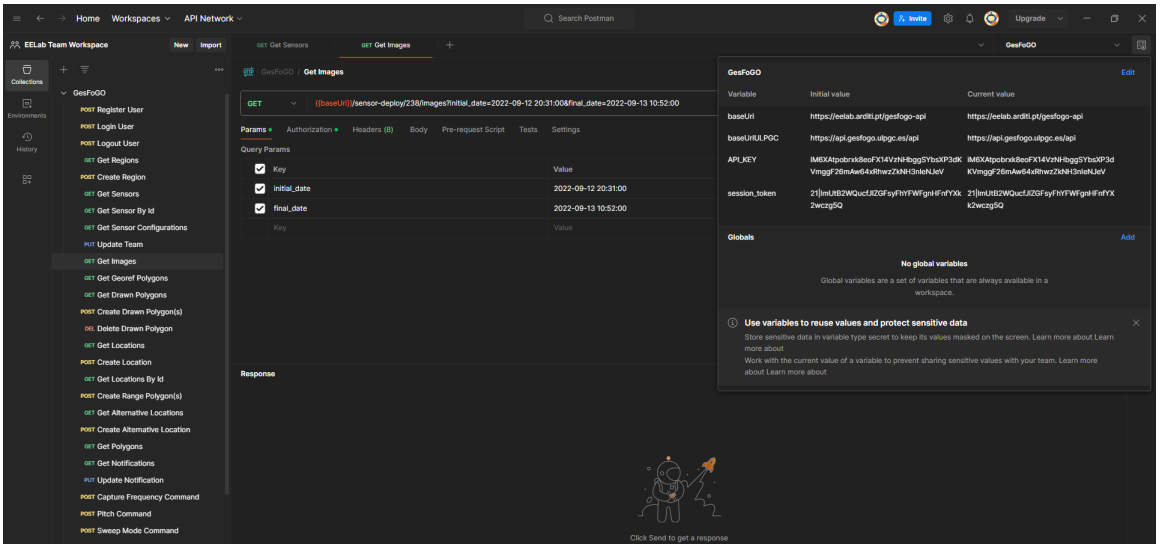


Figura 20. Captura de ecrã com 'coleção' e 'ambiente' do *Postman*

Relativamente às autenticações com *login* de utilizador, que utilizam o valor da variável *session_token*, de modo a manter essa variável sempre atualizada foram usados os *scripts* de teste do *Postman*, que são definidos no separador '*Tests*', como mostra a Figura 21. Estes *scripts* são feitos em *javascript* e podem usar variáveis dinâmicas, realizar asserções nas respostas aos pedidos e transmitir dados entre pedidos. Neste caso, ilustrado na Figura 21, os *scripts* foram usados nos *endpoints* '*Register User*' e '*Login User*' que, após ser feito um pedido a um destes *endpoints* extraímos o *session_token* da resposta e armazenamos esse valor na respetiva variável de ambiente que foi definida. Este *script* armazena a resposta obtida numa variável auxiliar (*jsonData*) e depois atribui o valor do *token* à variável *session_token*. Já no *endpoint* '*Logout User*' é feita a remoção desse valor da variável de ambiente.

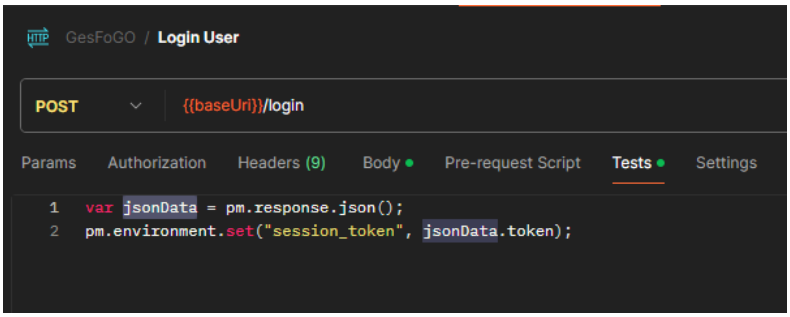


Figura 21. *Script* definido para armazenar o *session_token*

Depois de realizada esta configuração inicial, conforme fazia a implementação de cada *endpoint*, o respetivo *endpoint* era criado no *Postman*, o que ajudava bastante a verificar o correto funcionamento de cada rota durante o seu desenvolvimento, e ainda, permitia que se executasse toda a coleção para confirmar que os novos *endpoints* não tinha causado problemas aos já existentes. Para adicionar a autenticação por *login* de utilizador (*token*) aos pedidos é usado o separador '*Authorization*' selecionando o tipo '*Bearer Token*' e atribuindo o valor da variável '*session_token*', como é possível ver na Figura 22.

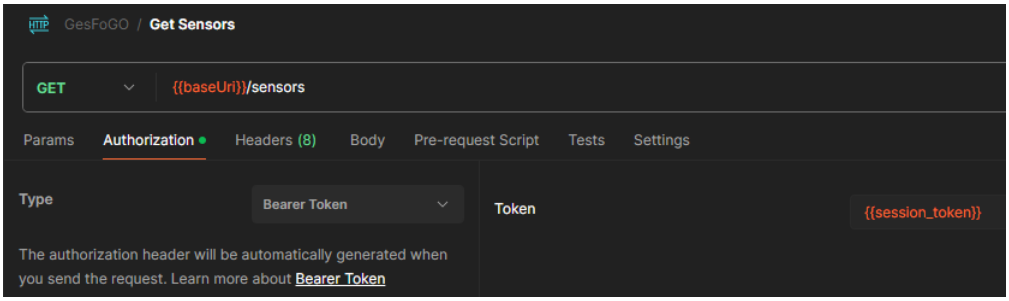


Figura 22. Captura de ecrã da autenticação por *login* no *Postman*

No caso da autenticação com *API Key*, como utiliza um cabeçalho com uma chave específica, estes valores são configurados no separador '*Headers*', a chave é '*X-Authorization*' e o valor é a própria *API Key*, que está armazenada na variável de ambiente. Este caso é ilustrado na Figura 23.

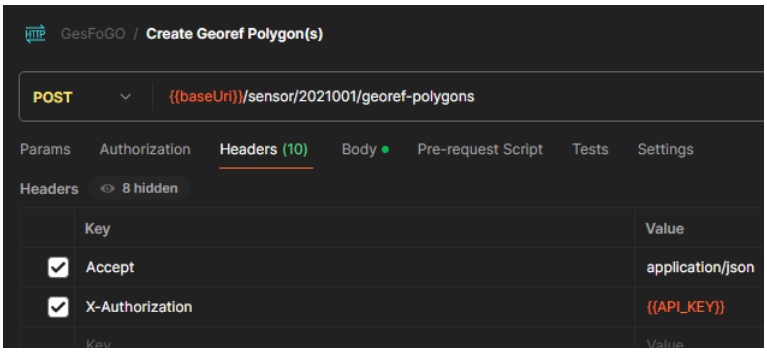


Figura 23. Captura de ecrã da autenticação por *API Key* no *Postman*

No caso dos *endpoints* que requerem o envio de informação no corpo (*body*) do pedido (*PUT* e *POST*), essa configuração é feita no separador '*Body*', como é possível ver no exemplo da Figura 24, relativo ao *endpoint* '*Create Drawn Polygon(s)*'. Neste exemplo, o *body* é definido em JSON, mas também pode ser definido de forma semelhante aos cabeçalhos, como na Figura 23. Além disso, na Figura 24, é exemplificada a resposta ao pedido, na parte inferior da imagem, onde é apresentado o código de *status* (201 *Created*) e abaixo a resposta no formato JSON, onde consta a nova instância de *Capture* e as respetivas instâncias de *DrawnPolygon* no *array* '*polygons*'.

Para além de testar os *endpoints* para os casos em que o pedido deve ser bem sucedido, também é necessário testar o comportamento de cada *endpoint* nos casos em que não deve ter sucesso, seja porque o pedido está mal formulado ou devido à falta de autenticação. Para verificar se os *endpoints* estão a agir corretamente quando recebem pedidos mal formulados foram feitos testes, como enviar tipos de dados errados (por exemplo, *string* onde deveria receber um *integer*), parâmetros com nomes mal escritos/errados, não enviar parâmetros obrigatórios, entre outras, e verificar se as respostas e os códigos de *status* refletiam esse erros. Em relação à validação do correto funcionamento dos mecanismos de autenticação, foram enviados pedidos HTTP para a API REST com credenciais incorretas ou em falta, e verificou-se que resultavam no código de erro HTTP 401 (*Unauthorized*), que é o comportamento esperado.

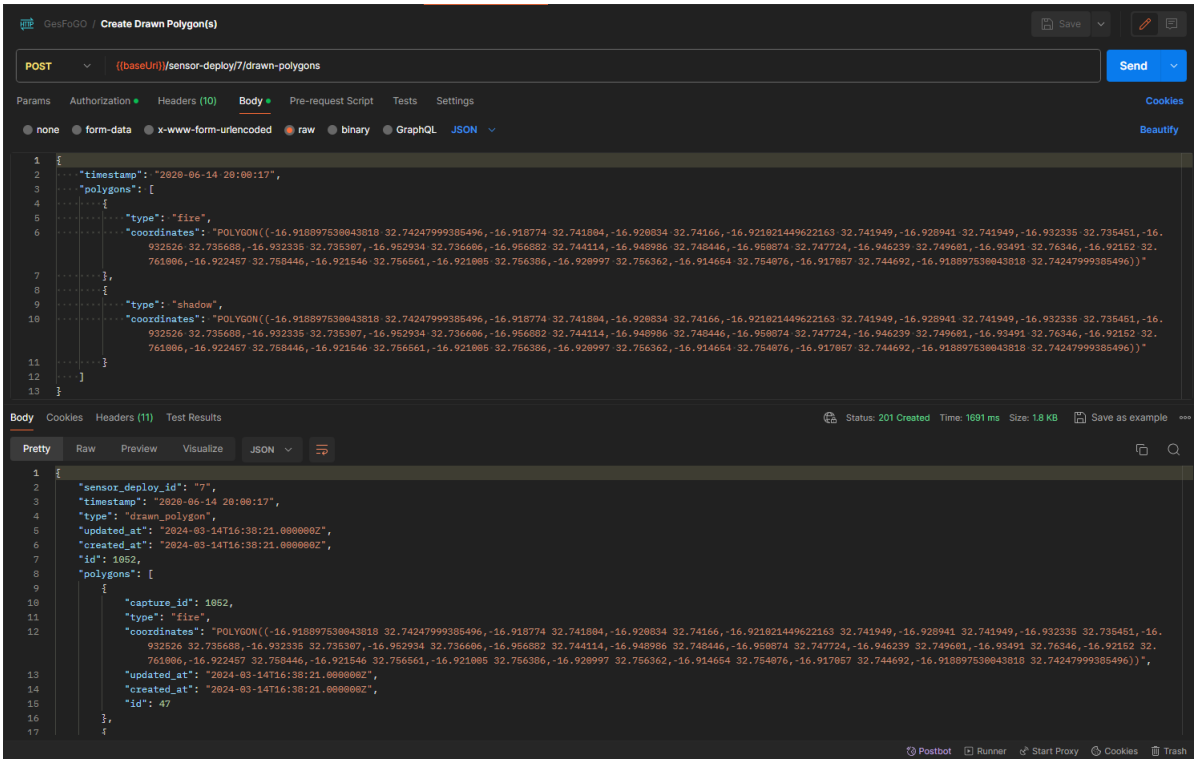


Figura 24. Captura de ecrã de um pedido *POST* no *Postman*

Por fim, para além de utilizar o *Postman* para todos estes testes, também é importante testá-los na GUI, isto é, usar as várias funcionalidades da GUI que envolvem pedidos à API para verificar se todas as rotas estavam a funcionar conforme esperado.

3.6 Documentação da API REST e dos Repositórios

Nesta secção, é apresentada uma visão geral acerca da documentação que foi feita para o projeto *GesFoGO*, não só a documentação da API, mas também a documentação acrescentada aos repositórios no ficheiro *README.md*, que fornece as instruções de como executar a aplicação e algumas outras informações relevantes para quem está a desenvolver as aplicações.

O projeto *GesFoGO* possui 2 repositórios: uma para a API e um para a GUI. Nas Figuras 25 e 26, são mostradas capturas de ecrã dos ficheiros *README* dos repositórios da API e da GUI, respetivamente. No *README* do repositório da GUI, Figura 26, consta apenas uma simples explicação de como fazer a preparação e execução do projeto, seja em ambiente de desenvolvimento ou de produção. Já no *README* do repositório da API, Figura 25, é necessária alguma informação acrescida, visto que o processo de preparar e executar a aplicação requer a cópia de alguns ficheiros e alteração de algumas configurações, que podem variar conforme está em ambiente de desenvolvimento ou de produção. Além disso, apresenta as instruções para gerar a documentação e também os comandos para gestão das *API Keys*, que não é mostrado na figura porque já estão descritos na Subsecção 3.3.2.

Quanto à documentação da API, isto é, dos seus *endpoints*, são fornecidas todas as informações essenciais sobre como integrar e interagir com a API. Inclui informações sobre os mecanismos de

README

gesfogo-api

Repository for the development of the GesFoGO REST API.

How to Execute the REST API

- git clone https://github.com/valentimc/gesfogo-api.git
- `$ composer install`
- Add the `.env` file to the project folder and change any information that is necessary
- `$ copy .\logo.png .\vendor\knuckleswtf\pastel\resources\images\logo.png`
- `$ php artisan scribe:generate` (Database must be online for the command to succeed)
- Execute this command, if the base URI for the API is '/gesfogo-api/' instead of '/api/': `$ sudo sed -i 's/api\x2f/gesfogo-api\x2f/g' ./public/docs/index.html`
- Update CORS `allowed_origins` configuration in `.\config\cors.php` file with the domain of the GUI App
- `$ php artisan serve --port=<PORT>` or use XAMPP

The `.env` file should **NOT** be tracked by Git.

Generate/Update API Documentation

To generate and/or update the API Documentation after changing it in the source code, execute the following command:

```
$ php artisan scribe:generate
```

In order to update the documentation according to the API as it is currently deployed in the EELab Server with different URIs, execute the following command that will replace the '/api/' with '/gesfogo-api/':

```
$ sudo sed -i 's/api\x2f/gesfogo-api\x2f/g' ./public/docs/index.html
```

To view the result of the generated Documentation open the browser and go to `/docs` URI.

Figura 25. *README* do repositório da API

README

gesfogo-gui

Repository for the development of the GesFoGO Graphical User Interface.

How to Setup the GUI Application

- git clone https://github.com/valentimc/gesfogo-gui.git
- `$ npm install`

1. How to Execute the GUI Application in Development Mode

- `$ npm start`
- Access `http://localhost:8080/` via the browser

2. How to Execute the GUI Application in Production Mode

- `$ npm run build` - this command will generate a bundle, inside a folder named 'dist', in the project's root directory, with every file needed: HTML, CSS, JS and images.
- After that the app can be served as a static HTML file which subsequently loads the other resources. It can be served through Apache, NGINX, among others.

Figura 26. *README* do repositório da GUI

autenticação, práticas recomendadas, parâmetros de cada pedido, formatos das respostas e exemplo de cada pedido.

Entre os módulos disponíveis para gerar a documentação da API, foi escolhido o ***Scribe***[23]. Uma outra opção, bastante popular, seria o *SwaggerUI*[24], mas a sua configuração é mais complexa e preferimos o formato da documentação gerada pelo *Scribe*. Em termos de funcionalidades são bastante equivalentes, mas o *Scribe* é desenvolvido especificamente para o *Laravel* e tem uma integração e configuração mais fácil e a sua documentação é bastante completa. Quanto à interface/documentação gerada, é interativa, fácil de navegar e compreender.

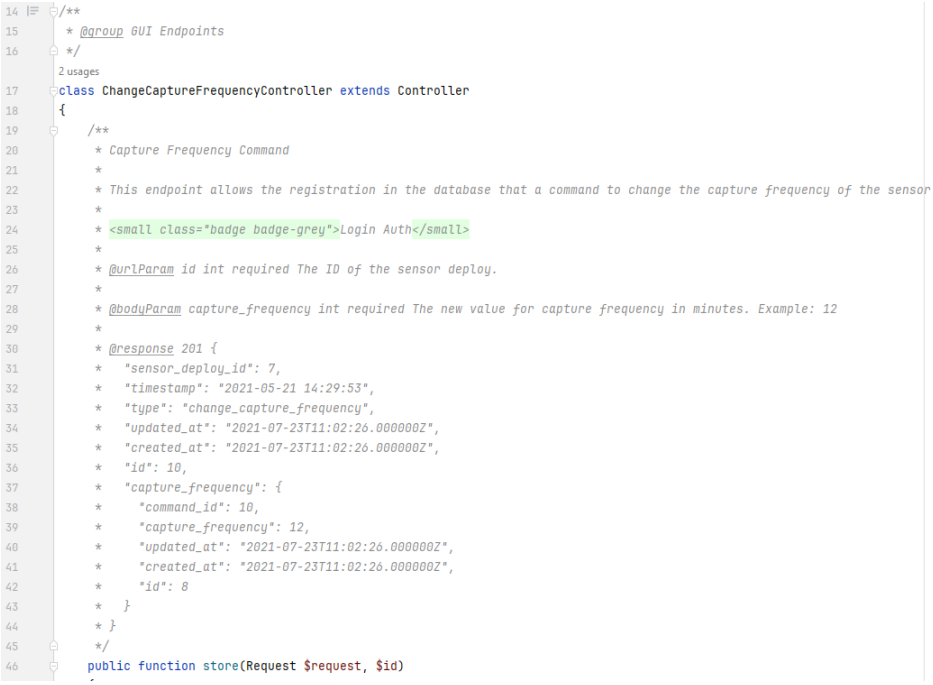
Os passos a seguir para a instalação são os seguintes [29]:

- Executar o comando '*composer require –dev knuckleswtf/scribe*' para instalar o módulo.
- Executar o comando '*php artisan vendor:publish –tag=scribe-config*' que cria o ficheiro *scribe.php* na diretoria *./config/*.
- No ficheiro de configuração do *Scribe* (*./config/scribe.php*), escolher o tipo de *setup* pretendido: *static* (gera um ficheiro HTML e os CSS e JS necessários na pasta *./public/docs/*) ou *laravel* (gera uma *view* e permite adicionar autenticação e/middlewares). Neste caso foi escolhido o tipo *static*.
- Ainda no ficheiro de configuração (*./config/scribe.php*), escrever o título da página, uma breve descrição da API, determinar para que rotas o *Scribe* deve criar documentação, escolher uma imagem para o *favicon*, entre outras.
- Para gerar a documentação, é usado o comando '*php artisan scribe:generate*'. Este comando deve ser executado sempre que são realizadas alterações à documentação.

Após feita a instalação e configuração inicial, é necessário adicionar a informação relativa a cada *endpoint* que deve constar na documentação gerada. Essa informação deve ser colocada antes da declaração da classe ou do método correspondente em forma de comentário. Na Figura 27 é mostrada uma captura de ecrã com o exemplo dessa informação para a rota *Capture Frequency Command*.

A configuração da informação necessária para cada *endpoint* é feita sobretudo através de anotações. Como é mostrado na Figura 27, nas linhas 14 a 16 é usada a anotação *@group*, que serve para adicionar todos os *endpoints* geridos por este *controller* a um determinado grupo, neste caso '*GUI Endpoints*', que ajuda bastante na organização e navegação na documentação que vai ser gerada. Esta anotação também pode ser usada individualmente em cada *endpoint* quando um *controller* possui *endpoints* de mais do que um grupo. Relativamente à informação de cada rota, começa pelo título/nome do *endpoint* (linha 20), seguida de uma breve descrição do que esse *endpoint* faz ou sobre o seu comportamento (linha 22). Em seguida, é adicionado um *badge* (linha 24) usando HTML, que indica o tipo de autenticação dessa rota. Depois, são declarados os parâmetros do URL através da anotação *@urlParam* (linha 26), que inclui o nome, o tipo, se é obrigatório (*required*), uma descrição e eventualmente um valor de exemplo para esse parâmetro. Os parâmetros do body (linha 28) com a anotação *@bodyParam* também seguem o mesmo formato. Por fim, é adicionado um exemplo de resposta de sucesso com o respetivo formato (linhas 30 a 44), através da anotação *@response*, seguida do código de *status* HTTP para sucesso, neste caso é 201 porque é criado um

[23]Scribe - `https://scribe.knuckles.wtf/laravel/`
[24]SwaggerUI - `https://swagger.io/tools/swagger-ui/`



```
/**
 * @group GUI Endpoints
 */
2 usages
class ChangeCaptureFrequencyController extends Controller
{
    /**
     * Capture Frequency Command
     *
     * This endpoint allows the registration in the database that a command to change the capture frequency of the sensor depl
     *
     * <small class="badge badge-grey">Login Auth</small>
     *
     * @urlParam id int required The ID of the sensor deploy.
     *
     * @bodyParam capture_frequency int required The new value for capture frequency in minutes. Example: 12
     *
     * @response 201 {
     *   "sensor_deploy_id": 7,
     *   "timestamp": "2021-05-21 14:29:53",
     *   "type": "change_capture_frequency",
     *   "updated_at": "2021-07-23T11:02:26.000000Z",
     *   "created_at": "2021-07-23T11:02:26.000000Z",
     *   "id": 10,
     *   "capture_frequency": {
     *     "command_id": 10,
     *     "capture_frequency": 12,
     *     "updated_at": "2021-07-23T11:02:26.000000Z",
     *     "created_at": "2021-07-23T11:02:26.000000Z",
     *     "id": 8
     *   }
     * }
     */
    public function store(Request $request, $id)
```

Figura 27. Captura de ecrã com a informação para documentação da rota *Capture Frequency Command*

novo recurso. Nas rotas em que não é requerida autenticação, como em *Login User* e *Register User* é acrescentada a anotação '*@unauthenticated*' para que não seja apresentado um *badge* com o texto '*requires authentication*' na documentação gerada. As restantes informações são deduzidas automaticamente.

Desde a fase inicial do projeto que a documentação foi introduzida, portanto esta foi desenvolvida e melhorada de forma contínua, sempre que era adicionado um novo *endpoint*, a respetiva documentação era criada simultaneamente, e também era atualizada ao realizar alterações/correções ao código. Uma vez feita a essa configuração e executando o comando '*php artisan scribe:generate*', podemos aceder à documentação gerada através do *browser*. O resultado é mostrado na captura de ecrã da Figura 28.

Como é possível ver na Figura 28, no lado esquerdo da página está um menu interativo que permite navegar pela documentação. A documentação gerada consiste numa única página. Clicando em cada opção, a parte principal da página é transportada para a respetiva localização. Logo no início, está uma pequena introdução acerca da API REST, em seguida, ao fazer *scroll* para baixo, encontramos a secção '*Authenticating requests*', mostrada na Figura 29, onde são explicadas as duas formas de se autenticar na API, que correspondem à informação que está na Secção 3.3. Estas partes da documentação também podem ser acedidas clicando no respetivo botão no menu do lado esquerdo da página.

No caso dos grupos de *endpoints* (*Auth Endpoints*, *GUI Endpoints* e *Info Reception Endpoints*), quando clicamos num deles, aparece a lista dos seus *endpoints* em que podemos clicar no *endpoint* pretendido e ser direcionados para essa parte da página. O comportamento dos grupos de *endpoints* pode ser observado através das Figuras 28 e 29. Na parte central da página está a informação do *endpoint*: título, descrição, tipo de autenticação, método HTTP, URI e parâmetros. No lado direito é mostrado um exemplo de pedido para aquele *endpoint* e também uma resposta de exemplo.

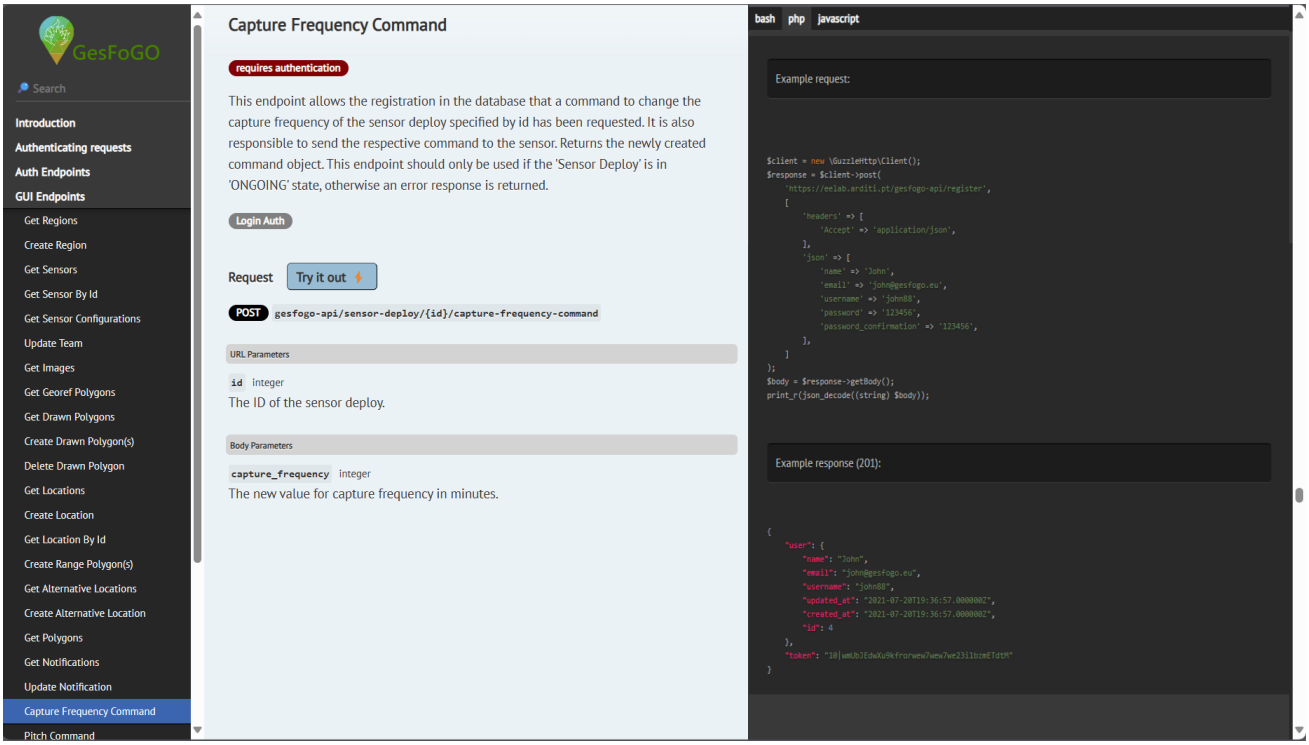


Figura 28. Captura de ecrã com a documentação gerada para *Capture Frequency Command*

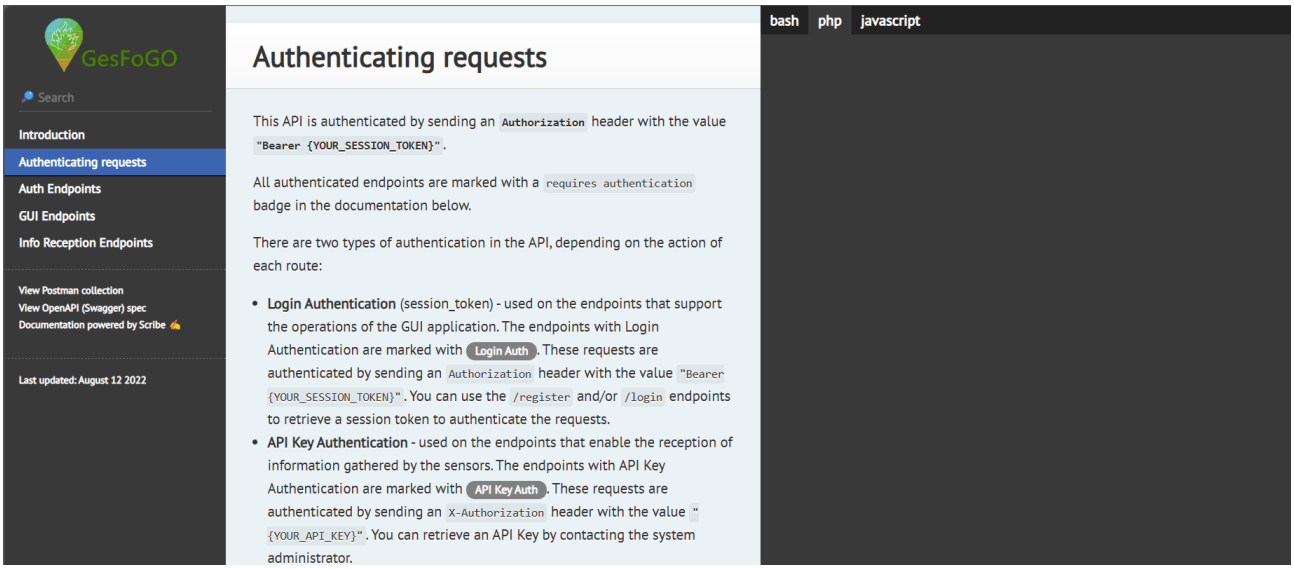


Figura 29. Captura de ecrã com a secção '*Authenticating requests*' da documentação

Fica assim concluída a parte do relatório acerca de como foi desenvolvida a API REST. Nas secções que se seguem, vai ser explicada a parte do desenvolvimento da GUI.

3.7 Estrutura do Código da GUI

A partir desta secção é explorado o desenvolvimento da Interface Gráfica do Utilizador (GUI) para o projeto *GesFoGO*, começando pela forma como o código da aplicação está estruturado. Como já foi mencionado na Subsecção 2.5.3, foi escolhido utilizar a tecnologia ***Node.js*** para desenvolver o *frontend* do *GesFoGO*, principalmente por permitir aproveitar as funções já implementadas e alterando apenas algum do código estrutural.

Ao contrário do código da API, em *Laravel*, não é necessário seguir uma estrutura específica para as diretorias nem para o código em si. Posto isto, a estrutura desenvolvida é descrita em seguida, explicando o propósito de cada diretoria e/ou ficheiro para o projeto em questão.

- **./dist/** - pasta gerada pelo *bundler Webpack* que contém uma versão otimizada do código-fonte da aplicação para distribuição, que vai ser servido de forma estática. Na Secção 3.8 é explicada a escolha e utilização do *bundler Webpack*.
- **./external/** - esta diretoria contém ficheiros externos, *javascript*, CSS e algumas imagens que fazem parte de alguns módulos instalados e precisam de ser carregados diretamente através do ficheiro *./src/index.html*.
- **./node_modules/** - nesta pasta ficam armazenadas as dependências do projeto, instaladas através do gestor de módulos do *Node.js*, o NPM.
- **./src/** - esta diretoria contém o código-fonte que está a ser desenvolvido para a aplicação, mais concretamente:
  - **./src/assets/** - diretoria que contém as imagens necessárias para a GUI, como imagens representativas dos sensores e o *favicon*.
  - **./src/css/** - esta pasta contém um ficheiro chamado *style.css*, que define o estilo de alguns dos elementos HTML que constituem a GUI do GesFoGO, como dos visores, tabelas, *timeline*, entre outros.
  - **./src/js/** - diretoria onde ficam armazenados os ficheiros principais com o código-fonte *javascript* que foi desenvolvido:
    - **./src/js/api-acess.js** - neste ficheiro estão implementadas as funções para fazer acessos à API para qualquer tipo de pedido.
    - **./src/js/api-operations.js** - neste ficheiro estão definidas as funções que se vão traduzir em chamadas à API, ou seja, fazem a preparação da informação necessária, e depois utilizam as funções de *api-acess.js* para realizar o pedido HTTP à API.
    - **./src/js/callbacks.js** - neste ficheiro estão implementadas as funções de *callback*, que são chamadas assim que as respostas aos pedidos à API chegam e processam essas respostas devidamente.
    - **./src/js/functions.js** - neste ficheiro estão implementadas variadas funções auxiliares que não se encaixam em nenhuma categoria específica.

    - **./src/js/index.js** - é o ponto de entrada da aplicação, responsável por importar alguns ficheiros CSS, inicializar e configurar alguns elementos visuais, como o mapa, e as várias camadas de informação, atribuir alguns *listeners* a elementos da página e despoletar o carregamento dos sensores no mapa.
    - **./src/js/listeners.js** - neste ficheiro estão definidas as funções que são definidas como *listeners* dos elementos HTML.
    - **./src/js/variables.js** - ficheiro onde constam todas as variáveis globais da aplicação.
  - **./src/index.html** - ficheiro HTML da página inicial (e única) da GUI do *GesFoGO*, onde são definidos os elementos HTML principais que constituem a página.
- **./config.json** - ficheiro de configuração da aplicação, onde podem ser definidos valores para algumas variáveis, como o URL base da API, as coordenadas centrais do mapa, o nível de zoom da mapa, entre outras.
- **./package.json** - ficheiro que contém metadados sobre o projeto e sobre as suas dependências, como nome do projeto, a versão, a descrição, os *scripts* e dependências.
- **./package-lock.json** - ficheiro gerado automaticamente pelo NPM quando é feita a instalação das dependências do projeto, onde ficam registadas as versões exatas de todas as dependências.
- **./README.md** - ficheiro informativo com algumas instruções úteis para quem está a desenvolver o projeto, nomeadamente como fazer a configuração inicial e como executar o programa.
- **./webpack.common.js** - ficheiro de configuração do *Webpack* com as configurações comuns, que são usadas tanto no modo de desenvolvimento (*dev*) como no modo produção (*prod*).
- **./webpack.dev.js** - ficheiro de configuração do *Webpack* com as configurações específicas para o modo de desenvolvimento (*dev*), que utiliza como base a configuração em *./webpack.common.js*.
- **./webpack.prod.js** - ficheiro de configuração do *Webpack* com as configurações específicas para o modo de produção (*prod*), que utiliza como base a configuração em *./webpack.common.js*.

Através desta estrutura já é possível ter uma ideia de como está organizada a aplicação da GUI. As explicações mais detalhadas acerca da implementação de cada funcionalidade, dos módulos utilizados e também acerca do *bundler Webpack* são apresentadas nas secções que se seguem.

3.8 *Bundler* para a Aplicação Web

Para a realização da componente web (cliente) foi decidido utilizar um *bundler*, que é uma ferramenta que serve para compilar módulos *Javascript* (neste caso), e a partir dos vários módulos/ficheiros implementados gera um único ficheiro com todo o código *Javascript* necessário e um ficheiro HTML correspondente à página inicial (e única, neste caso), e alguns ficheiros CSS e imagens. Estes ficheiros serão posteriormente servidos ao *browser* na primeira vez que a aplicação é carregada e contêm toda a aplicação. De resto, apenas são realizados pedidos HTTP à Web API para acesso aos dados provenientes da base de dados.

Entre os *bundlers* existentes foram escolhidos 3 para serem analisados e comparados de forma objetiva a fim de escolher o mais indicado para este caso específico. Os *bundlers* analisados foram:

- ***Parcel***[25] - é um *bundler* de aplicações web, que se diferencia pela experiência que oferece ao programador, com um desempenho extremamente rápido, utilizando processamento *multicore* e não requer configuração.
- ***Webpack***[26] - é um *bundler* de módulos estáticos para aplicações *Javascript* modernas. Constrói internamente um gráfico de dependências que mapeia todos os módulos que o projeto em questão precisa e gera um ou mais *bundles*.
- ***Browserify***[27] - é um dos primeiros *bundlers* que foi desenvolvido para permitir carregar módulos CommonJS Node nos *browsers*. É de pequena dimensão e pode ser estendido com *plugins*.

3.8.1 Tabela comparativa

Para uma comparação mais fácil foi elaborada a Tabela 1 que se segue, que classifica cada *bundler* relativamente a várias caraterísticas numa escala de "Muito Mau"a "Muito Bom".

Tabela 1. Comparativo dos diferentes *bundlers*

	Parcel	**Webpack**	**Browserify**
Facilidade de configuração	Muito Bom	Suficiente	Mau
Tempo de *build*	Muito Mau	Muito Bom	Suficiente
Liberdade/possibilidade de customização	Muito Mau	Suficiente	Muito Bom
Dimensão do pacote gerado	Suficiente	Bom	Insuficiente
Documentação	Bom	Muito Bom	Suficiente

Muito Mau - Mau - Insuficiente - Suficiente - Bom - Muito Bom

Nota: Os valores de tempo e dimensão utilizados para esta comparação foram obtidos no artigo [30].

3.8.2 Critérios de Escolha

No processo de passar a aplicação de *Electron* para web, o mais importante é que o *bundler* escolhido seja de fácil utilização/configuração, que permita um mínimo de possibilidade de configuração e gere o mínimo de erros possível perante o código já existente.

3.8.3 O *Bundler* Escolhido

Com base na Tabela 1 e nos critérios de escolha, foi excluído o *bundler Browserify*. Os *bundlers Parcel* e *Webpack* foram testados com a aplicação *Node.js* já existente e como já era expectável o *Parcel* funcionou bem logo no início e sem qualquer necessidade de configuração, já o *Webpack* necessitou de algumas configurações iniciais para começar a funcionar e ainda surgiram alguns erros, mas que foram facilmente resolvidos. Com base no que foi referido a escolha mais indicada seria o *Parcel*, no entanto, após alguma utilização este passou a gerar vários erros e como não permite muita configuração seria necessário fazer muitas alterações ao código já existente. Posto isto, o *bundler* escolhido acabou por ser o ***Webpack*** que é mais flexível e ainda acaba por ser mais eficiente, visto que gera um pacote de menor dimensão em comparação com o *Parcel*.

[25] Parcel - `https://parceljs.org/`
[26] Webpack - `https://webpack.js.org/`
[27] Browserify - `https://browserify.org/`

3.8.4 Configuração do *Bundler Webpack*

Para a configuração do *Webpack*, foram definidos ficheiros de configuração distintos para o modo de desenvolvimento e para o modo de produção, isto porque os objetivos em cada modo são diferentes. No modo de desenvolvimento, queremos tanta informação quanto possível e também um servidor em *localhost* que recarrega automaticamente sempre que são feitas alterações ao código. No modo de produção, o objetivo é que a dimensão dos pacotes gerados seja o mais reduzida possível e otimizada para diminuir os tempos de carregamento.

Os módulos utilizados são disponibilizados através do *Node Package Manager* (NPM), o gestor de módulos para projetos em *Node.js*. Após instalar o módulo principal do *webpack*[28], também foram instalados alguns módulos acessórios: o *webpack-cli*[29] para ter acesso aos comandos CLI e o *webpack-dev-server*[30] que fornece um servidor local que recarrega automaticamente sempre que são feitas alterações ao código. Além destes, o *Webpack* requer uma série de *plugins* e *loaders*, selecionados conforme as necessidades e tipos de ficheiros usados em cada aplicação. Neste caso, os *plugins* instalados foram: o *html-webpack-plugin*[31] para auxilio na criação dos ficheiros HTML para o *bundle*, o *copy-webpack-plugin*[32] que permite a cópia de ficheiros ou diretorias já existentes para a pasta *./dist/*, o *mini-css-extract-plugin*[33] para extrair o CSS em ficheiros separados e o *clean-webpack-plugin*[34] que faz a limpeza da pasta *./dist/* antes de gerar os novos ficheiros para remover ficheiros antigos que podem já não ser necessários. Os *loaders* necessários, que servem para fazer o pré-processamento de ficheiros são: o *file-loader*[35] para resolver/interpretar palavras-chave como '*import*' e '*require*', e ainda o *style-loader*[36] e o *css-loader*[37], que são usados em conjunto para processar os ficheiros CSS.

Em relação ao *javascript*, é necessário utilizar o *Babel*[38], que é um compilador de *javascript* usado para converter código *ECMAScript* [31] de 2015 em diante em código compatível com o ambiente de execução de *javascript* dos navegadores web. A sua instalação requer 3 módulos: o *@babel/core*[39], o *@babel/preset-env*[40] e o *babel-loader*[41] que permite transpilar os ficheiros *javascript*.

De forma a ter os dois modos, desenvolvimento e produção, foram seguidas as indicações descritas na documentação oficial do *Webpack* [32]. Portanto, em vez de ter apenas um ficheiro de configuração para o *Webpack*, temos 3 ficheiros distintos: o *webpack.common.js* com as configurações comuns para ambos os modos, o *webpack.dev.js* com as configurações específicas para o modo de desenvolvimento (*dev*) e o *webpack.prod.js* com as configurações específicas para o modo de produção (*prod*). Para utilizar esta abordagem é necessário instalar mais um módulo, o *webpack-merge*[42]

[28] webpack - `https://www.npmjs.com/package/webpack`
[29] webpack-cli - `https://www.npmjs.com/package/webpack-cli`
[30] webpack-dev-server - `https://www.npmjs.com/package/webpack-dev-server`
[31] html-webpack-plugin - `https://www.npmjs.com/package/html-webpack-plugin`
[32] copy-webpack-plugin - `https://www.npmjs.com/package/copy-webpack-plugin`
[33] mini-css-extract-plugin - `https://www.npmjs.com/package/mini-css-extract-plugin`
[34] clean-webpack-plugin - `https://www.npmjs.com/package/clean-webpack-plugin`
[35] file-loader - `https://www.npmjs.com/package/file-loader`
[36] style-loader - `https://www.npmjs.com/package/style-loader`
[37] css-loader - `https://www.npmjs.com/package/css-loader`
[38] Babel - `https://babeljs.io/`
[39] @babel/core - `https://www.npmjs.com/package/@babel/core`
[40] @babel/preset-env - `https://www.npmjs.com/package/@babel/preset-env`
[41] babel-loader - `https://www.npmjs.com/package/babel-loader`
[42] webpack-merge - `https://www.npmjs.com/package/webpack-merge`

que serve para agregar os ficheiros de configuração '*common*' com os ficheiros de configuração '*dev*' e '*prod*'.

No ficheiro de configuração comum (*webpack.common.js*) é definido o ficheiro que serve como ponto de entrada da aplicação. Em seguida, são instanciados os plugins necessários, nomeadamente o *HtmlWebpackPlugin* (*html-webpack-plugin*) e os *plugins* internos do *Webpack*: o *ProvidePlugin* que carrega módulos automaticamente para não ter de importá-los sempre que são necessários (usado para o módulo *jQuery*) e o *SourceMapDevToolPlugin* que permite um controlo mais pormenorizado da geração de *source-maps* para obter mais informações acerca do que vai acontecendo, útil para *debug*. Depois são definidas as regras para cada tipo de ficheiro e os *loaders* usados para cada tipo: para os ficheiros *javascript* é usado o *babel-loader* e para imagens é usado o *file-loader*.

Quanto à configuração específica para o modo desenvolvimento (ficheiro *webpack.dev.js*), é atribuído o valor '*development*' à propriedade '*mode*', as diretorias ./external e ./src/assets são configuradas para ser servidas estaticamente e são adicionadas as regras para os ficheiros CSS que usam o '*style-loader*' e o '*css-loader*'.

Na configuração específica para o modo produção (ficheiro *webpack.prod.js*), é atribuído o valor '*production*' à propriedade '*mode*', depois é escolhido o nome do ficheiro que vai ser gerado pelo *bundler* e a sua localização. Quanto aos *plugins*, são criadas instâncias do *MiniCssExtractPlugin* (*mini-css-extract-plugin*), do *CleanWebpackPlugin* (*clean-webpack-plugin*) e do *CopyWebpackPlugin* (*copy-webpack-plugin*) para a cópia dos ficheiros estáticos de imagens e código externo. Por fim, é acrescentada a regra para os ficheiros CSS que vão usar o *loader* do *MiniCssExtractPlugin* e o '*css-loader*'.

Após criadas estas configurações, a sua utilização é feita através dos *scripts* que são definidos no ficheiro *package.json*. Foram definidos 2 *scripts*:

- ***start*** - "*webpack-dev-server –config webpack.dev.js*"
- ***build*** - "*webpack –config webpack.prod.js*"

Desta forma, executando o comando "*npm start*"é inicializado o servidor de desenvolvimento do *webpack*, usando a configuração *webpack.dev.js*. Executando o comando "*npm run build*"é feito o processo de gerar o *bundle* completo dentro da pasta *./dist/*, usando a configuração *webpack.prod.js*.

3.9 Módulos Utilizados para a GUI

Para além dos módulos que já foram mencionados na Subsecção 3.8.4, relativa à instalação e configuração do *bundler Webpack*, foram utilizados alguns módulos adicionais para permitir a implementação das funcionalidades pretendidas para este projeto:

- ***jquery***[43] - uma implementação específica para *Node.js* da biblioteca *jQuery* [33] para *javascript* com funcionalidades que ajudam na manipulação e gestão de ficheiros HTML, animações, eventos e *Ajax*, com uma API compatível com diversos browsers.
- ***bootstrap***[44] - uma *framework* que possui um conjunto de componentes e ferramentas *HTML*, *CSS* e *Javascript* pré-concebidos e personalizáveis, como barras de navegação, formulários, botões e muito mais, que potenciam a interface gráfica do utilizador, tornando-a mais consistente e apelativa.

[43] jquery - `https://www.npmjs.com/package/jquery`
[44] bootstrap - `https://www.npmjs.com/package/bootstrap`

- ***leaflet***[45] - é o módulo mais importante desta aplicação pois é através dele que foram desenvolvidas as funcionalidades principais da aplicação do *GesFoGO*. Consiste numa biblioteca *javascript open-source* para integração de mapas interativos, cujas suas capacidades podem ser estendidas através de variados *plugins* organizados por categorias e possui uma documentação bastante completa.
- ***leaflet-freehandshapes***[46] - um *plugin* para o *leaflet* para adicionar/manipular polígonos, desenhando-os de forma livre num mapa do *leaflet*.
- ***leaflet-semicircle***[47] - um *plugin* para o *leaflet* para adicionar semicírculos ao mapa, que podem ter uma extensão variável (em graus).

A escolha do módulo *Leaflet* já tinha sido feita pelo colega que iniciou o desenvolvimento do projeto *GesFoGO*. No seu relatório [34], é apresentada a análise e teste a alguns módulos alternativos que possuem as caraterísticas e funcionalidades necessárias para o desenvolvimento deste projeto. Com base nos critérios de escolha e nos testes ao funcionamento de cada módulo, acabou por ser escolhido o *Leaflet*.

Resumidamente, foram explorados 3 módulos que se adequavam às necessidades deste projeto: o *CesiumJS*[48], o *OpenLayers*[49] e o *Leaflet*. A utilização do módulo *CesiumJS* é intuitiva e possui uma versão grátis e uma versão comercial, após uma implementação inicial chegou-se à conclusão de que não seria possível elaborar/testar todas as funcionalidades pretendidas apenas com a versão grátis. Quanto ao módulo *OpenLayers*, apresenta uma complexidade acrescida no desenvolvimento devido a ser bastante completo, uma vez que possui inúmeros conceitos, interações, funcionalidades, camadas, vetores, etc. Além disso, possui uma documentação bastante completa e pouca variedade de *plugins* compatíveis. Quanto ao módulo *Leaflet*, agrega as caraterísticas positivas dos módulos *CesiumJS* e do *OpenLayers*, ou seja, a sua utilização é intuitiva, a documentação é bastante completa, é facilmente integrado com *plugins* e suporta as funcionalidades pretendidas para o *GesFoGO*, daí ter sido o escolhido [34].

As secções que se seguem vão ser dedicadas a demonstrar as várias funcionalidades da aplicação (GUI) do *GesFoGO*, explicando o seu comportamento e, quando relevante, como foram implementadas.

3.10 Interface Gráfica do Utilizador (GUI)

Primeiramente, é importante demonstrar, de uma forma mais abrangente, como é constituída a interface gráfica do utilizador (GUI) desenvolvida para o projeto *GesFoGO*, tendo por base os requisitos estabelecidos na fase inicial do projeto. Em seguida, na Figura 30, é apresentada uma captura de ecrã da GUI geral do *GesFoGO* já na sua iteração final.

Como é possível observar na Figura 30, a interface está dividida em três secções/visores: *MainVisor* no lado esquerdo, *NodeVisor* no canto superior direito e *ImageVisor* na canto inferior direito. Todos os visores têm a possibilidade de ser colocados em modo de ecrã inteiro (*fullscreen*), caso necessário. Cada visor tem os seus propósitos e funcionalidades, que são abordados nas 3 subsecções que se seguem, dedicadas a cada um dos visores.

[45] leaflet - `https://www.npmjs.com/package/leaflet`
[46] leaflet-freehandshapes - `https://www.npmjs.com/package/leaflet-freehandshapes`
[47] leaflet-semicircle - `https://www.npmjs.com/package/leaflet-semicircle`
[48] CesiumJS - `https://cesium.com/platform/cesiumjs/`
[49] OpenLayers - `https://openlayers.org/`

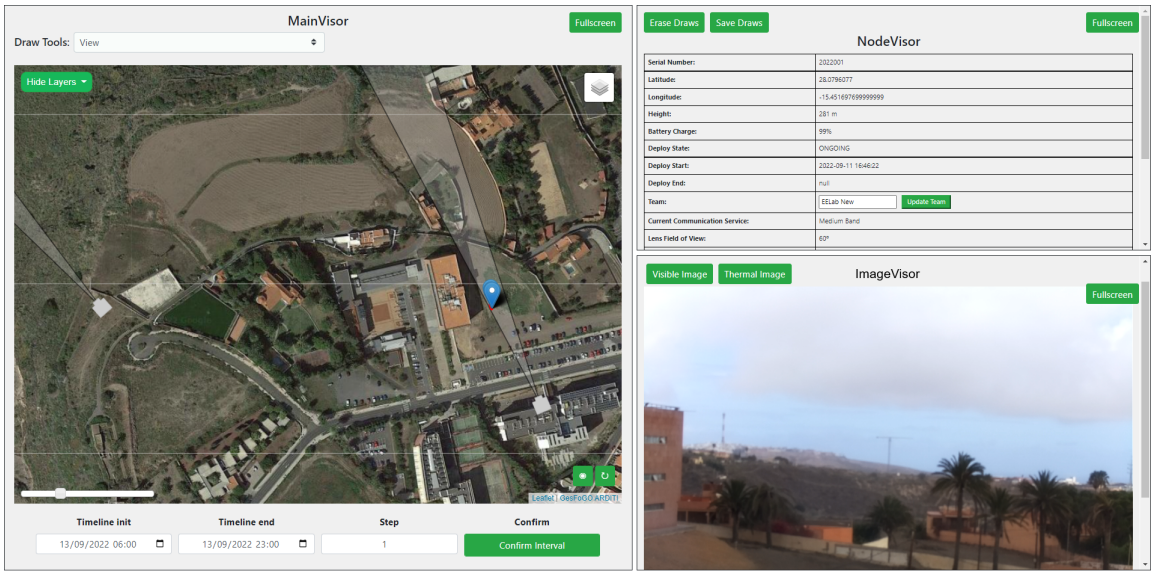


Figura 30. Captura de ecrã da GUI do *GesFoGO*

3.10.1 *MainVisor*

No lado esquerdo da página da GUI temos o *MainVisor*, que é a secção principal da aplicação, onde está localizado o mapa com os sensores, linhas de fogo e também a funcionalidade *timeline*. O *MainVisor* permite que os técnicos desenhem linhas de fogo e sombra conforme as circunstâncias no momento em que está a ocorrer um incêndio com base na sua experiência profissional. A *timeline* permite que os operadores analisem vários cenários conforme a evolução do incêndio para determinar a melhor forma de o combater. Na Figura 31, consta uma captura de ecrã do *MainVisor* onde também são mostrados os menus *drop-down* com as várias opções de configuração disponíveis.

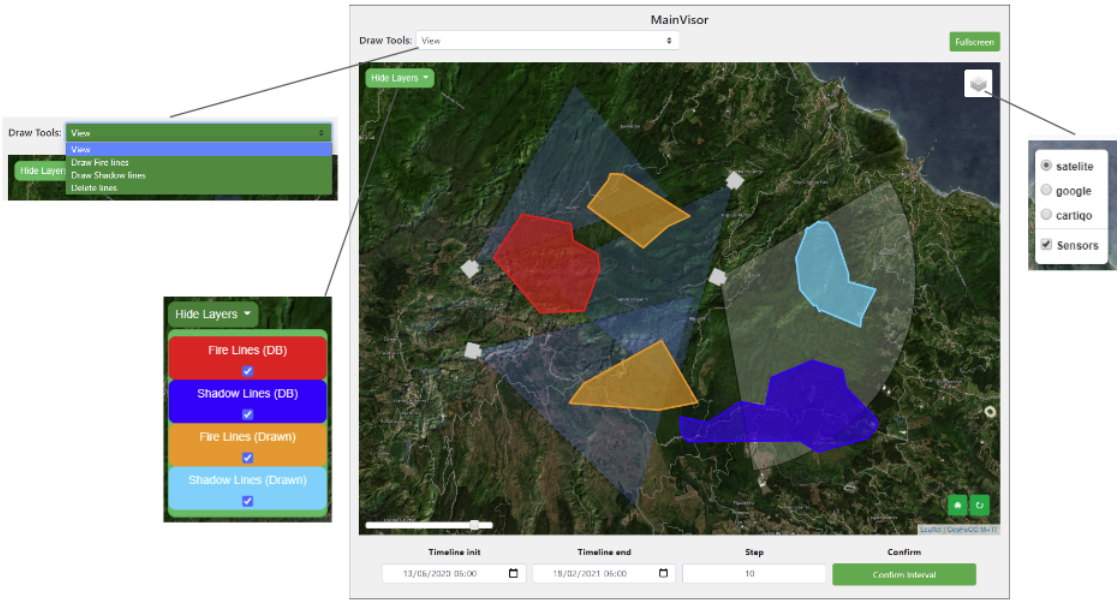


Figura 31. Captura de ecrã do *MainVisor* com menus *drop-down*

Primeiramente, é possível verificar que no menu '*Draw Tools*' (Ferramentas de Desenho), na Figura 31, temos as opções de operações que podemos realizar no mapa: visualizar (*View*), desenhar linhas de fogo (*Draw Fire lines*), desenhar linhas de sombra (*Draw Shadow lines*) e ainda, apagar as linhas que foram desenhadas por engano (*Delete lines*).

No canto superior esquerdo do mapa está o menu '*Hide Layers*' (Ocultar Camadas), que permite ao utilizador escolher que tipos de linhas de fogo pretende que apareçam no mapa, entre as seguintes opções: *Fire Lines (DB)*, *Shadow Lines (DB)*, *Fire Lines (Drawn)*, *Shadow Lines (Drawn)*. As camadas do tipo '*DB*' referem-se às linhas geradas pelo algoritmo de georreferenciamento e as camadas do tipo '*Drawn*' correspondem às linhas desenhadas por um operador através da GUI. Neste menu, cada opção possui a cor dos polígonos correspondentes no mapa.

No canto superior direito do mapa existe um seletor onde é possível escolher o tipo de mapa que queremos: '*satelite*', '*google*' ou '*cartigo*', conforme a preferência do utilizador. Também podemos escolher se queremos que os sensores sejam mostrados ou não através da caixa de seleção '*sensors*'.

No canto inferior direito do mapa existem 2 botões que, apesar de não fazerem parte dos requisitos, tomei a iniciativa de os acrescentar, por serem de simples implementação e melhorarem consideravelmente a usabilidade da aplicação. O botão da esquerda serve para recentrar o mapa, ou seja, voltar à posição inicial conforme o que está no ficheiro de configuração. O botão direito é utilizado para atualizar o mapa por iniciativa do operador, sem a necessidade de atualizar a página inteira.

Na Figura 32, em seguida, estão identificados os restantes elementos que constituem o *MainVisor*.

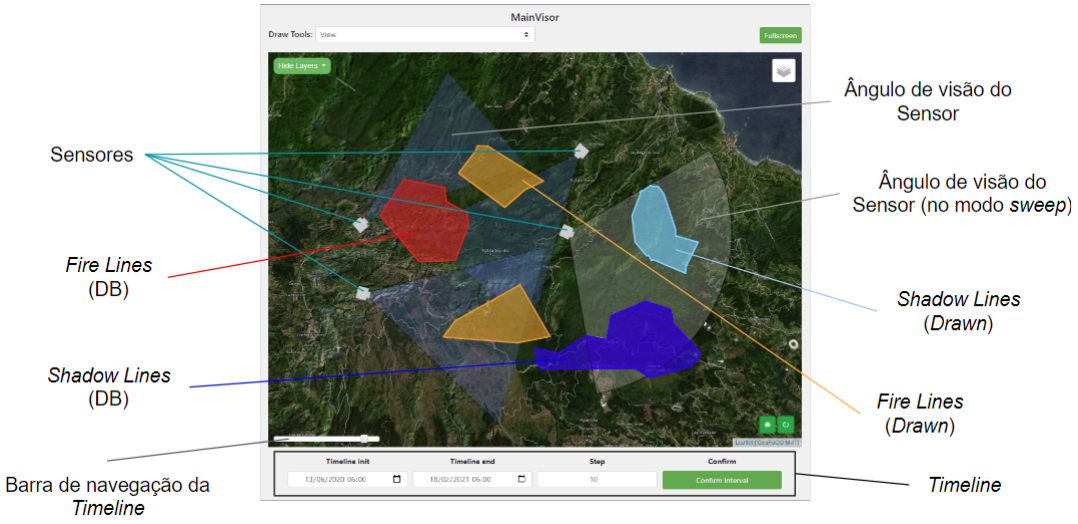


Figura 32. Captura de ecrã do *MainVisor* com identificação dos elementos

Em relação ao ângulo de visão dos sensores, existem dois tipos de representação dependendo se o sensor está em modo fixo ou em modo varrimento (*sweep*). No caso de estar em modo fixo, o campo de visão é representado por um triângulo com o ângulo de visão da lente do sensor, no caso de estar em modo de varrimento, é representado por um semicírculo com a amplitude total de movimento do sensor.

A Barra de navegação da *Timeline*, no canto inferior esquerdo do mapa, é onde o operador pode navegar entre instantes de tempo, de acordo com o intervalo de tempo escolhido na '*Timeline*'.

Na parte inferior do *MainVisor* está a '*Timeline*' (Linha do Tempo), que é a funcionalidade que permite ao operador escolher que amostras pretende que sejam mostradas no mapa. Para isso, o utilizador tem de escolher as datas de início e fim do intervalo de tempo que quer ver no mapa e o '*step*', que é o intervalo de tempo entre cada amostra (em minutos), que pode ter valores entre 1 e 1440 minutos; depois clica no botão '*Confirm Interval*' para confirmar e fazer a atualização das amostras que vão ser mostradas no mapa. A implementação e comportamento desta funcionalidade é explicada de forma mais aprofundada na Secção 3.11.

Inicialmente, era possível fazer a configuração da orientação (*yaw*) do sensor manualmente no mapa, através de duplo clique no respetivo sensor. Mas após alguma consideração decidimos remover a possibilidade de fazer esta configuração desta forma por ser pouco intuitiva, isto é, o utilizador pode nem saber que essa opção existe, e também porque tem menos precisão do que especificar o ângulo desejado.

Por fim, resta explicar como desenhar as linhas de fogo e/ou sombra no mapa. Antes de selecionar a ferramenta desejada em '*Draw Tools*', é necessário ter a respetiva camada selecionada para ser mostrada no mapa. Ou seja, para desenhar linhas de fogo temos de ativar a camada '*Fire Lines (Drawn)*' no menu '*Hide Layers*'. Para desenhar linhas de sombra temos de ativar a camada '*Shadow Lines (Drawn)*'. Na Figura 33, mostrada em seguida, encontra-se uma captura de ecrã exemplo para desenhar linhas de fogo.

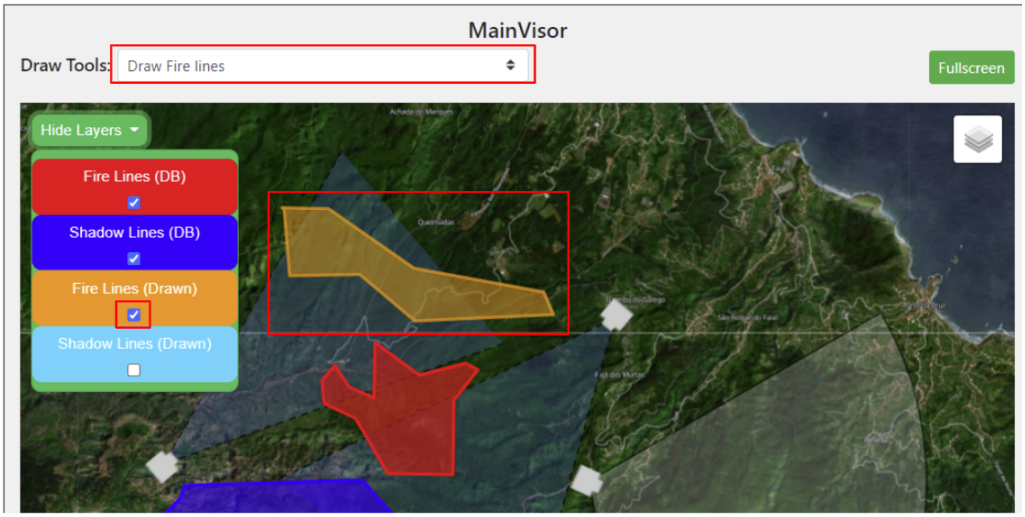


Figura 33. Captura de ecrã que ilustra a funcionalidade de desenhar no mapa

Na Figura 33 podemos observar que no menu '*Hide Layers*' foi selecionada a camada '*Fire Lines (Drawn)*', no menu '*Draw Tools*' foi escolhida a opção '*Draw Fire lines*' e o polígono laranja que foi desenhado à mão está destacado com um retângulo vermelho. Feito isto, é necessário clicar num dos sensores para definir a que sensor este polígono ficará associado, e no final, é necessário clicar no botão '*Save Draws*', localizado no *NodeVisor*.

Nota: as linhas de sombra correspondem a linhas de fogo cujos sensores não conseguem observar devido a fatores que condicionam a visibilidade, como por exemplo, montanhas ou vales.

3.10.2 *NodeVisor*

No canto superior direito da página temos o *NodeVisor*, onde é apresentada toda a informação relativa ao sensor e à respetiva instalação (*deploy*) atual, conforme selecionado pelo utilizador através de um clique no ícone do respetivo sensor no mapa. Além disso, permite o envio de comandos para esse mesmo sensor. Na Figura 34, é mostrada uma captura de ecrã do *NodeVisor*.

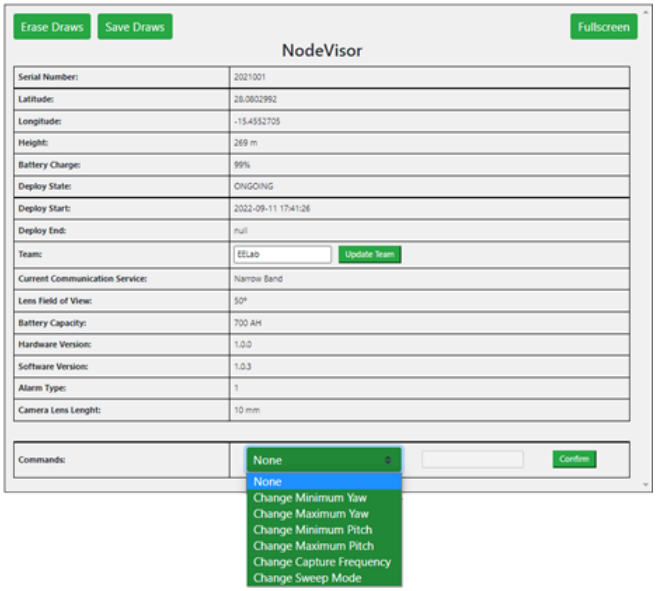


Figura 34. Captura de ecrã do *NodeVisor*

Como é possível ver na Figura 34, as informações apresentadas no *NodeVisor* são: o número de série do sensor, a latitude, longitude e altitude do local onde o sensor está instalado, a carga atual da bateria, as datas de início e fim do *deploy*, a equipa responsável por esse *deploy*, o tipo de comunicação, o ângulo de visão da lente do sensor, a capacidade da bateria, versões de hardware e software do sensor, tipo de alarme e o comprimento da lente.

Na parte inferior do *NodeVisor* existe uma tabela que permite que o operador envie comandos para o sensor que está selecionado. Na Figura 34, o menu *drop-down* com os comandos possíveis está aberto, mostrando todos os comandos disponíveis para aquele sensor. Dependendo do modo em que o sensor está a operar, fixo ou varrimento, o menu de seleção de comandos adapta-se. Se estiver no modo varrimento, permite alterar os valores mínimo e máximo de orientação e inclinação (como no exemplo da Figura 34), se estiver no modo fixo permite alterar apenas os valores de orientação e inclinação. Após selecionar um dos comandos disponíveis, o valor atual é mostrado na caixa de texto e pode ser editado pelo utilizador. Após escolher o novo valor para aquela configuração, basta clicar no botão '*Confirm*' para executar o comando.

No canto superior esquerdo do *NodeVisor* estão 2 botões: '*Erase Draws*' e '*Save Draws*' que servem para, respetivamente, apagar as linhas de fogo desenhadas que acabaram de ser feitas e para gravar na BD as linhas de fogo desenhadas.

Para concluir, o *NodeVisor* também permite alterar a equipa responsável pela instalação do sensor selecionado, através da edição da caixa de texto onde já está escrito o nome da atual equipa responsável e, posteriormente, clicando no botão '*Update Team*'.

3.10.3 *ImageVisor*

Por fim, no canto inferior direito da página está o *ImageVisor*, onde são mostradas as imagens (visíveis e térmicas) capturadas pelo sensor que está selecionado no momento (mostrado no *NodeVisor*) e para o instante de tempo que está selecionado na barra de navegação da *Timeline* (no canto inferior esquerdo do mapa). Na Figura 35, é mostrada uma captura de ecrã do *ImageVisor*.

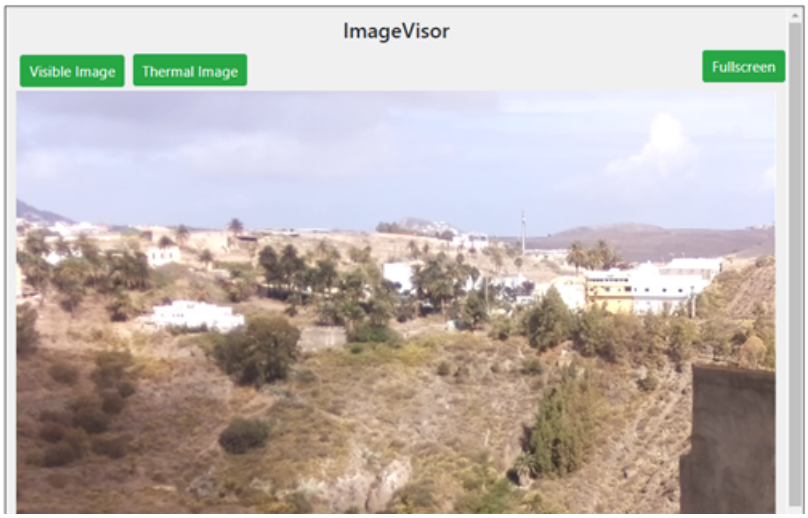


Figura 35. Captura de ecrã do *ImageVisor* com imagem visível

Como podemos observar na Figura 35, o *ImageVisor* possui dois botões no canto superior esquerdo: o botão '*Visible Image*' para mostrar a fotografia capturada pelo sensor naquele instante e o botão '*Thermal Image*' para mostrar a imagem térmica capturada pelo sensor naquele instante de tempo, se disponível. A imagem exibida por defeito é a imagem visível. Na Figura 36, em seguida, é mostrado o exemplo do *ImageVisor* com a imagem térmica correspondente à imagem visível da Figura 35.

Nas secções seguintes vão ser explicadas, de forma mais aprofundada, algumas das funcionalidades mais importantes e mais complexas da aplicação desenvolvida, isto é, como foram implementadas.

3.11 Implementação da *Timeline*

Nesta secção é descrito o processo de implementação da funcionalidade '*timeline*' que tem como objetivo permitir ao utilizador a escolha de um intervalo de tempo para o qual este pretende ver as linhas de fogo mostradas no mapa e também o espaço temporal entre cada amostra. Posteriormente, é possível navegar pelas várias amostras conforme as suas necessidades. Na Figura 37 a seguir, é mostrada a localização e aparência da *Timeline* na interface gráfica do *GesFoGO*.

Na parte de baixo da Figura 37 está o formulário onde são escolhidas as datas inicial (*Timeline init*) e final (*Timeline end*) do intervalo pretendido e o '*step*' que é um valor, em minutos, para

Figura 36. Captura de ecrã do *ImageVisor* com imagem térmica

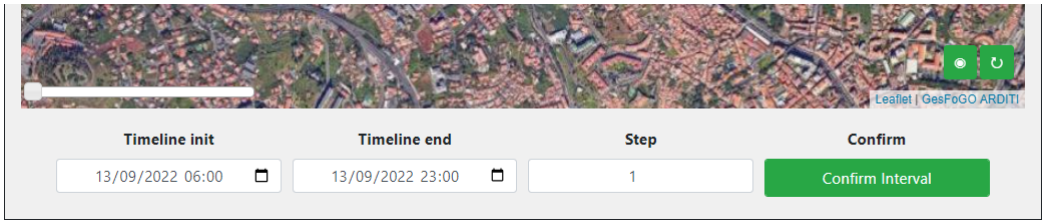


Figura 37. Funcionalidade *Timeline* na interface gráfica

indicar, em média, quantos minutos o utilizador quer entre amostras. No canto inferior esquerdo do mapa está a barra de navegação para navegar entre as amostras que foram escolhidas. Enquanto o utilizador está a navegar na *timeline* é mostrada a data e hora a que foram realizadas as capturas que estão a ser mostradas naquele momento.

Na subsecção que se segue é descrito como funciona o algoritmo que foi desenvolvido para dar suporte a esta funcionalidade. Optamos por acrescentar esta lógica à parte do *frontend*, porque é relativamente complexa e envolve várias operações que se tornam demoradas. Além disso, como esta funcionalidade serve sobretudo para visualizar dados passados, ou seja, que já não vão ser alterados, a possibilidade de já os ter em *cache* no navegador web pode ser útil, por exemplo, se quisermos apenas mudar o *step* ou até mesmo reduzir o intervalo de tempo total das amostras e desta forma já as teremos em *cache*.

3.11.1 Algoritmo Desenvolvido

Os dados recebidos dos sensores não são sincronizados, isto é, as capturas e outras informações são enviadas pelos sensores em instantes de tempo diferentes e com frequências de captura diferentes. Por esta razão, surge a necessidade de desenvolver este algoritmo que, através das amostras presentes na BD no intervalo de tempo escolhido, escolhe as amostras que melhor correspondem aos parâmetros introduzidos pelo utilizador.

O funcionamento do algoritmo é descrito em seguida, em forma de texto. Cada um dos passos descritos corresponde a uma ou mais funções *javascript* que foram desenvolvidas para desempenhar

cada uma dessas tarefas. Para exemplificar uma utilização deste algoritmo vai ser usado como referência o seguinte input:

- ***Timeline init*:** 03/08/2022 16:30:00
- ***Timeline end*:** 03/08/2022 18:50:00
- ***Step*:** 10

As datas escolhidas para *Timeline init* e *Timeline end* neste exemplo são no mesmo dia para simplificar a explicação do processo, ou seja, nos exemplos apenas dou foco às horas.

A partir do *input* recebido do formulário preenchido pelo utilizador é realizado o seguinte processamento:

1. Verificação dos *inputs*: verificar se as datas fazem sentido, isto é, se a data final é superior à inicial, verificar se o *step* é válido, ou seja, se está entre 1 e 1440 minutos e por fim, foi decidido estabelecer outro requisito, o intervalo de tempo total tem de ser no mínimo 10 vezes superior ao valor de '*step*', ou seja, se o '*step*' for 10 minutos o intervalo de tempo entre '*Timeline init*' e '*Timeline end*' tem de ser pelo menos 100 minutos.
2. Calcular os *timestamps* de referência. Os *timestamps* de referência são calculados com base nos inputs do utilizador e vão servir para determinar as amostras mais indicadas para serem apresentadas na GUI. Usando o exemplo acima, os *timestamps* de referência vão ser: 16:30:00, 16:40:00, 16:50:00, 17:00:00 e assim sucessivamente até 18:50:00. Ou seja, começa com o '*Timeline init*', depois é adicionado o *step* para o segundo *timestamp*, ao qual é novamente adicionado o *step* para obter o terceiro, e assim sucessivamente até chegar ao '*Timeline end*'.
3. Com base no '*Timeline init*' e no '*Timeline end*', é feito o pedido à API REST para obter todas as capturas (linhas de fogo e de sombra) dentro daquele intervalo de tempo.
4. No caso de não haver amostras naquele intervalo de tempo, a execução termina e é mostrado um *pop-up* com uma mensagem de erro. No caso de haver amostras, é chamada uma função que adiciona a um *array* todos os *timestamps* das amostras recebidas, ordenando-as.
5. Depois, através dos *timestamps* de referência, das datas obtidas no ponto anterior e do *step*, foi implementada uma função, que se encontra na Figura 38, que organiza e calcula os *timestamps* definitivos que vão ser mostrados na GUI.
6. Feito isto, conforme é mostrado na função da Figura 38, através de um ciclo *for* (linhas 165-167), são criadas várias propriedades no objeto '*organizedTimestamps*' em que cada propriedade tem como nome/chave um *timestamp* de referência e é iniciado como um *array* vazio. Depois, através de outro ciclo *for* (linhas 169-183) vão ser preenchidos esses *arrays* vazios com os *timestamps* obtidos da BD que lhe pertencem. Para determinar a que *timestamp* de referência pertence cada *timestamp* das amostras da BD, é usado o desvio (*deviation*) que serve para determinar o *timestamp* intermédio entre cada *timestamp* de referência, que vai definir os limites mínimo e máximo para cada *timestamp* de referência. Para melhor compreensão do que acontece na prática, a Figura 39 apresenta uma linha temporal parcial correspondente ao exemplo que está a ser usado nesta explicação, ou seja, os *timestamps* de referência estão representados como linhas verticais contínuas e os *timestamps* intermédios de cada *timestamp* de referência estão a tracejado. Os pontos que estão apresentados correspondem aos *timestamps* provenientes da BD. Por exemplo, seguindo o exemplo da Figura 39, ao *timestamp* de referência 17:00:00,

vão ficar associados os *timestamps* das amostras entre 16:55:00 e 17:05:00, que no caso são os *timestamps*: 16:57:41 e 17:01:06.

7. Após a organização feita aos *timestamps* nos passos anteriores, é executado um outro ciclo *for* (linhas 185-194) que entre os *timestamps* associados a cada *timestamp* de referência, seleciona o que está mais perto e insere-o no *array* '*finalList*' que será a lista com os *timestamps* definitivos das amostras que vão ser mostradas no mapa da GUI.

8. Por fim, com esta lista de *timestamps* (*finalList*), são selecionados da lista de amostras que veio da BD, as capturas/amostras cujos *timestamps* constam na lista '*finalList*'. As instâncias das linhas de fogo são criadas e armazenadas nas camadas (*layers*) correspondentes. Nota: se houver mais do que uma captura com o mesmo *timestamp* essas outras capturas também são selecionadas para serem mostradas no mapa.

```
157  function calculateDefinitiveTimestamps(referenceTimestamps, databaseTimestamps, step) {
158      var deviation = minutesToMilliseconds(step)/2
159      var organizedTimestamps = {}
160      var idx = 0
161      var finalList = []
162
163      if (databaseTimestamps.length == 0) return []
164
165      for (refTime of referenceTimestamps) {
166          organizedTimestamps[refTime] = []
167      }
168
169      for (refTime of referenceTimestamps) {
170          var min = refTime - deviation
171          var max = refTime + deviation
172          var currentDate = databaseTimestamps[idx]
173
174          while (min <= currentDate && currentDate < max) {
175              organizedTimestamps[refTime].push(currentDate)
176              ++idx
177              if(idx < databaseTimestamps.length) {
178                  currentDate = databaseTimestamps[idx]
179              } else {
180                  break
181              }
182          }
183      }
184
185      for (refTime of referenceTimestamps) {
186          var curr = organizedTimestamps[refTime]
187          if (curr.length == 0) {
188
189          } else if (curr.length == 1) {
190              finalList.push(curr[0])
191          } else {
192              finalList.push(pickClosestDate(refTime, curr))
193          }
194      }
195      return finalList
196  }
```

Figura 38. Excerto de código da função *calculateDefinitiveTimestamps()*

Importa ainda referir que no HTML foram acrescentados alguns elementos ocultos (*hidden*) para armazenar os valores atuais dos parâmetros da *Timeline* que está a ser mostrada, isto é, do '*Timeline init*', do '*Timeline end*' e do '*step*'. Estes valores são atualizados sempre que é feito um pedido de uma nova *Timeline* e servem para o caso de ser utilizado o botão de '*refresh*' que se encontra no canto inferior direito do mapa, que pode ser visto na Figura 37, para que a *Timeline* carregada seja a mais recente.

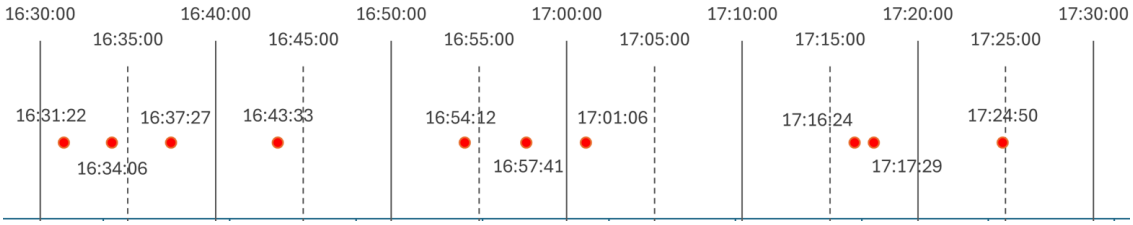


Figura 39. Linha temporal parcial com exemplo de algumas amostras

Uma vez selecionadas e criadas as instâncias das amostras pretendidas, é instanciada a própria barra de navegação da *Timeline* (explicada em mais detalhe na Subsecção 3.13.4) e esta barra adequa-se sempre à quantidade de *timestamps* disponíveis, não ficando necessariamente proporcional ao intervalo de tempo entre as amostras.

3.12 Módulo de Acesso à *API REST*

Como a aplicação do *GesFoGO* está dividida em *backend* e *frontend*, foi necessário implementar um módulo para dar suporte à comunicação entre eles, nomeadamente para a GUI aceder à API REST. Para tal, foi implementado o módulo *./src/js/api-access.js* onde estão desenvolvidas algumas funções para disponibilizar os vários tipos de pedidos HTTP aos restantes módulos da aplicação.

As funções públicas deste módulo são: *getRequest()*, *putRequest()*, *postRequest()* e *deleteRequest()* e correspondem aos pedidos cujo método HTTP é, respetivamente, *GET*, *PUT*, *POST* e *DELETE*. Estas são as funções que os restantes módulos da aplicação utilizam quando é necessário algum pedido à API. Estas funções recebem o *path* do *endpoint*, a função de *callback* e o *body* do pedido (apenas para pedidos *PUT* e *POST*). Estas funções são responsáveis por chamar as funções *optionsBuilder()* e posteriormente *request()* fornecendo as informações necessárias. O excerto de código da implementação destas duas funções encontra-se na Figura 40, em seguida.

```
32  function request(path, options, callback) {
33      fetch(host + path, options)
34          .then(function(response) {
35              if (response.status >= 400 && response.status < 500) {
36                  throw new Error('Bad Request')
37              } else if (response.status >= 500) {
38                  throw new Error('Bad response from server')
39              }
40              return response.json()
41          }).then((response) => {
42              callback(undefined, response)
43          })
44  }
```

```
46  function optionsBuilder(method, body) {
47      const options = {
48          method: method,
49          headers: {
50            'Content-Type': 'application/json',
51            'Authorization': 'Bearer ' + token
52          },
53          mode: 'cors'
54      }
55
56      if (body) {
57          options.body = JSON.stringify(body)
58      }
59
60      return options
61  }
```

Figura 40. Excerto de código das funções *optionsBuilder()* e *request()*

Como é possível observar no lado esquerdo da Figura 40, a função *request()* é responsável por realizar o pedido HTTP à API. Uma vez que este código é executado no *browser*, os pedidos são feitos usando a função global *fetch()* [35], removendo a necessidade de instalar algum módulo adicional para isso. A função *fetch()* recebe o URL do recurso desejado e o objeto '*options*' com as configurações do pedido e retorna uma *Promise* de objeto *Response*. Assim que a resposta a

esse pedido estiver pronta, é executada uma função que conforme o seu *status code* HTTP envia a resposta adequada (linhas 34-40 da Figura 40). Para concluir, é chamada a função de *callback*, conforme recebido no parâmetro da função.

Ainda na Figura 40, é possível observar a função *optionsBuilder()*, que prepara o objeto '*options*', onde constam as opções/configurações para o pedido que vai ser realizado, com base nos parâmetros recebidos. Apesar da função *fetch()* suportar muitas outras opções [35], estas são as necessárias para a aplicação do *GesFoGO*: o método HTTP (linha 48), os headers '*Content-Type*' e '*Authorization*' (linhas 49-52), o 'mode' que no caso tem o valor '*cors*' (linha 53), e por fim, se for um pedido que tenha '*body*', é adicionado ao objeto '*options*' convertido em *string* (linhas 56-58). Em relação à autenticação (linha 51), seria utilizado o *token* de sessão do utilizador autenticado, mas como a GUI acabou por ficar sem forma de se fazer *login*, como medida provisória, é utilizado um *token* que se encontra no ficheiro de configuração.

3.13 Módulo *Leaflet*

As funcionalidades principais da aplicação do *GesFoGO* foram implementadas utilizando o módulo *Leaflet*, como já foi mencionado na Secção 3.9. O *Leaflet*[50] é uma biblioteca *javascript open-source* para integração de mapas interativos, cujas capacidades podem ser estendidas através de variados *plugins* organizados por categorias. Ou seja, todas as funcionalidades relativas ao mapa que está no *MainVisor* e à interação com o mesmo foram concretizadas através deste módulo e também de alguns *plugins* para algumas funcionalidades extra, nomeadamente o *leaflet-freehandshapes* para ser possível desenhar linhas de fogo no mapa e o *leaflet-semicircle* para representar a zona de visualização dos sensores que estão no modo de varrimento (*sweep*).

Nas subsecções que se seguem vai ser explicado como foram implementadas as funcionalidades que estão relacionadas com o *Leaflet*, começando pela adição do mapa, a criação dos sensores, o desenho de linhas de fogo no mapa e a barra de navegação da *timeline*.

3.13.1 Adicionar o Mapa à GUI

A primeira tarefa a fazer para usufruir das funcionalidades do *Leaflet* é adicionar o mapa à GUI da aplicação. Para isso foram seguidas as instruções do guia de início rápido [36], que fazem parte da documentação do *Leaflet* [37].

Primeiramente é necessário criar um elemento * * na página HTML onde o mapa vai ficar, e também incluir, na secção *<head>* do documento HTML, o ficheiro CSS do *Leaflet*, seguido do seu ficheiro *javascript*. Feito isto, já é possível criar a instância do mapa, fornecendo algumas configurações iniciais, como é possível verificar no excerto de código da Figura 41, em seguida.

```
12  exports.mymap = L.map('map', {
13      center: [config.mapCenter.latitude, config.mapCenter.longitude],
14      zoom: config.mapZoom,
15      zoomControl: false,
16      fullscreenControl: true
17  })
```

Figura 41. Excerto de código da criação da instância do mapa do *Leaflet*

[50] Leaflet - `https://leafletjs.com/`

Como é mostrado na Figura 41, são configurados o centro e o nível de zoom do mapa. Como esses valores podem ser mudados conforme necessário e para não estarem *hardcoded*, ficam armazenados no ficheiro de configuração *config.json*, que está importado no objeto '*config*', como consta nas linhas 13 e 14. Depois, é necessário configurar o tipo de mapa que pretendemos, isto é, o estilo e/ou origem do mapa. No caso do *GesFoGO*, como é mostrado na Figura 42, temos 3 estilos/tipos diferentes: o mapa satélite da *Google* como *default* (linhas 13-14), o mapa satélite da API *maptiler* [38] (linhas 16-17) e um mapa estilo carta topográfica também obtido da API *maptiler* [38] (linhas 19-20). A configuração de cada um destes tipos de mapa requer a criação de uma instância de *TileLayer*, usando a função *L.tileLayer()*, que recebe como primeiro parâmetro o URL da API onde está hospedado o mapa que pretendemos utilizar e como segundo parâmetro (opcional) recebe um objeto que suporta vários tipos de configuração, que no *GesFoGO* apenas estamos a usar para configurar a '*attribution*' que corresponde à hiperligação que fica no canto inferior direito do mapa, que no caso é para o site oficial do projeto.

```
11  //Tiles
12  var mapTiles = {
13      "google": L.tileLayer('http://www.google.cn/maps/vt?lyrs=s@189&gl=cn&x={x}&y={y}&z={z}', {
14          attribution: '<a href="http://www.gesfogo.ulpgc.es/index.php/es/">GesFoGO ARDITI</a>'
15      }).addTo(variables.mymap),
16      "satelite": L.tileLayer('https://api.maptiler.com/maps/hybrid/{z}/{x}/{y}.jpg?key=HsE7HIt6q0MJ6monGnhT', {
17          attribution: '<a href="http://www.gesfogo.ulpgc.es/index.php/es/">GesFoGO ARDITI</a>'
18      }),
19      "cartiqo": L.tileLayer('https://api.maptiler.com/maps/topographique/{z}/{x}/{y}.png?key=HsE7HIt6q0MJ6monGnhT', {
20          attribution: '<a href="http://www.gesfogo.ulpgc.es/index.php/es/">GesFoGO ARDITI</a>'
21      })
22  }
23
24  //Add layers
25  var layerGroups = {
26      "Sensors": variables.sensorGroup
27  }
28  var control = L.control.layers(mapTiles, layerGroups).addTo(variables.mymap)
29
30  var command = L.control({ position: 'topleft' })
31  command.onAdd = listeners.commandOnAddListener
32  command.addTo(variables.mymap)
33
34  var controls = L.control({ position: 'bottomright' })
35  controls.onAdd = listeners.controlOnAddListener
36  controls.addTo(variables.mymap)
```

Figura 42. Excerto de código da configuração inicial do mapa do *Leaflet*

Por fim, é inicializada a camada *layerGroups*, onde serão posteriormente adicionadas as instâncias dos sensores. Na linha 28, da Figura 42, é chamada a função *L.control.layers()* onde é passado como primeiro argumento a camada base, que corresponde ao mapa em si ('*mapTiles*'), e como segundo argumento as camadas que se vão sobrepor ao mapa ('*layerGroups*'). Além dessas camadas, são acrescentados alguns controlos ao mapa, nomeadamente o menu '*Hide Layers*', que fica no canto superior esquerdo do mapa (linhas 30-32), e os botões de atualizar e recentrar o mapa que ficam no canto inferior direito do mapa (linhas 34-36). O resultado destas operações/configurações pode ser observado nas figuras da Secção 3.10.

3.13.2 Criação dos Sensores

Para adicionar os ícones e área de cobertura dos sensores foi necessário utilizar um *plugin* adicional, o *Leaflet.GeotagPhoto*[51]. A instalação deste *plugin* consiste apenas em adicionar, na secção <*head*> do documento HTML, os ficheiros CSS e *javascript* necessários, que são obtidos remotamente

[51] Leaflet.GeotagPhoto - `https://github.com/nypl-spacetime/Leaflet.GeotagPhoto`

quando a página é carregada. No repositório deste *plugin*, são disponibilizadas as imagens que vão representar os sensores no mapa. Essas imagens foram adicionadas à pasta do projeto para serem usadas.

Na Figura 43 encontra-se uma parte da função *getSensorInstancesFromDatabaseCallback()* que é chamada depois de ser feito o pedido à API REST para obter a informação dos sensores que estão armazenados na BD, e tem como responsabilidade criar as instâncias dos sensores e adicioná-los ao mapa. Como é apresentado na Figura 43, primeiramente são criados alguns objetos que possuem as configurações de cada sensor e, de seguida, é criada cada instância de sensor (linhas 74-80) usando a função *L.geotagPhoto.camera()*. Nota: este excerto de código encontra-se dentro de um ciclo *for*.

```
22      var cameraPoint = [longitude, latitude]
23      var type = 'Feature'
24
25      var properties = {
26          angle: lens_fov,
27          bearing: (yaw_min + yaw_max) / 2,
28          distance: config.default_sensor_distance
29      }
30
31      var geometries = {
32          type: "Point",
33          coordinates: cameraPoint
34      }
35
36      var points = {type: type, properties: properties, geometry: geometries}
37
38      var options = {···
56      }
57
58      if (sweep_status !== 0) {···
72      }
73
74      var cam = L.geotagPhoto.camera(points, options)
75      cam._cameraIcon.options.iconUrl = './gesfogo/assets/images/camera.png'
76      cam._targetIcon.options.iconUrl = './gesfogo/assets/images/down-arrow.png'
77      cam._targetIcon.options.iconSize = [25, 40]
78      cam._angleIcon.options.iconUrl = './gesfogo/assets/images/marker.svg'
79      cam._angleIcon.options.iconSize = [25, 20]
80      cam.options.id = result[i].current_deploy[0].id
```

Figura 43. Excerto de código da criação dos sensores usando o *Leaflet.GeotagPhoto*

Ainda a respeito da Figura 43, a função *L.geotagPhoto.camera()* (linha 74) recebe 2 parâmetros: '*points*' e '*options*'. O '*options*' diz respeito às configurações do triângulo que representa a área que o sensor consegue cobrir/visualizar quando está no modo fixo, onde podemos definir a cor, opacidade, ângulo máximo, entre outras. Já o objeto '*points*' (linha 36) corresponde às informações como a localização (latitude e longitude) do sensor (linhas 31-34), o ângulo de visão (linha 26) que é definido conforme o campo de visão da lente física, a direção para onde o sensor está a apontar (linha 27) que corresponde ao valor de *yaw* e, por fim, a distância (linha 28) desde o sensor até ao final do triângulo, que representa o campo de visão do mesmo. Nas reuniões técnicas, foi decidido que este valor devia ser igual para todos os sensores, independentemente daquele ser ou não o alcance real, visto que este valor poderia ser muito inferior ou muito superior, no caso do campo de visão estar ou não obstruído, respetivamente. Portanto, é um valor constante que está armazenado no ficheiro de configuração *./config.json*.

No caso do sensor estar em modo de varrimento (*sweep*), em vez de ter o triângulo que delimita a área de visualização/alcance do sensor, tem um círculo parcial correspondente aos valores de *yaw* mínimo e máximo configurados, como é ilustrado na Figura 32. Do ponto de vista da implementa-

ção, foi necessária a utilização do módulo *leaflet-semicircle*, cuja utilização é mostrada na Figura 44, com outro excerto de código da função *getSensorInstancesFromDatabaseCallback()*.

```
58      if (sweep_status !== 0) {
59          L.semiCircle([latitude, longitude], {
60              radius: config.default_sensor_distance,
61              startAngle: yaw_min,
62              stopAngle: yaw_max,
63              color: '#000000',
64              opacity: 0.5,
65              weight: 2,
66              fillOpacity: 0.3,
67              fillColor: '#b4b4b4'
68          }).addTo(variables.mymap)
69
70          options.outlineStyle.opacity = 0
71          options.fillStyle.fillOpacity = 0
72      }
```

Figura 44. Excerto de código da utilização do módulo *leaflet-semicircle*

Como é possível observar na Figura 44, para criar uma instância de *Semicircle* é usada a função *L.semiCircle()* (linhas 59-67) e de seguida, chamada a função *addTo()* para adicioná-la ao mapa (linha 68). A função *L.semiCircle()* recebe as coordenadas (latitude e longitude) onde vai ficar, e um objeto com algumas configurações como o raio (linha 60) que segue a mesma lógica explicada acima, os limites mínimo e máximo de *yaw* para saber onde começa e termina o semicírculo, as cores do contorno e do preenchimento e a opacidade. Por fim, nas linhas 70 e 71 a opacidade do contorno e do preenchimento criados através do processo para os sensores em modo fixo, são configurados para ficarem invisíveis e mostrar apenas o semicírculo no mapa.

3.13.3 Desenho de Linhas de Fogo no Mapa

O *GesFoGO* também possui a opção dos utilizadores/operadores desenharem diretamente no mapa as linhas de fogo onde sabem que existe fogo, mesmo que não tenha sido detetado pelos sensores, com base na sua experiência e/ou informações complementares, por exemplo, de meios aéreos ou de outros profissionais que estão no terreno. Para acrescentar esta funcionalidade foi necessário outro *plugin* para o *Leaflet*, o *Leaflet.FreeHandShapes*.

É possível desenhar linhas de fogo e também linhas de sombra. Na Figura 45, encontra-se ilustrado o excerto de código para a criação da instância de *L.FreeHandShapes* para as linhas de fogo, que representa uma espécie de camada que é adicionada ao mapa e que pode estar ativada ou desativada conforme o modo escolhido no menu '*Draw Tools*' que fica acima do mapa (Figura 33), onde ficam guardadas as linhas/polígonos desenhados temporariamente.

Para a criação das instâncias *L.FreeHandShapes*, como mostra a Figura 45, é necessário passar como argumentos algumas configurações, como um '*polyline*' (linhas 31-34) que representa as linhas que estão a ser mostradas enquanto estamos a desenhar, onde é necessário especificar uma cor (linha 32) e um fator de suavidade/simplificação (linha 33) em que, quanto maior o valor, maior a suavidade dos contornos, conforme é feito zoom no mapa. Também são passadas como argumento as configurações para '*polygon*', que é baseado em *polyline*, portanto, além dos campos equivalentes, possui a cor do preenchimento (linha 26), a opacidade (linha 27) que é inferior à linha de contorno, o peso (linha 28) que é a largura do traço em pixeis e, ainda, '*className*' (linha 24) para identificação posterior. Com base no *Polyline* traçado pelo utilizador e no nível de suavidade/-simplificação configurado, é feita a conversão para um *Polygon*, que passa a ser mostrado no mapa

```
22  exports.drawnfireLines = new L.FreeHandShapes({
23      polygon: {
24          className: 'drawnFireLine',
25          color: config.colors.inputFireLineDrawn,
26          fillColor: config.colors.inputFireLineDrawn,
27          opacity: 0.7,
28          weight: 3,
29          smoothFactor: 1
30      },
31      polyline: {
32          color: config.colors.inputFireLineDrawn,
33          smoothFactor: 1
34      },
35      simplify_tolerance: 0.001,
36      merge_polygons: true
37  })
```

Figura 45. Excerto de código da utilização do módulo *L.FreeHandShapes*

após o utilizador soltar o botão esquerdo do rato. Além destas configurações, é ainda configurada a tolerância de simplificação (linha 35) em que, quanto menor o número, menor a simplificação, ou seja, mais pontos utilizados para delimitar os polígonos e, ainda, o booleano '*merge_polygons*' que está a '*true*', isto é, no caso dos polígonos desenhados estarem sobrepostos junta-os num só.

Quanto ao funcionamento desta camada, é possível ter vários estados/funções, através da seguinte linha de código: "*variables.drawnfireLines.setMode(<MODO>);*"em que <MODO> é substituído por '*add*', '*view*' ou '*delete*' para adicionar/desenhar, visualizar ou apagar, respetivamente. Através da opção que é selecionada no menu '*Draw Tools*' é executado o comando correspondente.

Assim que o utilizador termina os desenhos, deve carregar no botão '*Save Draws*' que se encontra na parte superior do *NodeVisor* (Figura 34), que vai resultar na extração das linhas de fogo desenhadas que se encontram armazenadas na instância de *L.FreeHandShapes*, para formatá-las corretamente e fazer um pedido à API REST para gravar estes dados de forma persistente. Depois disso, é chamada a função *clearLayers()* para as instâncias de *L.FreeHandShapes* para as linhas de fogo e linhas de sombra, que elimina os polígonos que estavam desenhados mas já foram guardados na BD. Esta chamada à função *clearLayers()* também é feita quando se carrega no botão '*Erase Draws*' (Figura 34) que apaga as linhas que estão desenhadas mas que ainda não foram guardadas.

3.13.4 Barra de Navegação da *Timeline*

Por fim, foi necessária a utilização do *plugin LeafletSlider*[52], que serve para adicionar a barra de navegação que se encontra no canto inferior esquerdo do mapa, como ilustrado na Figura 37. Ao contrário dos restantes módulos em que a instalação é feita através do *NPM*, este requer o *download* de 4 ficheiros com o código necessário e também que sejam carregados na secção *<head>* do ficheiro HTML da GUI do *GesFoGO*. Esses ficheiros são o *SliderControl.js*, que é o código do *LeafletSlider* propriamente dito, e ainda 3 ficheiros do módulo *jQuery* com o respetivo código *javascript* e CSS: *jquery-1.9.1.min.js*, *jquery-ui.js* e *jquery-ui.css*.

Quanto à sua utilização, segue o mesmo processo que os restantes *plugins*, ou seja, é criada uma instância através da função *L.control.sliderControl()*, que recebe um objeto com a configuração pretendida e, em seguida, é adicionada ao mapa. Como é mostrado no excerto de código da Figura 46, a instância do *slider* é criada (linha 265) com a configuração da posição onde a barra deve ficar localizada dentro do mapa, no caso foi no canto inferior esquerdo (*bottomleft*), depois é passada a variável onde se encontram os polígonos (linhas de fogo e sombra) que pretendemos mostrar, com base nos inputs da *Timeline*. Além disso, recebe a configuração de '*follow*' que se tiver o valor 1

[52] LeafletSlider - `https://github.com/dwilhelm89/LeafletSlider`

(ou *true*) exibe apenas as amostras do *timestamp* que está selecionado na barra de navegação e, no caso de estar a *null* (ou *false*), apresenta as amostras do *timestamp* selecionado e também dos *timestamps* anteriores. Recebe ainda, o '*timeAttribute*' que serve para indicar qual dos atributos dos polígonos recebidos representa o *timestamp* a utilizar para a organização da barra de navegação.

```
264 function createSlider() {
265     variables.sliderControl = L.control.sliderControl({ position: 'bottomleft', layer: variables.layersGeoJSON, follow: 1, timeAttribute: 'time' })
266     variables.mymap.addControl(variables.sliderControl)
267     variables.sliderControl.startSlider()
268 
269     variables.sliderControl.on('rangechanged', listeners.sliderRangeChangedListener)
```

Figura 46. Excerto de código da utilização do módulo *LeafletSlider*

Após a criação da instância para a barra de navegação, é necessário adicioná-la ao mapa, através da chamada à função *addControl()* (linha 266) e ainda, fazer uma chamada à função *startSlider()* (linha 267) para inicializá-la, conforme é apresentado na Figura 46. Feito isto, de modo a obter exatamente o comportamento pretendido na utilização da barra de navegação, é configurado um *listener* (linha 269), a ser executado sempre que é feito qualquer movimento na barra de navegação.

A necessidade deste *listener* surge porque na aplicação do *GesFoGO*, para além de pretendermos que os polígonos (linhas de fogo e sombra) sejam mostrados conforme o *timestamp* que está selecionado na barra de navegação (que já é feito automaticamente pelo módulo *LeafletSlider*), também queremos que seja mostrada/atualizada a imagem que aparece no *ImageVisor*, de acordo com o *timestamp* atual da barra de navegação e do sensor que está selecionado no momento. A implementação desta função é relativamente simples, e pode ser vista na Figura 47.

```
282 exports.sliderRangeChangedListener = function sliderRangeChangedListener(event) {
283     exports.toggleFireLinesDBListener()
284     exports.toggleShadowLinesDBListener()
285     exports.toggleFireLinesDrawnListener()
286     exports.toggleShadowLinesDrawnListener()
287 
288     document.getElementById('current-timestamp').value = event.markers[0].options.time
289 
290     var sensorDeployId = document.getElementById('selected-sensor-deploy-id').value
291     if (sensorDeployId != '') {
292         functions.updateImageVisor('visible')
293     }
294 }
```

Figura 47. Excerto de código da função *sliderRangeChangedListener()*

De acordo com a Figura 47, o valor do *timestamp* que está selecionado na barra de navegação é armazenado num elemento oculto HTML, o '*current-timestamp*' (linha 288). Este valor é útil para quando são gravados novos polígonos desenhados na BD, sabermos que *timestamp* lhes devemos atribuir. Em seguida é usado um outro elemento HTML oculto onde está indicado o identificador do sensor que está selecionado naquele momento (linha 290), para saber de qual dos sensores é suposto mostrar as imagens. Por fim, é feita a verificação desse identificador (linha 291), porque no caso de não estar selecionado nenhum sensor não é mostrada nenhuma imagem, portanto a execução da função termina. No caso de haver um sensor selecionado é chamada a função *functions.updateImageVisor()* (linha 292), que por defeito, mostra a imagem visível.

Relativamente às funções que são chamadas nas linhas 283-286 da Figura 47, a sua necessidade surgiu devido a um *bug* em que os polígonos/linhas de fogo por vezes podiam aparecer a mais

ou a menos conforme se navega na barra, isto é, por vezes as linhas de fogo e/ou de sombra que apareciam no mapa não correspondiam ao que estava selecionado nas *checkboxes* do menu '*Hide Layers*' (Figura 33). Estas funções são os *listeners* que são executados quando é feito clique nessas *checkboxes*, que nesta situação são chamadas para que os polígonos mostrados no mapa correspondam ao que está selecionado no menu.

Fica assim concluída esta secção acerca da utilização que foi dada ao módulo *Leaflet* e aos respetivos *plugins*, que serviram para a implementação das principais funcionalidades da aplicação do *GesFoGO*. Posto isto, segue-se uma secção acerca dos mecanismos dos *browsers* para a partilha de recursos entre domínios, mantendo a segurança.

3.14 *Cross-Origin Resource Sharing* (CORS)

Ainda numa fase inicial, devido a termos feito a separação da aplicação em API e GUI, e a sua utilização através do navegador web, surge o *Cross-Origin Resource Sharing* (CORS) [39], um mecanismo dos navegadores web para a partilha de recursos entre domínios. Por defeito, os browsers seguem a *Same-Origin Policy* (SOP) [40], que apenas permite que sejam feitos pedidos ao mesmo domínio que os solicitou. O CORS permite que sejam realizados pedidos entre diferentes domínios (*origins*) especificados/configurados no servidor, permitindo que páginas web interajam com recursos de domínios diferentes de forma segura.

A necessidade de configurar o CORS surgiu quando foi iniciada a implementação da GUI em *Node.js* ao testá-la no *browser*, porque mesmo executando os componentes (API e GUI) na mesma máquina, estavam à escuta em portos diferentes. Apesar do IP ser o mesmo, são considerados domínios distintos. Como a API REST ainda não estava preparada para isto, surge o erro apresentado na Figura 48 na consola do navegador web, que indica que o pedido foi bloqueado devido à política do CORS.

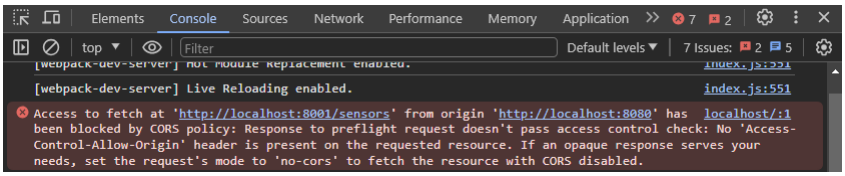


Figura 48. Erro obtido na consola do *browser* devido ao CORS

A versão 8 do *Laravel*, que foi a utilizada para desenvolver a API REST do *GesFoGO*, por defeito já vem com o *middleware* '*HandleCors*' [41] incluído. Este *middleware* é responsável por responder aos pedidos *preflight* (método HTTP *OPTIONS*) que o *browser* realiza automaticamente para verificar se o servidor compreende o protocolo CORS [42]. Para funcionar corretamente apenas requer alguma configuração no ficheiro *./config/cors.php*, nomeadamente a propriedade '*allowed_methods*' com o valor '*['GET', 'PUT', 'POST', 'DELETE']*' e a propriedade '*allowed_origins*' com o valor '['`https://api.gesfogo.ulpgc.es`']' que é o domínio da API da ULPGC, que realiza os pedidos à API desenvolvida. Durante o desenvolvimento, o domínio '`http://localhost:8080`' também estava incluído na lista para permitir a implementação e os respetivos testes locais da GUI, utilizando a API que já estava disponível no servidor.

No que diz respeito ao *frontend*, também foram necessárias algumas alterações para corrigir os erros gerados pelo CORS. Após realizar o *debug* ao módulo que realiza os pedidos à API (*api-*

access.js), o problema detetado foi que o '*body*' dos pedidos da GUI à API não estava corretamente formatado, daí não estarem compatíveis com o conteúdo da resposta ao pedido *preflight* que o *browser* realiza ao servidor da API REST. A resolução deste problema passou por explicitar o '*Content-Type*' como '*application/json*' e também configurar o modo do pedido (*RequestMode*) [43] como '*cors*' que é feito através da propriedade '*mode*' que pertence ao objeto '*options*', onde são configuradas as opções do pedido. Estas duas configurações encontram-se nas linhas 50 e 53 da função *optionsBuilder()* que se encontra na Figura 40.

4 Colocação em Produção e Testes

Neste capítulo, é explicado como foi feita a colocação em produção (*deploy*) da aplicação do *GesFoGO* no servidor do EELab e a integração das partes desenvolvidas pela ARDITI e pela ULPGC, incluindo os desafios que surgiram durante esse processo e como foram contornados. Por fim, são expostos os resultados dos testes no terreno e das demonstrações que se realizaram.

4.1 *Deploy* da aplicação do *GesFoGO* no Servidor do EELab

Após as componentes da responsabilidade da ARDITI (BD, API REST e GUI) estarem concluídas (eventualmente com alguns *bugs*, mas no geral, prontas), foi necessário fazer *deploy* para um servidor, de forma a ser possível aceder à aplicação através de qualquer máquina usando um navegador web, necessário para fazer a integração da parte desenvolvida pela ARDITI com a parte desenvolvida pela ULPGC, explicada mais detalhadamente na Secção 4.3. Até ao momento, o desenvolvimento e os testes tinham sido feitos apenas localmente.

Como o EELab já tinha um servidor, pertencente ao servidor principal da ARDITI, onde estava hospedado apenas o seu site, e sem a possibilidade de ter outro(s) servidores para cada projeto que está a ser desenvolvido pelo EELab, foi-me atribuída a tarefa de configurar o servidor já existente de forma a disponibilizar no servidor a *API REST* e GUI do *GesFoGO* (como servidores separados), mantendo o site do EELab disponível no URI principal. O servidor do EElab é acedido através de `https://eelab.arditi.pt/`. Portanto, a ideia era manter o site a funcionar incluindo todos os seus recursos, mas para determinados URI's específicos, em vez de serem processados pelo *WordPress*[53] (plataforma onde o site do EELab foi desenvolvido), redireciona-os para uma outra aplicação específica:

- `https://eelab.arditi.pt/` -> Site *WordPress* do EELab
- `https://eelab.arditi.pt/gesfogo-api` -> API REST do *GesFoGO* (*Laravel*)
- `https://eelab.arditi.pt/gesfogo` -> GUI do *GesFoGO* (*Node.js*)

O objetivo era disponibilizar a aplicação do *GesFoGO*, mas deixar o servidor configurado de forma a ser possível e relativamente simples, posteriormente, adicionar outros sites/aplicações *web* para outros projetos do EELab, seguindo a mesma lógica.

Antes de fazer a configuração do servidor para ter este comportamento, foi decidido instalar um painel de controlo com uma interface gráfica para gestão do servidor do EELab, para facilitar a configuração e também a futura manutenção do servidor. Todo o processo de escolha, instalação e teste é descrito na Secção 4.2. Após a instalação e configuração inicial desse painel de controlo, procedi à configuração do servidor para cumprir os objetivos mencionados acima.

Devido à limitação de apenas haver um servidor alocado ao EELab para hospedar o site e todas as aplicações dos vários projetos em vigor, foi necessário arranjar uma abordagem alternativa. O servidor do EELab já estava a utilizar o servidor HTTP *Apache*, portanto, a primeira abordagem foi tentar perceber se, através de uma configuração específica do Apache, era possível obter o resultado pretendido. Após pesquisa e teste de algumas alternativas que não funcionavam como pretendido, acabei por encontrar uma solução que consiste em usar *Virtual Hosts* para conseguir

[53] WordPress - `https://wordpress.com/`

hospedar vários sites/aplicações num mesmo servidor e, posteriormente, configurar um *Reverse Proxy* no Apache que vai redirecionar os pedidos a cada URI para a respetiva aplicação/site.

Primeiramente, para a configuração de *Virtual Hosts* que permitem hospedar vários sites/aplicações no servidor do EELab, foi seguido o tutorial [44]. Existem 3 formas de o fazer e que são descritas nesse tutorial, pode ser *Name-based*, *IP-based* ou *Port-based*. Neste caso o mais adequado é o *Port-based Virtual Hosting*, porque pretendemos ter cada aplicação/site hospedado num determinado porto. Os passos seguidos para esta configuração foram os seguintes:

1. Adicionar as linhas "*Listen 8081*"e "*Listen 8082*"ao ficheiro */etc/apache2/ports.conf*, para o *Apache* ficar à escuta nestes 2 portos para a API REST e GUI do *GesFoGO*, respetivamente.
2. Colocar as pastas de ambas as aplicações (API REST e GUI) dentro da pasta */var/www/* do servidor.
3. Mudar o proprietário dessas pastas para '*www-data:www-data*'.
4. Criar os ficheiros de configuração para cada um dos *virtual hosts* dentro da diretoria */etc/apache2/sites-available/*, no caso, os ficheiros *gesfogo-api.conf* e *gesfogo-gui.conf*, onde são especificadas informações como o porto onde a aplicação vai estar a correr e também a diretoria onde está localizada a sua pasta principal, como pode ser observado no exemplo da Figura 49 para o ficheiro *gesfogo-api.conf*, da API REST do *GesFoGO*.
5. Executar os comandos '*sudo a2ensite gesfogo-api*' e '*sudo a2ensite gesfogo-gui*' para as aplicações começarem a funcionar.
6. Por fim, reiniciar o Apache através do comando '*sudo systemctl restart apache2*'.

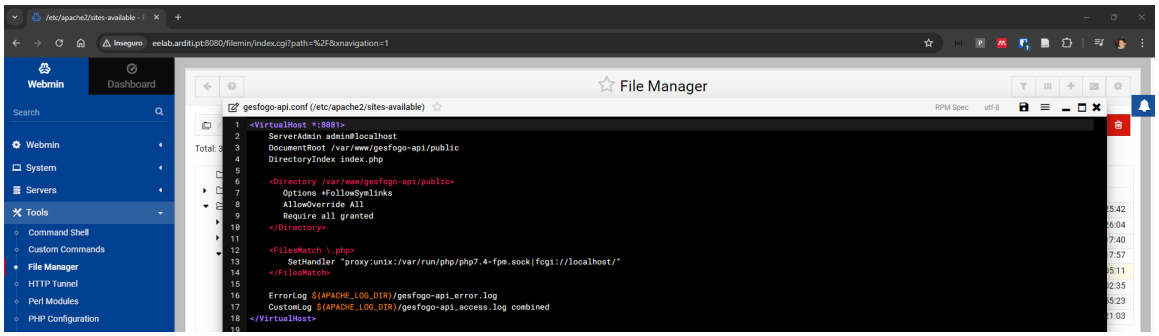


Figura 49. Configuração do *Virtual Host* para a API REST

Em segundo lugar, para configurar o *Reverse Proxy*, que é responsável por direcionar os pedidos dos URI's (mencionados no início desta secção) para a respetiva aplicação, foram necessárias as seguintes alterações/configurações:

1. Como o site *WordPress* do EELab está hospedado em `https://eelab.arditi.pt/`, todos os URI's seriam processados pelo *WordPress* e nunca chegariam ao Apache para o processar como pretendido. Portanto, é necessário ir ao ficheiro *.htaccess*, que fica na diretoria */var/www/eelab.arditi.pt/htdocs/.htaccess*, e adicionar as linhas:
   -RewriteCond %{REQUEST_URI} !^/gesfogo.*$
   -RewriteCond %{REQUEST_URI} !^/gesfogo/.*$
   -RewriteCond %{REQUEST_URI} !^/gesfogo-api.*$

-RewriteCond %{REQUEST_URI} !^/gesfogo-api/.*$
para que o *WordPress* ignore os URI's especificados para ser o *Apache* a processá-los.

2. Para configurar o *Reverse Proxy* é necessário adicionar algumas configurações ao ficheiros de configuração do *Apache* relativos ao site do EELab, que tem 2 ficheiros na pasta */etc/apache2/sites-available/*: o *eelab.arditi.pt.conf* e o *eelab.arditi.pt-le-ssl.conf*, que são para o site a correr no porto 80 e no porto 443, para o site em HTTP e HTTPS, respetivamente. As configurações nestes dois ficheiros são idênticas, à exceção do porto. Portanto, é necessário adicionar algumas configurações a esses ficheiros, que podem ser observadas nas linhas 22-33 da Figura 50. Através das configurações *ProxyPass* e *ProxyPassReverse*, necessárias para cada URI que pretendemos especificar, determina-se a que porto local cada URI vai ser direcionado, por exemplo, o URI */gesfogo* vai ser direcionado para a aplicação a correr em `http://localhost:8082` que é a GUI do *GesFoGO*.

3. Por fim, reiniciar o Apache através do comando '*sudo systemctl restart apache2*'.

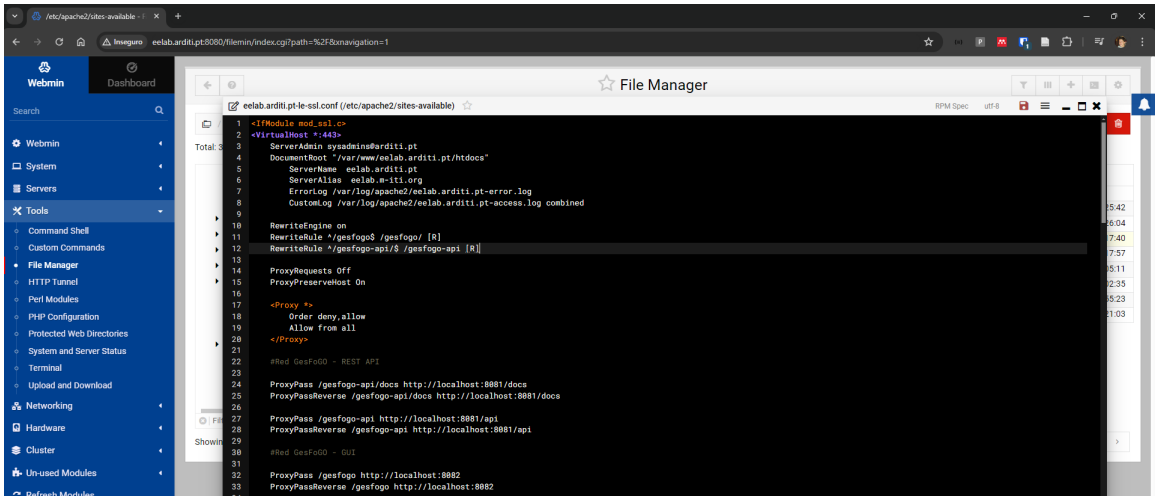


Figura 50. Configurações necessárias para o *Reverse Proxy*

Após todas estas configurações, algumas conseguidas após várias tentativas e erros, as aplicações ficaram a funcionar como tínhamos idealizado. Além disso, para adicionar outras aplicações e/ou sites ao servidor basta seguir estes mesmos passos adequando ao caso em questão.

4.2 Painel de Controlo para o Servidor do EELab

Antes de fazer a configuração do servidor que está descrita na Secção 4.1 foi decidido instalar um painel de controlo com uma interface gráfica para gestão do servidor do EELab, para facilitar a configuração e também a futura manutenção do servidor, por exemplo, para gerir as bases de dados, manipular o sistema de ficheiros, editar ficheiros, alterar configurações, executar comandos, monitorizar parâmetros como utilização de CPU, de memória, espaço de armazenamento, atualizações, etc. Além disso, estes painéis de controlo são acedidos remotamente através de um *browser*, eliminando a necessidade de ter instalados programas como clientes SSH, como por exemplo o *PuTTY*[54], e/ou clientes FTP, como o *FileZilla*[55].

[54] PuTTY - `https://www.putty.org/`
[55] FileZilla - `https://filezilla-project.org/`

Através de alguma pesquisa, foram analisados alguns programas que oferecem estas funcionalidades, tendo em consideração alguns critérios de preferência, nomeadamente, ser um programa gratuito e não requerer uma instalação limpa do Sistema Operativo, visto que, o servidor já tinha o site do EELab, as bases de dados, e também algumas configurações feitas. O Sistema Operativo instalado no servidor é a distribuição *Ubuntu* 20.04 LTS de *Linux*. Com base nestas caraterísticas foram selecionados dois para experimentar e decidir qual o melhor: o ***Webmin***[56] e o ***aaPanel***[57].

Para a realização destes testes não interferir nem danificar o servidor do EELab, foi utilizado o programa *Oracle VirtualBox*[58], onde foram criadas máquinas virtuais para instalar e testar ambos os painéis de controlo. Para tentar garantir que estes programas não causariam estragos ao servidor no seu estado atual, comecei por criar uma máquina virtual com a mesma versão do *Ubuntu* que está no servidor, depois fiz o *setup* inicial e repliquei o site do EELab e a base de dados do *GesFoGO*. Como é possível observar na Figura 51, após cada adição/alteração fiz um *Snapshot* para que, no caso de haver algum erro, seja possível voltar atrás para uma versão a funcionar.

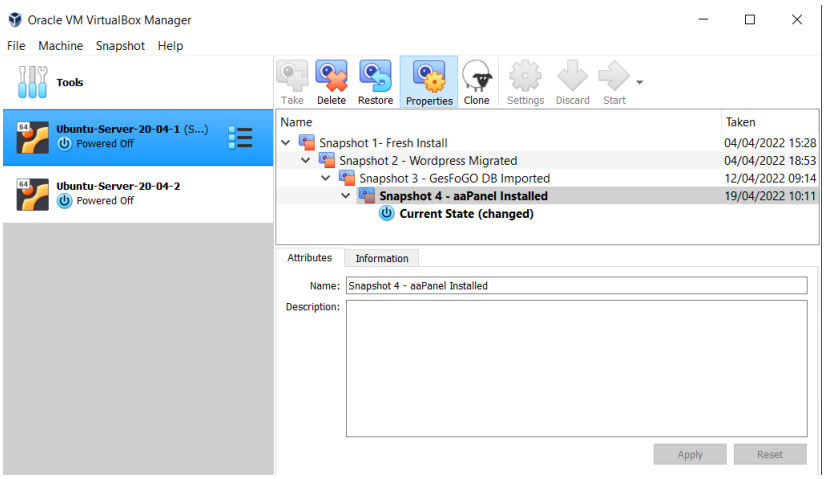


Figura 51. Máquinas virtuais criadas com o *VirtualBox*

Portanto, como mostra a Figura 51, o *Snapshot 1* foi feito após ter a instalação do *Ubuntu* concluída e também a instalação do *Apache*, do *MySQL*[59] e do PHP[60], depois foi acrescentado o site do EELab e criado o *Snapshot 2*. De seguida acrescentei a BD do site do EELab e também do *GesFoGO* e criei o *Snapshot 3*. Com base neste *Snapshot 3* fiz uma cópia desta máquina virtual para ter uma para testar cada painel, o *Webmin* e o *aaPanel*.

No que diz respeito ao ***aaPanel*** (Figura 52), após a sua instalação na respetiva máquina virtual, surgiram alguns erros de fácil resolução. Após isso tive alguma dificuldade em determinar exatamente qual o IP e porto a usar para aceder ao painel. Após ultrapassar esses contratempos, consegui utilizar e testar o painel do *aaPanel*. A sua interface gráfica é apelativa, simples e possui as funcionalidades pretendidas. Além das funcionalidades normais deste tipo de software, o *aaPanel* possui uma secção '*Website*', como é mostrado na Figura 52, que permite a criação de sites e suporta

[56]Webmin - `https://webmin.com/`
[57]aaPanel - `https://www.aapanel.com/`
[58]Oracle VirtualBox - `https://www.virtualbox.org/`
[59]MySQL - `https://www.mysql.com/`
[60]PHP - `https://www.php.net/index.php`

WordPress e também *Node.js*. No entanto, apesar de inicialmente parecer uma funcionalidade útil para o servidor do EELab, não funcionava como pretendido e as configurações necessárias não eram muito claras. Por fim, notei também que no geral as páginas demoravam mais tempo a carregar quando comparado ao *Webmin*.

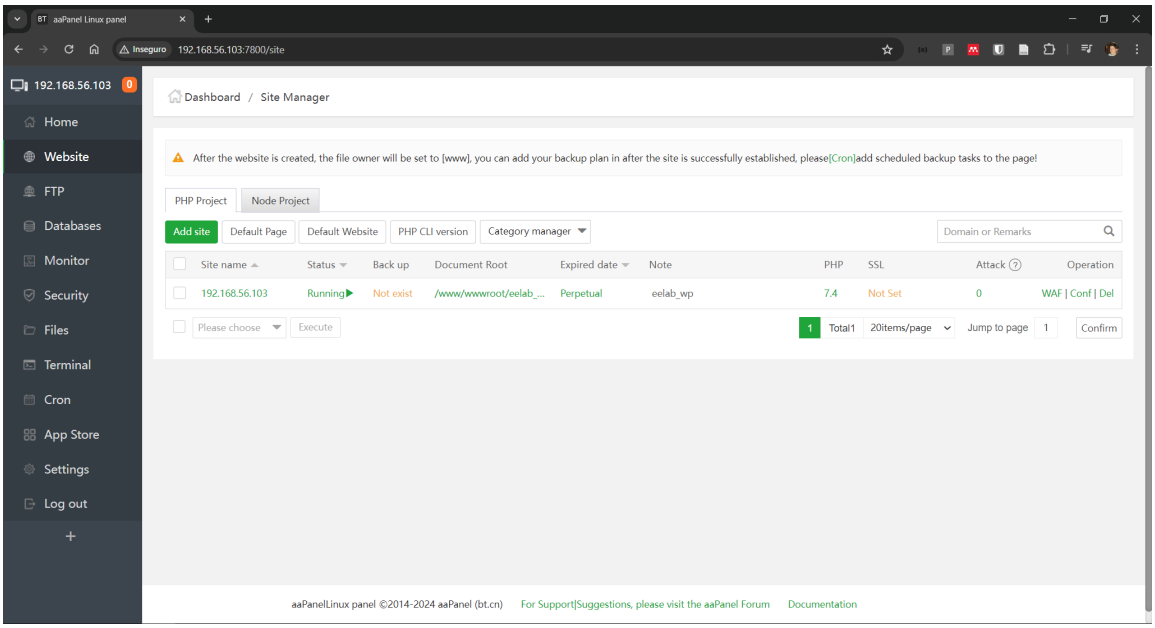


Figura 52. Exemplo dos testes realizados ao *aaPanel* (Menu '*Website*')

Em relação ao ***Webmin*** (Figura 53), a sua instalação foi bastante fácil e ficou a funcionar de imediato. Após a instalação, testei as várias funcionalidades e, apesar da interface gráfica ser apelativa e possuir muitas funcionalidades, não é tão simples como a interface do *aaPanel*, porque o menu lateral do *Webmin* tem mais opções e estas estão em menus colapsáveis. Ainda assim, as funcionalidades que foram testadas funcionam perfeitamente e estão bem organizadas no menu lateral.

Tendo em conta os testes que foram realizados, a plataforma escolhida para instalar no servidor do EELab foi o ***Webmin***. Apesar de ambos terem as funcionalidades necessárias, o *Webmin* destacou-se sobretudo na facilidade de instalação e configuração inicial e também pela sua fluidez. Além disso, na sua página inicial, o *Webmin* possui um sistema de notificações para quando há novas atualizações nos *packages* instalados no servidor, que também pode ser visto nas restantes páginas através do sino que está perto do canto superior direito da página, como mostra a Figura 53.

Uma vez escolhido o painel de controlo para o servidor, e antes de passar à sua instalação, foi feito um backup do mesmo para o caso de haver algum problema durante este processo. Feito o backup, procedi à instalação do *Webmin* no servidor do EELab seguindo o procedimento recomendado na sua documentação [45]. A instalação correu perfeitamente e tudo continuou a funcionar, incluindo o site do EELab. Quanto ao painel do *Webmin*, bastou aceder ao URL adequado e fazer *login* usando as credenciais de utilizador do *Ubuntu*. No entanto, ao aceder ao site do EELab a ligação aparecia como 'Insegura', sendo que deveria ser HTTPS. Aparentemente era um *bug* com a

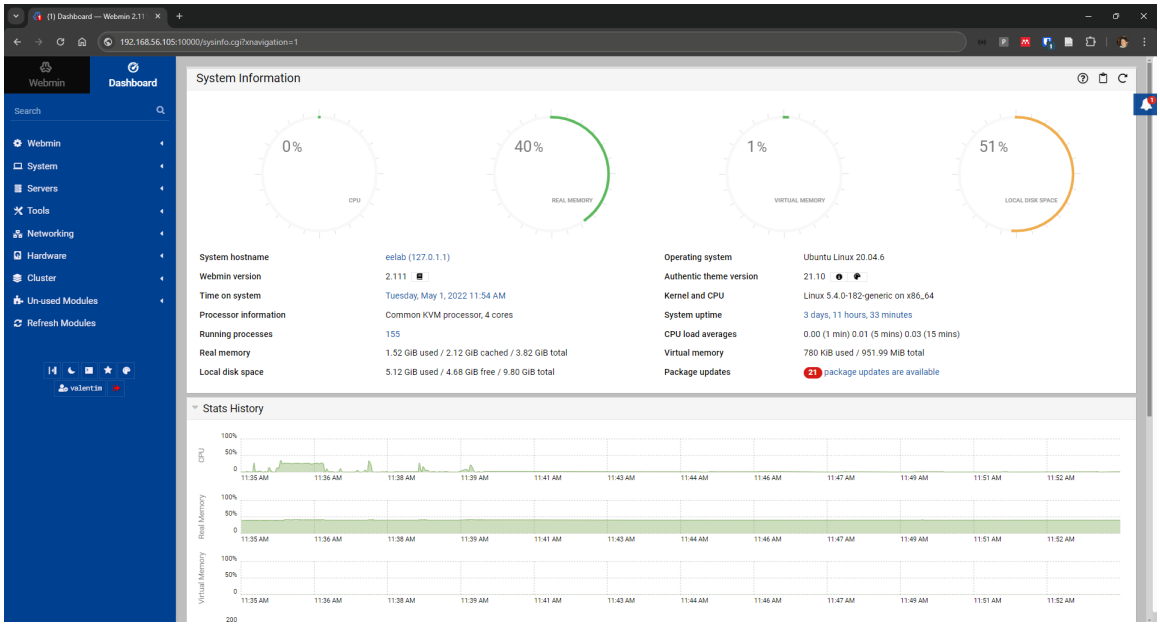


Figura 53. Página inicial do *Webmin*

cache do *browser*, até porque isto apenas acontecia ao aceder ao site do EELab depois que aceder ao painel do *Webmin* (que utiliza HTTP, ou seja, ligação considerada insegura).

Mais tarde, detetei um outro problema, apenas conseguia aceder ao *Webmin* dentro da rede Wi-Fi da ARDITI. Isto acontecia porque, por defeito o *Webmin* fica a correr no porto 10000, e este porto não estava aberto no *router* da ARDITI. A resolução deste problema passou por configurar um reverse proxy no *Apache*, que recebe os pedidos no porto 8080 e internamente realiza os pedidos a `http://localhost:10000`. Portanto, assim já é possível aceder ao painel mesmo fora da ARDITI através de `http://eelab.arditi.pt:8080/`, porque o porto 8080 está aberto.

Concluído este processo, foi então realizado o procedimento de *deploy* da aplicação do *GesFoGO*, seguindo o procedimento descrito na Secção 4.1. Além disso, foram instalados no Servidor do EELab, de forma nativa, os *softwares* necessários para o correto funcionamento de toda a aplicação do *GesFoGO*, nomeadamente o *Apache*, o *PHP*, o *MySQL* e o *Node.js*.

4.3 Integração com a Parte da ULPGC

A partir do momento em que as componentes do *GesFoGO* da responsabilidade da ARDITI e da ULPGC já estavam num estado avançado de desenvolvimento, isto é, já funcionavam individualmente apesar de ainda terem algumas arestas por limar e eventualmente alguns *bugs* por corrigir, e com a aproximação das datas previstas para a realização de testes no terreno, foi necessário proceder à integração das duas partes. A descrição e explicação das componentes que são responsabilidade da ARDITI e da ULPGC estão detalhadas na Secção 2.2. De uma forma simplificada, existem duas "vias"de comunicação entre as partes, realizadas através de pedidos HTTP usando o formato JSON:

– a API REST da ULPGC comunica com a API REST da ARDITI para envio de capturas dos sensores (linhas de fogo/sombra e imagens visíveis/térmicas) e atualizações regulares dos mesmos.

- a API REST da ARDITI comunica com a API REST da ULPGC para envio de comandos aos sensores.

No que diz respeito ao envio das capturas dos sensores, isto é, linhas de fogo/sombra, imagens visíveis/térmicas e também as suas atualizações regulares, para a API REST da ARDITI forneci aos desenvolvedores da ULPGC a documentação da API, bem como uma *API Key* para se conseguirem autenticar e testar os *endpoints* desenvolvidos. Além disso, expliquei de forma breve o funcionamento desses *endpoints*, isto é, em que situações cada *endpoint* deve ser usado. Como já foi mencionado anteriormente, os *endpoints* disponibilizados à ULPGC foram os do grupo '*Info Reception Endpoints*', explicado mais detalhadamente na Secção 3.4. Como a API estava acompanhada pela documentação, o processo de utilização por parte dos parceiros da ULPGC foi relativamente fácil.

Relativamente ao envio de comandos para os sensores, devido a alguns atrasos no desenvolvimento do hardware (sensores), a equipa de desenvolvimento da ULPGC não chegou a fazer documentação para a sua API, o que dificultou um pouco a utilização, nomeadamente porque não sabia bem como formar os pedidos HTTP, sobretudo o '*body*', porque tinha de saber o nome e o tipo de cada propriedade. Apesar disso, acabei por conseguir utilizar a API da ULPGC com sucesso.

Além de estabelecer a comunicação entre as API's da ARDITI e da ULPGC, foram também solicitados, por parte da ULPGC, uma série de pontos ao longo do mapa da Madeira para calibrar os sensores. Foi necessário introduzir esses pontos no mapa da GUI e, posteriormente, enviar uma captura de ecrã, com o máximo detalhe possível, dessas localizações. Foram solicitados 8 pontos/localizações:

- **Ponto 1:** 32.856243, -17.159021
- **Ponto 2:** 32.752350, -17.224297
- **Ponto 3:** 32.646096, -16.928653
- **Ponto 4:** 32.706778, -16.763612
- **Ponto 5:** 32.741147, -16.710450
- **Ponto 6:** 32.820046, -16.975533
- **Ponto 7:** 32.754314, -16.870746
- **Ponto 8:** 32.733057, -17.052804

O objetivo deste procedimento é garantir que as coordenadas obtidas pelos sensores correspondem às coordenadas mostradas no mapa. Na Figura 54, está exemplificada uma dessas capturas de ecrã, correspondente ao Ponto 5.

Após a integração de todos os componentes, fizemos vários testes de envio/receção de informações entre os vários componentes para garantir que tudo funcionava, de forma a ter tudo pronto para os testes no terreno realizados na Ilha da Madeira. Os detalhes desses testes são apresentados na Secção 4.4, em seguida.

Figura 54. Captura de ecrã para o Ponto 5

4.4 Testes no terreno

Os testes no terreno foram feitos em 3 fases, e cada uma delas teve lugar numa região diferente da Macaronésia. A primeira fase teve lugar na Ilha de Santiago, em Cabo Verde, entre os dias 20 e 22 de junho de 2022. Como o foco destes testes era apenas o Hardware, isto é, o próprio sensor, não participei. No entanto, estiveram envolvidos cerca de 40 participantes, entre eles comandantes e técnicos do Serviço Nacional de Proteção Civil e Bombeiros de Cabo Verde, dos bombeiros, da agricultura e do exército.

A segunda fase de testes decorreu na Ilha da Madeira entre os dias 2 e 3 de agosto de 2022. Foram realizados pela primeira vez os testes integrados de todo o sistema, incluindo os sensores, os servidores e a GUI. Os testes na Ilha da Madeira contaram com o envolvimento e contributo de equipas de especialistas de diferentes serviços de prevenção e combate a incêndios florestais, nomeadamente, o IFCN - Instituto das Florestas e Conservação da Natureza (Madeira), o Cabildo de Gran Canária (Canárias) e o Serviço Nacional de Proteção Civil e Bombeiros (Cabo Verde), contabilizando cerca de 25 participantes. Nas subsecções 4.4.1 e 4.4.2 são descritos detalhadamente como decorreram cada um dos dias de testes na Madeira e também a demonstração.

A terceira e última fase de testes foi em Gran Canária (Canárias) entre os dias 12 e 13 de setembro de 2022. Para este último teste já tinham sido aplicadas várias correções e melhorias com base na experiência adquirida dos testes anteriores e teve a participação de seis especialistas em prevenção e combate a incêndios, representando cada uma das três regiões envolvidas no projeto (Canárias, Madeira e Cabo Verde). Durante estes testes foi feita também a demonstração final do projeto *GesFoGO*, descrita na Secção 4.5.

4.4.1 Dia 02/08/2022 (Testes)

O primeiro dia de testes começou com a receção dos parceiros da ULPGC, IFCN e Cabildo de Gran Canaria na ARDITI. Em seguida, arrancamos para o parque de estacionamento do Teleférico do Jardim Botânico, o local escolhido para fazer a instalação do sensor e realizar os testes, como é possível observar na Figura 55.

Enquanto isso, um outro elemento foi para o outro lado do vale, onde foi colocada uma placa de vitrocerâmica para simular um ponto de elevada temperatura, simulando assim um incêndio, como

Figura 55. *Deploy* do Sensor no primeiro dia de testes

mostra a Figura 56. Nota: como estes testes foram realizados no mês de agosto, em que o risco de incêndio é elevado, não era permitido testar com fogo, mesmo que controlado e na presença de bombeiros.

Após a calibração dos parâmetros do sensor e a resolução de alguns problemas de energia (*powerbank* e/ou cabo USB que fornecia energia ao sensor) foram obtidas as primeiras capturas (polígonos e imagens). O primeiro problema que identificamos foi que estavam a ser detetados imensos 'focos' de incêndio que coincidiam com telhados de casas, carros ou outros objetos bastante quentes devido ao sol e não apenas o ponto de elevada temperatura simulado através da placa de vitrocerâmica. A resolução deste problema passou por ajustar o valor mínimo a partir do qual se considera um incêndio. Após alguma tentativa e erro, o valor foi aumentado para 6000, e passou a detetar corretamente. Na Figura 57, mostrada de seguida, está uma fotografia do software de controlo do sensor onde é configurado esse valor e onde se pode observar o ponto onde se encontra a placa vitrocerâmica. Nota: apesar deste valor ser diretamente proporcional à temperatura não corresponde à temperatura em graus mas a um índice de radiação detetada através das imagens térmicas.

Devido aos primeiros testes em que estavam a ser geradas imensas falsas deteções de incêndio, a BD acabou por receber cerca de 11000 capturas de linhas de fogo e cerca de 50 imagens visíveis/térmicas. Devido a esta elevada quantidade de dados, aliada à fraca conexão à Internet que tínhamos no local (*Hotspot* Móvel 4G), a GUI não estava a conseguir receber estes dados para mostrá-los. Apesar da API REST ter conseguido receber e armazenar todas as capturas do sensor, quando a GUI fazia o pedido para obter esses dados, quando a resposta continha demasiados dados esta nem era recebida.

Na parte da tarde, voltei à ARDITI para tentar resolver este problema das grandes quantidades de dados e assim que tentei fazer o pedido através da GUI este funcionou perfeitamente, conseguia

Figura 56. Placa de vitrocerâmica que simula um foco de incêndio

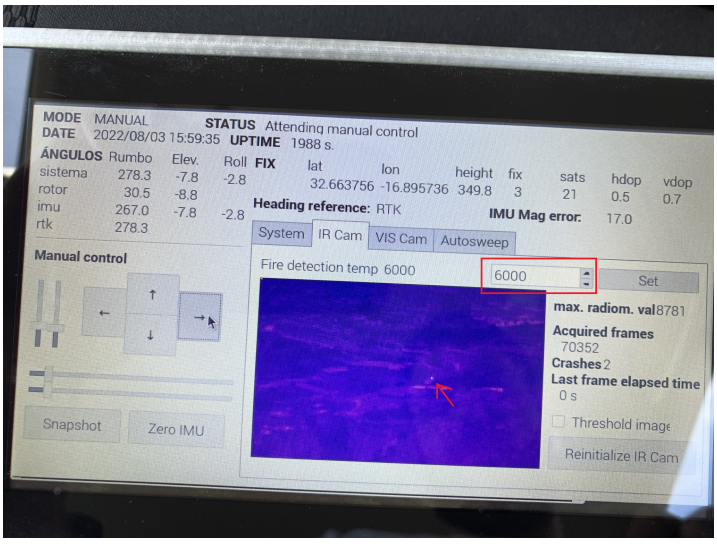


Figura 57. Software de controlo do sensor com a nova configuração

ver as linhas de fogo e as imagens capturadas na parte da manhã. Na Figura 58, em seguida, encontra-se uma captura de ecrã com um exemplo das amostras obtidas nos testes, onde assinalei com setas o sítio onde estava a placa vitrocerâmica no mapa do *MainVisor* e na imagem do *ImageVisor*.

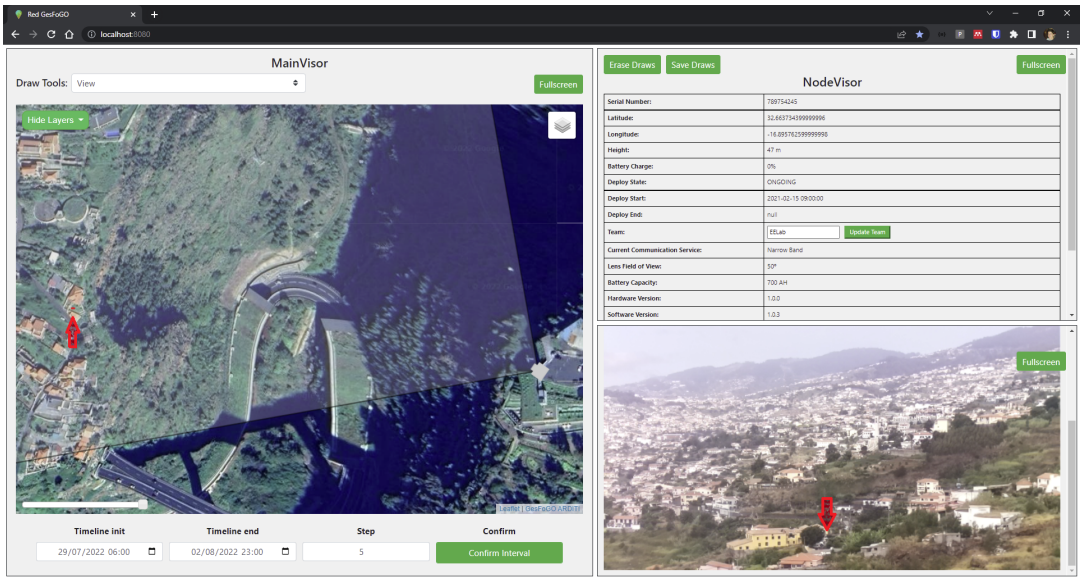


Figura 58. Captura de ecrã da GUI do *GesFoGO* com exemplo do primeiro dia de testes

A conclusão a que cheguei foi que a fraca conexão à Internet é que estava a impedir a receção destes dados pela GUI. Embora receber esta quantidade de dados seja uma situação excecional, acabei por fazer algumas alterações para a demonstração às entidades oficiais, no dia seguinte, e correção de alguns problemas que detetei. A primeira alteração foi feita ao *endpoint* '*Get Images*' que antes enviava todas as imagens do *deploy* do sensor selecionado, e passei a restringir ao intervalo de tempo selecionado na *Timeline*. Além disso, detetei um *bug* relativo às datas inicial e final do intervalo de tempo da *Timeline*, que davam problemas quando ambas as datas tinham o mesmo dia mas horas diferentes. O problema acontecia na API REST nos *endpoints* '*Get Images*' e '*Get Polygons*', porque estavam a utilizar as funções do *Query Builder* do *Laravel*, e a função *whereBetween()* não estava a ter o comportamento esperado, ou seja, não estava a retornar apenas as amostras dentro do intervalo de tempo especificado na *Timeline*. Para resolver este *bug* a solução foi utilizar a função *DB::raw()* e passar como argumento a *query* SQL completa (com as devidas variáveis), como pode ser verificado na Figura 14.

Após estas alterações/correções a aplicação do *GesFoGO* já estava pronta para a demonstração às entidades oficiais no dia seguinte.

4.4.2 Dia 03/08/2022 (Demonstração)

No segundo dia começamos por levar alguns equipamentos para o local onde vai ser feita a demonstração do *GesFoGO* às entidades oficiais, nas instalações da AJAMPS - Associação dos Jovens Agricultores da Madeira e Porto Santo, que fica mesmo acima do parque de estacionamento do Teleférico do Jardim Botânico, onde está instalado o sensor. Aqui já tinha uma melhor conexão à *internet* e confirmei que estava tudo a funcionar. Além disso, como o nosso foco de incêndio

era bastante pequeno e difícil de visualizar no mapa, foi sugerido adicionar *markers* para ajudar na demonstração. Para isso, bastou utilizar a função *L.marker()* do *Leaflet*. O resultado pode ser observado na Figura 59.

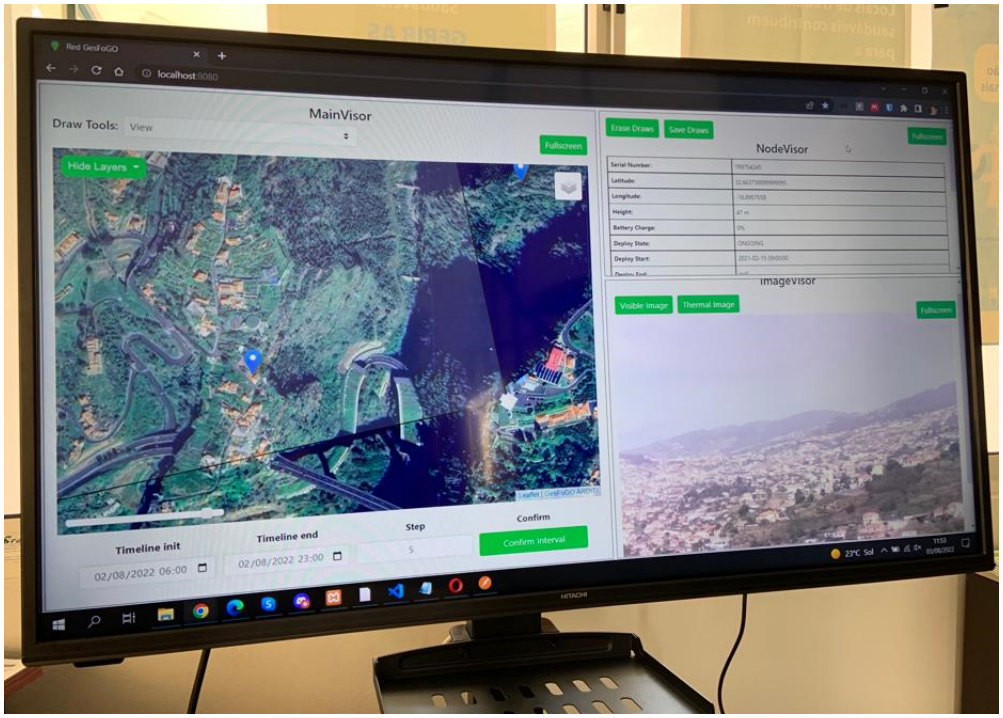


Figura 59. GUI do *GesFoGO* com adição dos *markers*

Na parte da tarde, voltamos para o local da demonstração onde estiveram presentes vários elementos do IFCN e também o presidente da ARDITI, Rui Caldeira. Primeiramente, foi feita a demonstração do sensor no parque de estacionamento do Teleférico do Jardim Botânico, e depois dirigiram-se para a AJAMPS para eu demonstrar a GUI, mostrando as capturas acabadas de obter em tempo real e também para esclarecer algumas dúvidas, como mostra a Figura 60.

Em suma, durante estes testes, o sistema foi demonstrado e testado por vários técnicos de gestão florestal do Governo Regional da Madeira. O feedback recebido foi bastante positivo, os técnicos acharam o sistema adequado à finalidade e consideram que atende a todas as suas necessidades. Com a experiência obtida nestes dias e com alguns problemas que detetamos foram feitas algumas melhorias e correções para ter todo o sistema a funcionar corretamente antes dos últimos testes e demonstração final em Gran Canária.

4.5 Demonstração Final

A conclusão oficial do projeto *GesFoGO* teve lugar em Gran Canária (Canárias) entre os dias 12 e 13 de setembro de 2022, onde foram realizados os últimos testes no terreno e foi feita a sua demonstração final, que contaram com a presença seis especialistas em prevenção e combate a incêndios, representando cada uma das três regiões envolvidas no projeto (Canárias, Madeira e Cabo Verde). Apesar de eu não ter estado presencialmente nestes testes, estive a dar suporte remotamente para garantir que toda a informação que estava a ser recebida na API REST estava correta e que a GUI estava a mostrar essa informação adequadamente.

Figura 60. Fotografia da demonstração da GUI do *GesFoGO* às entidades oficiais

Durante o período entre os testes feitos na Madeira e esta demonstração foram feitas algumas correções e melhorias com base na experiência que foi obtida. Apesar dos principais erros terem sido corrigidos já durante os testes na Madeira, algumas dessas correções foram melhoradas, visto que, tinham sido feitas de forma rápida/provisória para fazer a demonstração na Madeira. Por exemplo, alterei a função que adiciona os *markers* aos polígonos para facilitar a visualização apenas quando os focos de incêndio são muito pequenos. Usei a função *L.GeometryUtil.geodesicArea()* do *Leaflet* que permite calcular a área, em metros quadrados, de cada polígono para decidir quando deve ser adicionado um *marker*. No caso, é adicionado quando a área é inferior a 200 metros quadrados. Também me apercebi que, quando era criado um novo *deploy* devido ao sensor ter mudado de lugar, este ficava sem configuração associada. Para resolver esta situação, o *endpoint Update Sensor Deploy*, passou a ter a responsabilidade de replicar as configurações do *deploy* antigo para o novo (quando é criado um novo *deploy*).

Nestes testes finais, foram usados 2 sensores, como é mostrado na Figura 61, e para além de um foco de incêndio simulado através de uma placa vitrocerâmica como foi feito nos testes da Madeira (Figura 56), foi também usando um foco de incêndio real usando um fogareiro, como é possível verificar na Figura 62.

Durante a demonstração, o fogareiro que se encontra na Figura 62 foi mudado de lugar algumas vezes para comprovar o comportamento do sistema conforme um incêndio progride.

O resultado destes testes pode ser visualizado na Figura 63, uma captura de ecrã da GUI com as capturas dos sensores. No *MainVisor* estão representados os 2 sensores que foram instalados neste demonstração, bem como as respetivas linhas de fogo destacadas através de *markers*. No *ImageVisor* encontra-se uma fotografia do sensor que está mais à direita no mapa onde é visível

Figura 61. *Deploy* de 2 sensores para demonstração em Gran Canária

Figura 62. Fogareiro usado para simular um incêndio

o fogareiro. Perto do canto superior esquerdo do mapa é apresentada a captura correspondente à placa vitrocerâmica, obtida através do outro sensor.

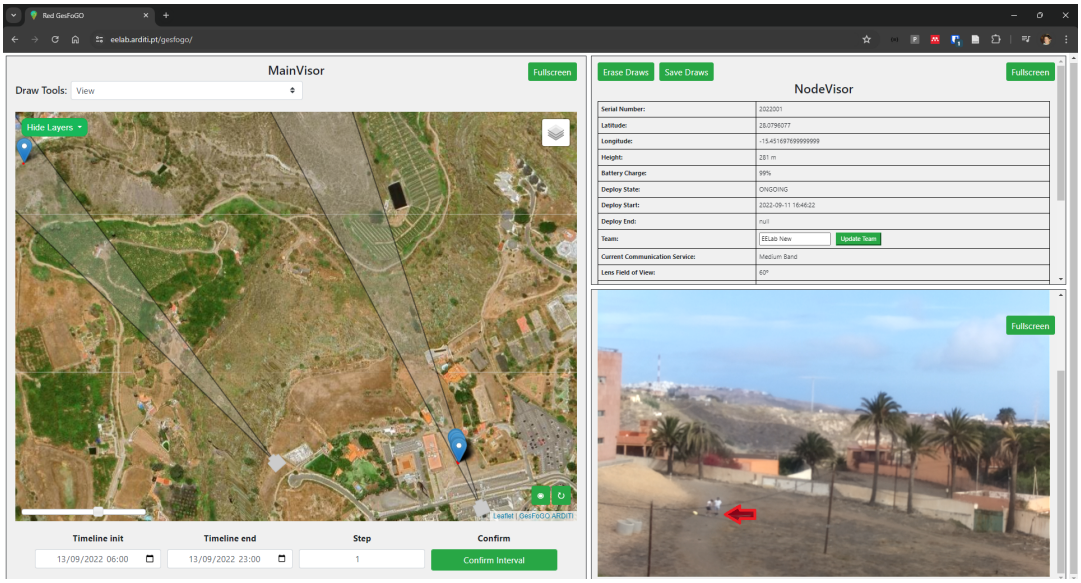


Figura 63. Captura de ecrã da GUI do *GesFoGO* com o resultado da demonstração

Para concluir, estes últimos testes e a demonstração final correram bastante bem, não tivemos problemas no funcionamento do sistema. A única intervenção que tive de fazer foi remover algumas capturas da BD que continham informação que não era relevante, obtida quando os sensores foram ligados. Os testes incluíram a deteção de ignição de incêndios florestais pelo sistema, que foram analisados pelos especialistas em prevenção e combate a incêndios quanto à sua abrangência e completude dos dados recolhidos e das funcionalidades da GUI. O feedback que recebemos do funcionamento do sistema foi muito positivo e os especialistas não identificaram nenhuma informação em falta, por isso consideramos que o projeto foi concluído com sucesso.

5 REST na Plataforma DISME

Este capítulo é dedicado à integração de mecanismos de API REST para criação e gestão de *endpoints* na plataforma *low-code Dynamic Information Systems Modeller and Executer* (DISME). Inicialmente é feita uma contextualização onde é apresentada a motivação para esta nova funcionalidade/componente e é feita uma breve introdução do DISME. Depois, são apresentados os modelos DEMO mais relevantes para este caso específico. Em seguida, é feita a revisão de literatura para analisar a investigação já existente na temática de criação e gestão automática/rápida de APIs REST. No final, é então apresentada a nova abordagem proposta para integração desta funcionalidade no DISME, detalhando os vários tipos de *endpoints* e as adições necessárias a fazer à gramática EBNF existente para as regras de ação do DISME.

5.1 Contexto

Esta secção serve como uma breve introdução ao contexto geral do projeto DISME, fornecendo a motivação para a realização desta atividade de investigação, os objetivos definidos e uma breve apresentação da plataforma DISME.

5.1.1 Motivação

A utilização de APIs REST como serviço de *backend* é uma abordagem cada vez mais popular para a gestão de dados nas organizações. As APIs [10] (*Application Programming Interface*) fornecem acesso programático a serviços e/ou dados dentro de uma aplicação ou base de dados. REST [11] (*Representational State Transfer*) é uma abordagem às comunicações frequentemente utilizada no desenvolvimento de serviços Web, que expõe a sua informação através de representações sem estado dos seus recursos, permitindo interoperabilidade entre diferentes tipos de clientes.

O desenvolvimento do *software* de *backend* é uma tarefa exigente que pode consumir bastante tempo, devido à necessidade de lidar com a integridade, confidencialidade, disponibilidade e privacidade dos dados, bem como gerir adequadamente centenas de pedidos/tarefas em simultâneo. Além disso, a implementação destes serviços requer um amplo conhecimento e qualificações em programação. Este trabalho de investigação tem como foco principal apresentar o desafio de criar e gerir interfaces REST de entrada e de saída de forma rápida e automática, recorrendo à abordagem de modelação DEMO [4], viabilizada através de uma *Low-Code Platform* (LCP), especificamente o DISME. O objetivo é permitir que gestores e funcionários com papéis semelhantes nas organizações sejam capazes de criar e gerir *endpoints* de API REST, eventualmente sem a necessidade de competências em programação.

As LCPs oferecem uma solução viável para a geração rápida e automática de APIs, abrangendo leitura e escrita de dados e também a realização de ações específicas. Ao aproveitar os modelos de dados num sistema de informação *low-code*, a criação de *endpoints* – que vão desde listagens de itens básicas a resultados de *queries* complexas – torna-se acessível através de operações de *drag-and-drop* numa interface gráfica intuitiva. Esta plataforma não só permite a execução de tarefas internas com base em pedidos externos ao sistema, como também facilita a obtenção e análise de dados de APIs externas, alinhando-os com dados e regras internas.

5.1.2 Objetivos

Um dos componentes do DISME utiliza o *Blockly*[61] que, seguindo uma gramática formal *Extended Backus-Naur Form* (EBNF), permite aos utilizadores configurar facilmente regras de negócio para implementar a lógica de negócio interna. O *Blockly* é uma biblioteca que adiciona um editor de código visual a aplicações web e móveis. O editor do *Blockly* utiliza blocos gráficos interligados para representar conceitos de código, como variáveis, expressões lógicas, ciclos e muito mais. Permite aos utilizadores aplicar princípios de programação sem se preocuparem com a complexidade da sintaxe.

Após ter colaborado no desenvolvimento dos vários projetos mencionados no Capítulo 1, nomeadamente no que diz respeito às APIs REST, adquiri bastante conhecimento e competências acerca do tema. O objetivo é aproveitar essa experiência adquirida, acerca do que está envolvido no desenvolvimento de uma API REST para uma determinada aplicação/sistema, para enriquecer o DISME com os mecanismos de criação e gestão de APIs REST, que designamos por *Rapid REST API Management* (RRAM). Isto é, desenvolver um novo componente do DISME que permite essa gestão de interfaces de API para *input* e *output* de dados e realização de ações através de uma interface gráfica intuitiva. Adicionalmente, as configurações e formatos/atributos dos dados também podem ser gerados de forma (semi-)automática, bem como a sua documentação que pode ser gerada automaticamente utilizando ferramentas como o *Swagger UI*[62]. Esta abordagem permite, não só facilitar a que pessoas sem conhecimentos técnicos de programação, apenas com alguma formação básica tenham a capacidade de criar *endpoints* de uma API, mas também para programadores torna-se mais rápido do que criar uma API de raiz.

5.1.3 *Dynamic Information Systems Modeller and Executer*

A plataforma *low-code opensource Dynamic Information Systems Modeller and Executer* [6] (DISME), que está a ser desenvolvida pela equipa do EELab, é uma ferramenta que tem como base a metodologia *Design and Engineering Methodology for Organizations* [4] (DEMO). Esta solução está a ser desenvolvida seguindo princípios e tendências informáticas inovadores como *Software as a Service* (SaaS), *Direct Model Execution*, *Model-Driven Software Engineering* e *Automatic Code Generation* que visam tornar as organizações cada vez mais independentes de programadores ou empresas de *software*. Permite a representação de modelos de processos, *workflows*, responsabilidades, interfaces, factos e ontologia de informação, papéis, funções organizacionais e os agentes associados. Estes modelos podem ser executados diretamente como um sistema de informação com o fluxo de trabalho inerente, sendo executado dinamicamente e as notificações enviadas automaticamente aos utilizadores [6]. O DISME é composto por 2 componentes principais:

1. ***System Modeler*** - um vasto conjunto de ferramentas dentro do DISME, que permite aos utilizadores descrever artefactos organizacionais de forma eficaz. Inclui:
   - ***Diagram Editor*** - permite a representação visual dos processos através de modelos DEMO, automatizando as entradas na base de dados e melhorando o *design*.
   - ***System Specifier*** - simplifica a gestão de elementos dos modelos, funções, processos, transações e tipos de entidades, retirando a necessidade de competências de programação.

[61] Blockly - https://developers.google.com/blockly
[62] Swagger UI - `https://swagger.io/tools/swagger-ui`

- ***Action Rules Management*** - utiliza inglês estruturado e o *Blockly* para especificar *inputs* e lógica de processos, simplificando a criação de regras de negócio complexas sem muito conhecimento de programação.
- ***Forms Management*** - integra o *Form.io*[63] para a criação de formulários, garantindo um alinhamento rigoroso entre os *inputs* especificados e os formulários preenchidos pelo utilizador.
- ***Dynamic Query Management*** - componente que permite aos utilizadores criar *queries* facilmente através de uma interface gráfica, utilizando operadores propriedade-operador-valor. Esta funcionalidade permite a obtenção de dados de forma dinâmica e flexível, contendo diversas necessidades de informação.

2. ***System Executor*** - facilita a execução direta do sistema de informação modelado em modo de produção. Consiste no *Dashboard*, que fornece a interface do utilizador para as tarefas organizacionais, e no *Execution Engine*, garantindo que a informação e o fluxo do processo estão alinhados com as especificações do sistema.

5.2 Modelos DEMO

Com base nos conceitos fundamentais da metodologia DEMO - *Design and Engineering Methodology for Organizations* [4], a abordagem inovadora proposta para a geração automática de APIs baseia-se nos avanços mais recentes dos modelos DEMO. Estes avanços, detalhados em [6,7,46–52], representam uma mudança de paradigma na sua abordagem *user-friendly* quando comparada com os métodos tradicionais [4]. Nas subsecções seguintes são explorados, em detalhe, os componentes principais:

5.2.1 Modelo de Factos

A peça central do DEMO é o *Fact Model* (FM) [48], que visa representar os produtos e serviços de uma organização. O FM define o estado de uma organização e os espaços de transição dentro do mundo de produção [4]. Um artefacto relevante dentro do FM é o *Fact Diagram* (FD), que fornece *insights* através de duas perspetivas: o *Concept and Relationships Diagram* (CRD) e o *Concept Attribute Diagram* (CAD).

No caso do CRD, são utilizadas setas para retratar relações que, na prática, consistem no atributo de um conceito cujas instâncias vão representar uma referência a instâncias de um outro conceito. Na Figura 64 é apresentado um exemplo do CRD, mostrando os modelos desenvolvidos para um processo de licenciamento de construção [48]. Esta representação visual mostra de forma evidente as dependências entre conceitos. Notavelmente, os tipos de factos binários são categorizados em três relações distintas, cada uma distinguida por símbolos específicos:

1. um-para-um (*one-to-one*) é representada por um conector com dois símbolos de seta no meio (Nota: este exemplo não existe na Figura 64);
2. muitos-para-um (*many-to-one*) é representada por um conector com apenas uma seta a apontar para o lado "um" da relação;

[63] Form.io - `https://form.io/`

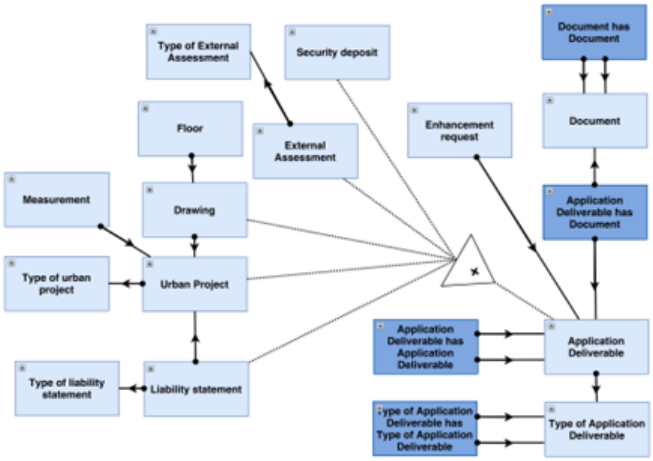


Figura 64. *Concept and Relationships Diagram* [48]

3. muitos-para-muitos (*many-to-many*) é representada com um conceito intermédio retratado com uma cor mais escura; este conceito possui relações muitos-para-um com os conceitos que participam nesta relação de muitos-para-muitos; ambas as relações seguirão critérios de dependência no lado deste conceito intermédio.

O CAD, uma extensão do CRD, descreve de forma mais pormenorizada os atributos dos conceitos. Cada conceito é contido numa caixa colapsável, detalhando os seus atributos e os tipos de valor correspondentes. Esta análise detalhada melhora a compreensão das propriedades dos conceitos [48] [46].

5.2.2 Modelo de Ação

O *Action Model* (AM) [47] do DEMO compreende as operações da organização, guiando os atores através de *coordination acts* de transações. As regras de ação são especificadas para orientar os atores nas suas funções. O que diferencia a metodologia DEMO é a sua adesão à racionalidade comunicativa, que proporciona aos atores a flexibilidade para adaptar o seu comportamento com base no seu entendimento e experiência profissional.

A evolução da linguagem das *Action Rule Specification* (ARS) [51] tem sido um fator chave para garantir clareza e precisão. A atual estrutura de uma ARS é composta por três partes: evento (ato de coordenação que despoleta a regra), a avaliação (validação de condições de validade) e resposta (instruções de ação, incluindo possíveis chamadas a outras transações/regras), o que traz complexidade desnecessária e falta de precisão. Em [51] foi proposta uma linguagem ARS alternativa, formalizada com a gramática EBNF. Na Figura 65, é apresentado um exemplo de uma regra de ação no *Blockly*. Esta nova abordagem integra expressões, condições lógicas, validações, formulários de *input* e *outputs* para o utilizador de documentos baseados em *templates*. Ao simplificar a sintaxe, é melhorada a aplicação prática e as funcionalidades necessárias para a abordagem de RRAM proposta.

5.3 Revisão da Literatura

Nesta secção, é feita a revisão e análise da investigação já existente relacionada com a geração, gestão e documentação de APIs REST, de forma automática.

Figura 65. Regra de Ação para o pedido (*request*) de uma transação "Rental Contracting"

Alguns investigadores seguiram uma abordagem de geração de código. Wang et al. [53] introduziu um método baseado em modelos para gerar código para aceder a bases de dados, que depois foi encapsulado em APIs RESTful. Embora esta abordagem minimize os esforços e melhore a flexibilidade e a reutilização, também apresenta diversas desvantagens, tais como esforços iniciais significativos, rigidez do código e aumento da complexidade técnica.

Outros estudos defendem a *Model-driven Engineering* (MDE) [54] [55] como um processo de desenvolvimento de software iterativo e incremental. Mora-Segura et al. [54] propôs uma solução baseada na modelação multi-nível representando o conhecimento de domínio. Suporta o carregamento de dados sob pedido através de *domain injectors*, descritos através de *queries* semanticamente ricas, que recebem dados num determinado formato, de uma determinada plataforma e produzem um modelo em conformidade com o *domain metamodel* [54].

Hussein et al. [56] desenvolveu a *REST API Automatic Generation* (RAAG), uma *framework* integrada que abstrai camadas para APIs REST, lógica de negócio, acesso a dados e operações de modelo. O RAAG, desenhado para reutilização, facilidade de manutenção, escalabilidade e desempenho, reduziu significativamente o tempo de desenvolvimento em comparação com implementações tradicionais de APIs REST, como verificado por uma avaliação preliminar. Os utilizadores, incluindo programadores não experientes, consideraram o RAAG intuitivo, fácil de manter e produtivo, destacando a sua facilidade de utilização e eficiência.

Overeem et al. [57] avaliou a maturidade da gestão de APIs nas principais *Low-code Development Platforms* (LCDPs). Surgiram desafios em tornar as APIs acessíveis a utilizadores inexperientes devido à simplificação inerente dos LCDPs, em conflito com a complexidade das soluções de software. Encontrar um equilíbrio entre simplicidade e funcionalidade é crucial.

Brajesh De [58] destacou a importância de uma documentação da API *user-friendly* para a adoção/utilização de APIs REST. A documentação deve oferecer uma interface de fácil compreensão, permitindo que os programadores, independentemente da sua experiência, compreendam as funcionalidades da API e comecem a utilizá-la sem complicações.

Adicionalmente, em [59], foi proposto um algoritmo para a geração de micro-serviços em conformidade com o standard *OpenAPI* a partir de um modelo ontológico DEMO. No entanto, esta abordagem gera serviços para todos os atos de transação, o que é excessivo. Além disso, foram encontradas limitações de implementação, como a definição de tipos de valor e operações de *DELETE*. Ao abordar estas questões observou-se que, embora ontologicamente os factos não possam ser alterados ou eliminados, as implementações práticas exigem tais capacidades, considerando regulamentos como o Regulamento Geral sobre a Proteção de Dados (RGPD) europeu. A solução proposta nas secções seguintes visa superar essas limitações.

Apesar dos vários estudos sobre a geração e gestão de APIs REST, alguns métodos dependem de ferramentas complexas ou requerem experiência específica, o que limita a sua acessibilidade. Alguns não dão suporte a várias bases de dados, apenas permitem operações de leitura de dados, enquanto outros se concentram principalmente na geração de código [53]. No entanto, é evidente, a partir dos artigos analisados [53] [55] que a geração (semi-)automática de APIs REST é uma opção viável. Corretamente executada, esta abordagem simplifica o processo, permitindo a participação de pessoas com experiência limitada em programação. Igualmente importante é o foco na criação de uma documentação amigável para a API, garantindo que tanto os utilizadores experientes como os inexperientes possam compreender e utilizar a API de forma eficaz.

5.4 *Rapid REST API Management*

A abordagem *Rapid REST API Management* (RRAM) agrega o desenvolvimento do DISME e a metodologia DEMO para gerar *endpoints* tanto acessíveis dentro do sistema local como externamente. O objetivo é integrar a geração e gestão rápida e/ou semiautomática de APIs REST no DISME, um sistema em evolução. A RRAM está a ser implementada com base em funcionalidades existentes, como a *Action Rule specification* (ARS) [51] que são instruções/regras de negócio para a gestão de eventos com base nas quais os atores devem agir e a modelação de *queries* complexas, desenvolvida através do *Blockly* e do *jQuery QueryBuilder*[64], que proporciona uma base bastante sólida para esta funcionalidade. O facto da extensão destes componentes utilizar informação existente, como factos de negócio, atributos, regras de ação e *queries*, irá complementar a criação deste componente dedicado a modelar e especificar os *endpoints* da API REST.

Para dar resposta a este desafio de investigação, são descritas as quatro funcionalidades principais necessárias para uma boa integração com outros sistemas através de APIs REST. As funcionalidades são as seguintes:

– **Operações CRUD simples** - envolve operações de *Create*, *Read*, *Update*, e *Delete* em dados locais do sistema DISME. Estas operações são facilitadas através de interfaces intuitivas, permitindo ao utilizador especificar os dados que pretende expor através da API. A complexidade destas operações é reduzida, uma vez que cada operação envolve um único conceito do sistema.

[64] jQuery QueryBuilder - `https://querybuilder.js.org/`

- **Fornecimento de dados baseado em *queries*** - o DISME facilita o fornecimento de dados baseado em *queries*, permitindo que os utilizadores obtenham conjuntos de dados específicos com base nas suas necessidades.
- **Integração da execução interna com chamadas a *endpoints* externos** - esta funcionalidade envolve o alinhamento entre regras de ação internas e os dados locais correspondentes com os dados obtidos de sistemas externos.
- **Ativação da execução interna com chamadas de *endpoints* locais** - o DISME combina os *endpoints* locais disponibilizados para sistemas externos com as regras de ação internas, garantindo uma integração consistente.

Os detalhes dos *endpoints* da API, incluindo os métodos HTTP, parâmetros e campos das respostas, são inferidos a partir das suas especificações dentro do sistema. Esta informação não só especifica o *endpoint*, como também permite gerar automaticamente a documentação da API, facilitada pelo *Swagger UI*.

Nas 4 subsecções que se seguem são explicadas mais a fundo cada uma destas funcionalidades. Importa mencionar que apenas a "Ativação da execução interna com chamadas de *endpoints* locais"foi implementada por mim, as restantes foram implementadas por outros colegas da equipa do EELab. No entanto, a grande maioria da escrita dos *papers* sobre este tema foi feita por mim.

5.4.1 Operações CRUD simples

Esta funcionalidade pretende dar suporte à criação de *endpoints* para operações CRUD simples, isto é, *Create*, *Read*, *Update*, e *Delete* para cada um dos conceitos/tipos de entidades internos do DISME que pretendemos expor através da API. Gerar *endpoints* para estas operações é simples, porque envolvem apenas um conceito/tipo de entidade do sistema, o que reduz consideravelmente a sua complexidade. O DISME fornece uma interface que permite aos utilizadores selecionar que informação pretendem expor através da API para as operações CRUD, incluindo as propriedades/atributos específicos dos tipos de entidade, ou seja, é possível selecionar todas ou apenas um subconjunto das propriedades/atributos de um tipo de entidade para uma determinada operação CRUD. Estas seleções são armazenadas nas tabelas da BD do DISME, gerando automaticamente os *endpoints* correspondentes.

5.4.2 Fornecimento de dados baseado em *queries*

Em cenários em que as operações simples de CRUD não são suficientes, o DISME oferece outra solução, a associação de *queries* complexas a *endpoints* específicos. Desenvolvemos uma interface intuitiva que permite aos utilizadores configurar consultas complexas quase sem esforço. Este componente opera com base na especificação de *queries* e dos respetivos filtros através de operadores propriedade-operador-valor, todos selecionados através de ações de *drag-and-drop* numa interface gráfica *user-friendly*, sem a necessidade de qualquer conhecimento técnico em programação.

No primeiro passo, os utilizadores selecionam o(s) tipo(s) de entidade pretendidos para a pesquisa, como se pode ver na Figura 66, com a seleção inicial a servir como Tabela Base — o tipo de entidade onde ocorre a pesquisa com base nos filtros definidos.

O segundo passo envolve a definição das propriedades a incluir na *query*. Para cada tipo de entidade selecionado, os utilizadores podem especificar as propriedades a incluir na resposta, bem como as propriedades a utilizar nos filtros, como se pode observar na Figura 67.

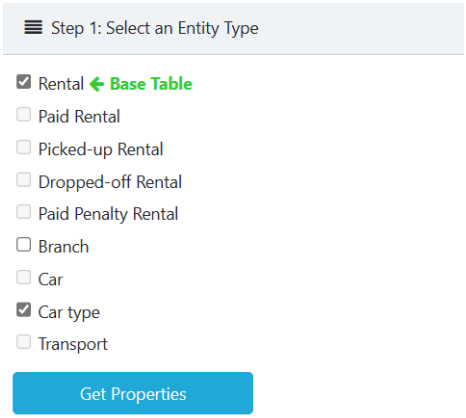


Figura 66. Interface para definição de *queries* - Passo 1 - Selecionar Tipo(s) de Entidade

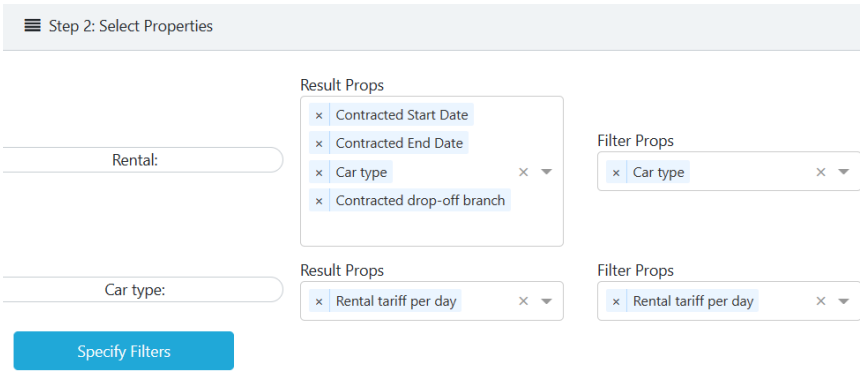


Figura 67. Interface para definição de *queries* - Passo 2 - Selecionar as Propriedades

Estas propriedades selecionadas constituem a base para a especificação dos operadores propriedade-operador-valor no passo seguinte. A partir daí, os utilizadores definem as regras e que conjuntos de regras são aplicadas à *query* principal, semelhantes a condições e sub-condições. O utilizador escolhe o tipo de regra (AND/OR), a propriedade a filtrar, o operador da *query* e o valor que restringe o resultado, como é ilustrado na Figura 68.

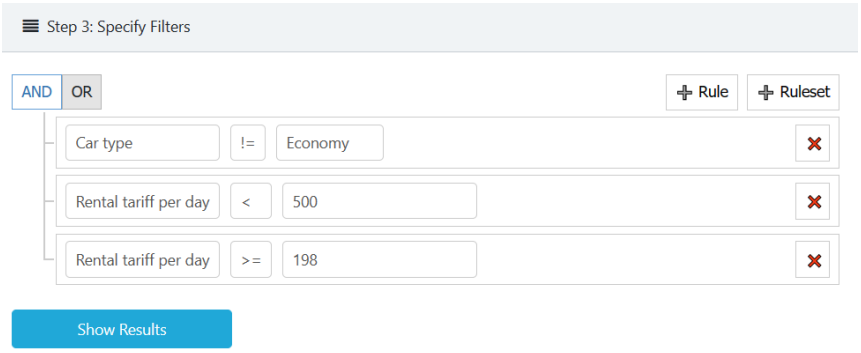


Figura 68. Interface para definição de *queries* - Passo 3 - Especificar os Filtros

Como podemos ver na Figura 69, as *queries* são configuradas com propriedades de resposta especificadas de forma clara, o que facilita a configuração dos *endpoints* da API cujas respostas correspondem aos resultados da *query*. Os utilizadores podem visualizar a lista de propriedades relacionadas com a *query* e selecionar aquelas que consideram relevantes para incluir na resposta da API. Isto também se aplica aos filtros, ou seja, os utilizadores podem especificar um ou mais filtros como parâmetros da *query*, deixando os valores para atribuir em tempo de execução - por exemplo, quando um *endpoint* é chamado com valores de parâmetros específicos.

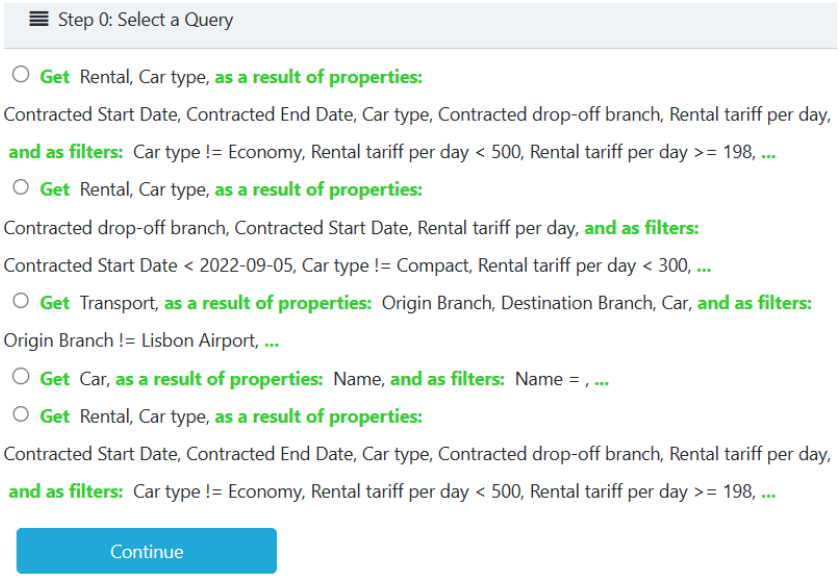


Figura 69. *Queries* configuradas

É importante salientar que a BD do DISME utiliza tabelas com nomes: tipo de entidade (*entity type*) e propriedade (*property*) para especificar o modelo de dados da organização, mas na interface de utilizador do DISME utilizamos os nomes: conceito (*concept*) e atributo (*attribute*), do metamodelo FM do DEMO adaptado, correspondentes a tipo de entidade e propriedade, respetivamente.

5.4.3 Integração da execução interna com chamadas a *endpoints* externos

Para a integração de execução interna com a chamada a *endpoints* externos, foi necessário realizar uma série de tarefas, começando com a criação das novas ações internas. Isto foi conseguido através da adição de vários elementos novos à gramática e respetivos blocos ao *Blockly* e à sua estrutura de suporte, nomeadamente:

- **Mostrar Resultado (*Show Result*)** - uma ação desenvolvida para mostrar os resultados das chamadas a APIs externas aos utilizadores através de uma interface dedicada. Isto envolveu a criação de um novo tipo de ação no DISME, que permite aos utilizadores visualizar os dados obtidos a partir de sistemas externos.
- **Criar Entidade(s) (*Create Entity(ies)*)** - foi implementado um outro tipo de ação para importar valores para o DISME provenientes de sistemas externos. Isto exigiu o mapeamento dos dados externos para entidades e propriedades internas correspondentes, garantindo assim a integridade dos dados no DISME.
- **Criar Entidade(s) Temporária(s) (*Create Temporary Entity(ies)*)** - foi desenvolvida uma ação especializada para realizar chamadas externas e armazenar dados temporários no DISME para input do utilizador durante a execução das regras de ação.
- **Correspondência (*Matching*)** - componente que permite ao utilizador criar uma correspondência entre as propriedades da entidade selecionada no bloco *create entity(ies)* e as propriedades existentes no conteúdo do resultado da chamada à API.
- **Correspondência Temporária (*Temporary Matching*)** - um componente semelhante ao *matching*, mas com a responsabilidade de fazer a correspondência entre as propriedades do sistema e as propriedades retornadas pela API REST.

A segunda tarefa foi modificar o componente REST API, para acomodar as novas necessidades, nomeadamente criando o *API Request Service* para facilitar as chamadas às APIs externas. Este serviço permite a configuração de parâmetros comuns como *headers*, URL, conteúdo do *body* do pedido e tipo de pedido (*POST*, *GET*, *PUT* e *DELETE*). Esta abstração reduziu a duplicação de código e melhorou a modularidade do sistema. Além disso, acomoda a *Integration Logic*, garantindo que o componente de API REST foi devidamente integrado na lógica interna do DISME. Isto envolveu a criação de uma forma de comunicação entre o *backend* do DISME e os sistemas externos.

Para finalizar a segunda tarefa, o tratamento das respostas externas, foi projetado um componente para lidar com as respostas dos sistemas externos, que podem estar no formato JSON ou XML. O DISME processa estas respostas, permitindo que os utilizadores interajam com os dados obtidos.

A última tarefa foi a gestão dos dados, nomeadamente a criação de um armazenamento temporário de dados para tratar os dados temporários dos sistemas externos. Para tal, foi delineada uma estratégia para armazenar estes dados em tabelas temporárias na BD do DISME. Isto garantiu

que os dados externos pudessem ser utilizados no contexto DISME sem comprometer a integridade global do sistema.

A Geração de Formulários e o *Input* do Utilizador permitem que os formulários DISME sejam gerados dinamicamente para o *input* do utilizador com base em dados temporários. Estes formulários permitem aos utilizadores interagir e validar dados externos antes de os incorporar na estrutura interna do DISME.

Por fim, o mapeamento e a correspondência são especificados implementando mecanismos para mapear as propriedades externas às propriedades internas, permitindo a correta integração com os dados temporários. Este mapeamento garante que os dados temporários são utilizados de forma eficaz nas entidades e propriedades do DISME.

Realizadas estas três tarefas foi então possível integrar a execução interna com chamadas a *endpoints* externos no Componente API REST. Na Figura 70 é mostrado um exemplo de configuração de uma chamada a uma API externa.

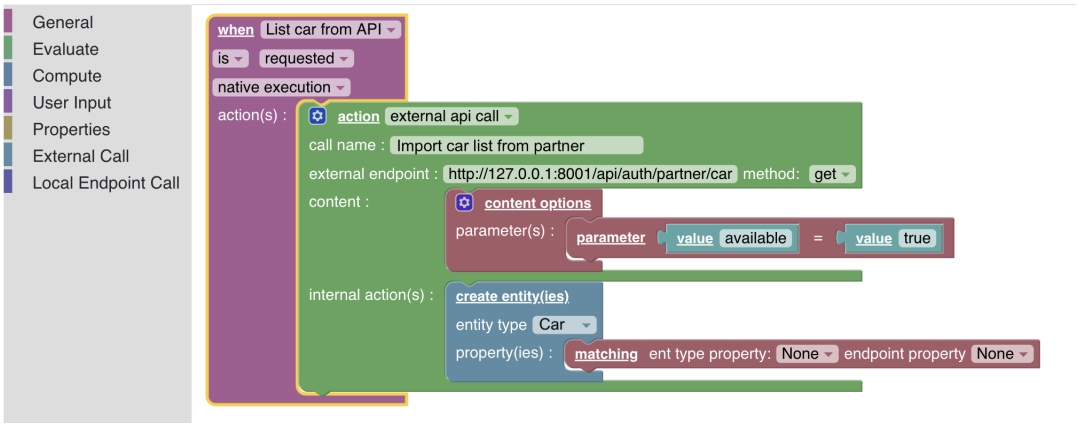


Figura 70. Exemplo de configuração de uma chamada a uma API externa

5.4.4 Ativação da execução interna com chamadas de *endpoints* locais

Para permitir a implementação de *endpoints* locais para a criação de novos objetos, primeiro foi preciso desenvolver os novos blocos *Blockly* que vão conter/solicitar todas as informações necessárias para criar cada *endpoint* de forma adequada, na forma de regras de ação internas, bem como algumas informações adicionais para criar a respetiva documentação.

As informações necessárias para cada *endpoint* são as seguintes, sendo que algumas delas podem ser deduzidas ou usadas para deduzir outras:

- Método HTTP
- URI
- Parâmetros do URI
- Nome do *Endpoint*
- Listagem dos atributos do objeto a ser criado, onde cada um contém:

- Nome do atributo
- Descrição do atributo
- Tipo de Entidade (*Entity Type*)
- Tipo de Valor (*Value Type*) (número, texto, data,...)
- Restrições e/ou validações (por exemplo, verificar se um número é maior que 0)
- Um exemplo de um valor possível para esse atributo
- *Flag* para marcar o atributo como obrigatório ou opcional

– A(s) ação(ões) a ser executada(s):

- Criar as entradas na base de dados usando os atributos
- Fazer a correspondência de cada atributo do *endpoint* com o respetivo atributo do Tipo de Entidade

– As respostas:

- Sucesso:
  * Mensagem de sucesso genérica
  * Objeto recém criado, selecionando os atributos que devem ser enviados na resposta
- Erro - conforme o tipo de erro é gerada uma mensagem adequada

Apesar do componente de Gestão de Regras de Ação do DISME já ter vários blocos do *Blockly* personalizados, foi necessário criar alguns blocos novos e/ou personalizar alguns dos blocos já existentes para dar suporte a esta nova funcionalidade. Primeiramente, o bloco principal, que é usado como ponto de partida para criar regras de ação, teve uma nova opção acrescentada: *"local endpoint call"*, que quando selecionada transforma esse bloco num *template* especifico para criar um novo *endpoint*, nomeadamente o *path* do *endpoint*, os argumentos/parâmetros do *endpoint*, o *body* do pedido, a(s) ação(ões) a serem executada(s) e as respostas possíveis. O bloco principal, com a opção *"local endpoint call"* selecionada é mostrado na Figura 71.

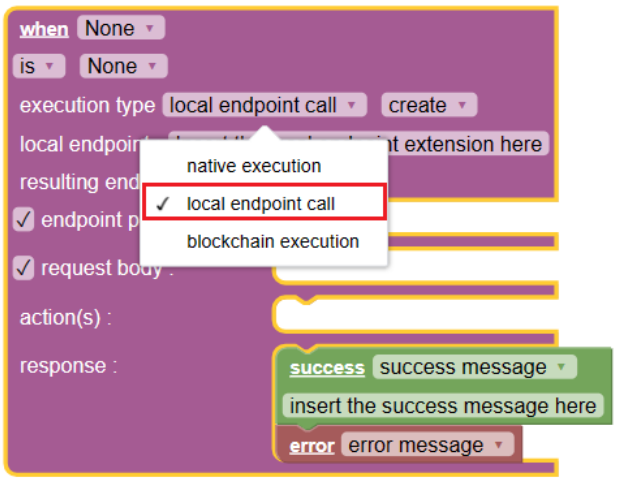


Figura 71. Bloco inicial, com o novo *execution type*

Depois, foi criado um bloco para recolher as informações de cada parâmetro, que podem ser usadas para definir parâmetros/argumentos do *endpoint*, bem como os parâmetros do *body* do pedido. As informações necessárias para preencher este bloco são: o nome do parâmetro, uma breve descrição que é usada para a documentação, o tipo de entidade interno, a propriedade interna, um valor de exemplo para esse parâmetro, uma caixa de seleção para determinar se o parâmetro é ou não obrigatório e também, se necessário, pode ser escolhida a opção de adicionar *"validation condition(s)"*. Nesse caso, será possível escolher uma ou mais condições de validação entre uma lista limitada de validações permitidas com base no tipo de valor do parâmetro. Um exemplo do bloco *"parameter"* pode ser encontrado na Figura 72.

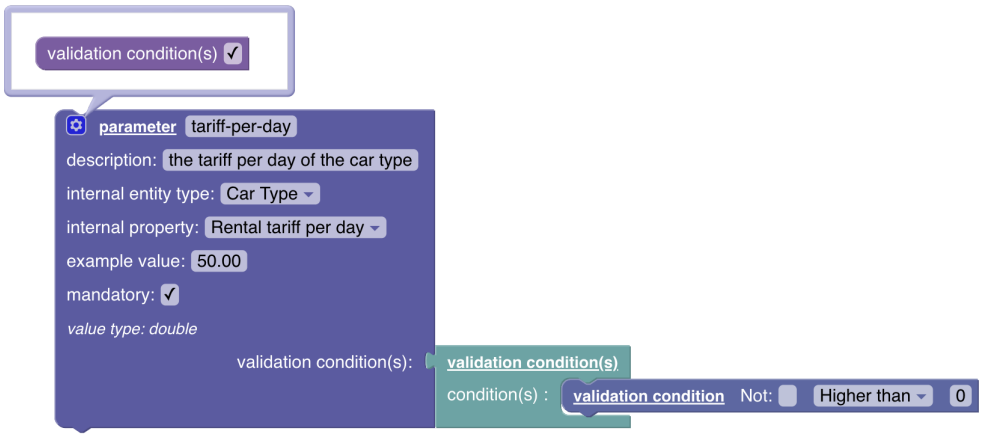


Figura 72. Bloco *"parameter"* com condição de validação ativada

Caso o *endpoint* possua objetos aninhados no pedido, há também a opção de utilizar o bloco *"parameter set"*, que utiliza os mesmos blocos *"parameter"* no seu interior, conforme mostrado na Figura 73.

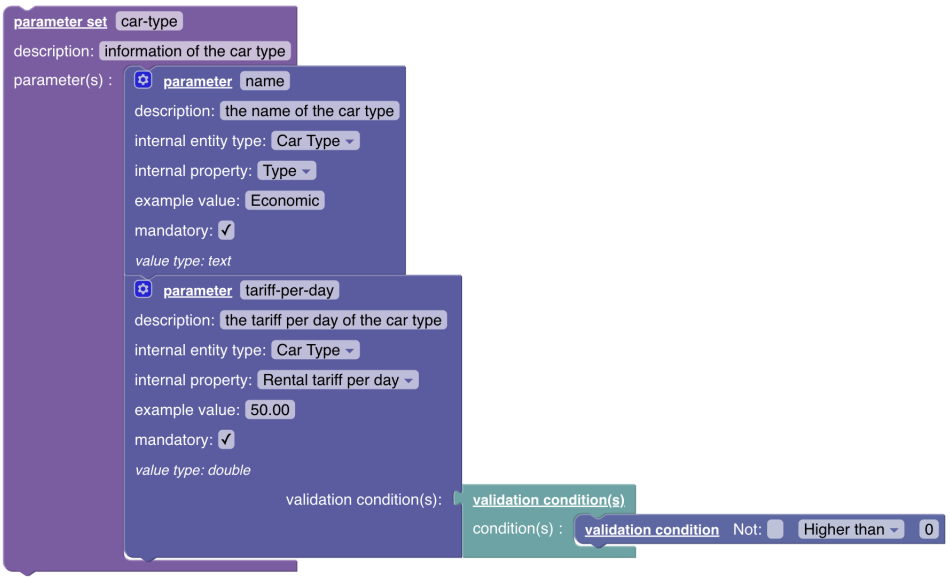


Figura 73. Bloco *"parameter set"*

Em seguida, foi alterado o comportamento do bloco *"action"* quando associado ao bloco inicial da Figura 71 com a opção *"local endpoint call"* selecionada. Como é possível observar na Figura 74, o bloco *"action"* tem agora a opção *"create record"*, o que vai permitir criar as entradas na base de dados. No *"entity type"* escolhemos de qual das entidades existentes estamos a criar uma nova entrada e marcamos uma caixa de seleção caso sejam permitidos registos duplicados. Em *"property(ies)"* usamos um novo bloco chamado *"matching"* que faz a correspondência entre a propriedade do tipo de entidade com a propriedade recebida como parâmetro do *endpoint*. Na Figura 74, é apresentado um exemplo do bloco *"action"* com dois blocos *"matching"*.

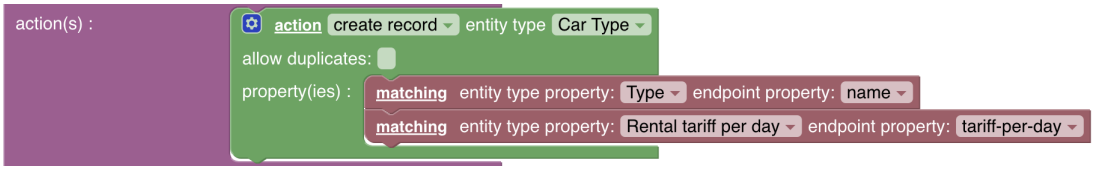


Figura 74. Bloco *"action"* e blocos *"matching"*

Para concluir, é necessário configurar a resposta do *endpoint*. Se existir algum erro durante a sua execução, será retornada uma mensagem e um código de *status* HTTP de acordo com o tipo de erro que ocorreu. Em caso de sucesso, há a opção de enviar uma mensagem genérica de sucesso, conforme mostrado na Figura 71, ou podemos optar por retornar o(s) objeto(s) recém-criado(s) como resposta. Para isso, selecionamos a opção *"created object"* e depois podemos adicionar quantos blocos *"response property"* forem necessários, onde preenchemos o nome da propriedade como queremos que fique na resposta, indicando a que tipo de entidade e propriedade interna correspondem, conforme mostrado na Figura 75. A razão para definir essas propriedades em vez de deduzi-las a partir do pedido ou da própria entidade é que os objetos podem conter mais propriedades que talvez tenham sido geradas pela execução do *endpoint* e também porque podemos não querer retornar/expor todas as propriedades, mas apenas algumas delas.

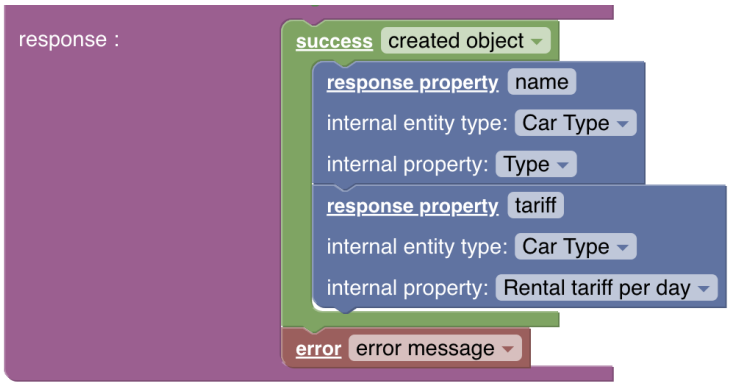


Figura 75. Blocos *"response property"* dentro de um bloco *"success"*

Além disso, de forma semelhante aos blocos *"parameter"*, é possível utilizar o bloco *"response set"* para retornar objetos aninhados, que utiliza os mesmos blocos de "response property" no interior, conforme apresentado na Figura 76.

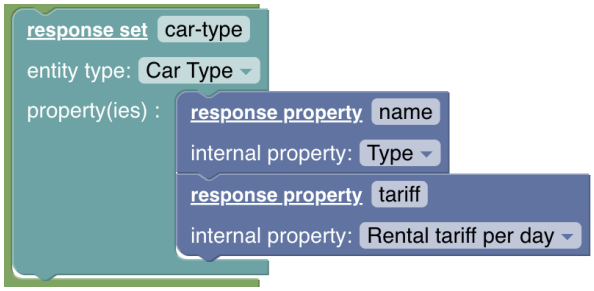


Figura 76. Bloco *"response set"*

5.4.5 Extensão da gramática EBNF do meta modelo de ação

Este trabalho de investigação contribui ainda na sugestão de uma extensão da gramática *Extended Backus-Naur Form* (EBNF) existente para as regras de ação do DISME. No âmbito da engenharia informática, a gramática EBNF constitui uma notação de meta-sintaxe, que pode ser usada para representar uma gramática livre de contexto. A principal aplicação do EBNF está na sua utilidade para a articulação formal de linguagens formais, incluindo as linguagens de programação utilizadas na informática.

As modificações descritas nesta parte referem-se diretamente à Subsecção 5.4.4. Na Tabela 2, apresentada mais abaixo, apenas são especificados os conceitos novos ou atualizados dentro da gramática EBNF das regras de ação do DISME. Esta abordagem é adotada para evitar o uso excessivo de espaço e para manter o foco na discussão relevante. Todas as regras da gramática EBNF podem ser encontradas em [60], mas para compreender melhor a Tabela 2, ficam aqui listadas as regras mais utilizadas:

- "[ ]" é para elementos opcionais (zero ou uma vez).
- "{ }" é para elementos opcionais (zero ou mais vezes).
- "|" significa alternativa.
- "-" significa *Syntactic Exception*. Separa uma regra que deve ser utilizada (à esquerda) da regra que descreve o que não pode ser utilizado (à direita) ("Consoante = Letra - Vogal;"). Se não houver nada à direita, só pode ser interpretado como algo que deve ser utilizado ("UmOuMais = Algo-;").
- elementos em MAIÚSCULAS significam um símbolo terminal com um comportamento específico atribuído no sistema/painel.

Uma regra de ação opera no contexto de um tipo de transação predefinido no sistema. Esta regra é ativada durante um estado específico da transação. Esta proposta expande a definição da regra de ação através da introdução um tipo de execução categorizado em dois modos distintos: a execução nativa, que corresponde à execução padrão de regras de ação através das interações do utilizador no *dashboard* da plataforma, e a execução por chamada a um *endpoint* local, que envolve a execução da regra através de uma chamada à API por parte de um sistema externo para um *endpoint* definido, como descrito anteriormente.

A Tabela 2 está estruturada da seguinte forma: a coluna da esquerda enumera os conceitos que requerem definição, em que a maioria corresponde a blocos específicos do sistema, e outros referem-se a campos selecionáveis dentro desses blocos; e a coluna direita da tabela é dedicada a fornecer

as definições explícitas desses conceitos. No caso de ser definida uma regra de ação com o tipo de execução '*local endpoint call*', como mostrado anteriormente, é necessário especificar o *endpoint* local com o qual os sistemas externos vão interagir. Em seguida, são detalhados os parâmetros da chamada à API, que podem incluir propriedades do sistema obrigatórias ou opcionais. Por fim, é necessário definir a resposta esperada da chamada. Isto permite a especificação explícita das propriedades, incluindo o seu nome e tipo de valor. Alternativamente, pode ser definido um grupo de propriedades especificando os seus nomes e selecionando-os a partir de um tipo de entidade do sistema.

Tabela 2. Tabela EBNF das regras de ação com a especificação dos conceito adicionados/atualizados para chamadas de API internas

execution_type	NATIVE_EXECUTION | LOCAL_ENDPOINT_CALL
local_endpoint_call_info	local_endpoint_call_type local_endpoint_path {local_endpoint_parameter} [local_endpoint_request_body] local_endpoint_response
local_endpoint_call_type	CREATE | READ | UPDATE | DELETE NOTE: API Call CRUD operation that will be carried. Influences the action types that will be present in the 'action' block. For 'create', we will have the 'assign expression' and 'create record' action types, for 'read' we will have the 'query ledger' and 'query asset' action types, for 'update' we will have the 'update asset' action type, and for 'delete' we will have the 'delete asset' action type.
local_endpoint_path	STRING NOTE: path that will extend the application's domain. Ex: "/getParcel". The url parameters are automatically inserted after the path by Blockly as they are added into the action rule, originating the resulting endpoint. Ex: "/getParcel/id"
local_endpoint_parameter	STRING property documentation_description parameter_example_value {validation_condition} [MANDATORY] NOTE: the parameter name that will be used to specify the parameter is independent of the internal property that is linked to it, and therefore doesn't have to be the equal to its name.
documentation_description	STRING NOTE: Description of the parameter/parameter_set, needed for the generation of the api endpoint through Swagger.
parameter_example_value	STRING NOTE: An example of the parameter's accepted value. Needed for the generation of the api endpoint through Swagger.
local_endpoint_request_body	{local_endpoint_parameter | local_endpoint_parameter_set}- NOTE: defines the structure of the API Call's reqest_body, defining its parameters and/or parameter_sets.
local_endpoint_parameter_set	STRING documentation_description {local_endpoint_parameter}- NOTE: After specifying the set_name, it is needed to specify the parameters that are to be inserted into it.

local_endpoint_response	local_endpoint_success_response local_endpoint_error_response NOTE: must have a behaviour selected for each one. One defined in case the API call is carried out successfully, and one in case it encounters any errors in the process.
local_endpoint_success_response	local_endpoint_success_response_affected_object | local_endpoint_success_response_queried_object | STRING NOTE: When successful, it can return the created/queried/updated/deleted object or simply a custom success message. NOTE: API Call CRUD operation that will be carried influences the dropdown choices that will be present in the 'response' block. For 'create', we will have the 'created object', for 'read' we will have the 'response_queried_object', for 'update' we will have the 'updated object', and for 'delete' we will have the 'deleted object'. Except for the 'read' crud operation, we have the option to output a custom success message instead of the object.
local_endpoint_success_response_affected_object	STRING {response_property | response_set}- NOTE: defines the returning object with the object's name and the properties/property_sets that should be returned inside it.
response_property	STRING property NOTE: has to be a property belonging to the object that was created/updated/queried/deleted.
response_set	STRING {response_property}- NOTE: defines the set name and the properties that should be included in this response set.
local_endpoint_success_response_queried_object	STRING NOTE: the variable names defined in the 'query records' blocks will appear here in a dropdown and the user must select which ones to include in the api call's response. Will return the objects from the selected queries.
local_endpoint_error_response	STRING NOTE: error message to be returned.
action	causal_link | assign_expression | user_input | edit_entity_instance | user_output | produce_doc | if | while | for_each | post | get | create_record | query_records | update_record | delete_record
set	STRING set of elements NOTE: set of elements. Has to be a name of a 'parameter set' defined in the action rule's 'request body' input.
create_record	entity_type {matching_property}- [ALLOW_DUPLICATES] NOTE: Allows the creation of an entity in the system. 'Allow duplicates' is a checkbox that will/won't allow the insertion of duplicate records of entities of the entity type selected.
entity_type	STRING *NOTE: has to be an existent entity_type specified in table ent_type*

matching_propeqrty	MATCHING property local_endpoint_property NOTE: matches an internal property to an input parameter/property of the API Call defined in the 'local_endpoint_parameter'/ 'local_endpoint_request_body' inputs.
local_endpoint_property	STRING NOTE: has to be a property referenced by a local_endpoint_parameter inside the action rule's definition.
matching_id_property	MATCHING ID_PROPERTY local_endpoint_property NOTE: the only matching_property inserted here is the identifying property of the selected entity type, i.e. the 'id' property. Blockly automatically inserts it into the block as the matching_property's property.
query_records	query [QUERY_DELETED_OBJECTS] NOTE: get query results from the specified query. Blockly will have a dropdown to select the queries currently defined in the system.
update_record	entity_type matching_id_property {matching_property} NOTE: properties to be updated in the entity are the matching_properties inserted. The matching_id_property is to know which entity to update.
delete_record	entity_type matching_id_property NOTE: will search for the selected entity type's entity with the received api call id and will delete that entity.

5.4.6 Geração automática da documentação API

No que diz respeito às APIs REST, a existência da respetiva documentação é igualmente importante, não só para as equipas de desenvolvimento, mas também para os consumidores finais da API, para que possam visualizar e interagir com os seus recursos sem ter nenhuma lógica implementada. Assim sendo, esta proposta de RRAM inclui a geração automática da documentação da API, com base nas propriedades da API especificadas com a abordagem *low-code*. Para tal, será utilizado o Swagger UI, uma ferramenta *opensource* que gera uma página web que documenta a API, seguindo a *OpenAPI Specification*[65] (OAS, anteriormente conhecida como *Swagger Specification*). Esta documentação da API é simples, *user-friendly* e também fácil de alterar/atualizar.

A geração da documentação com o *Swagger UI* será baseada na utilização de anotações para definir a informação que aparece na página web gerada. A maior parte desta informação, como o método HTTP, parâmetros, campos de resposta e assim por diante, pode ser deduzida com base nas especificações feitas com os componentes mencionados nas secções anteriores. As informações legíveis por humanos, como nomes e descrições, já serão definidas quando estiverem a ser criadas as *queries* complexas e/ou regras de ação específicas, ou podem ser geradas automaticamente com base nas especificações/nomes mencionados. Sem qualquer lógica de implementação, a UI do *Swagger* permite que qualquer pessoa, incluindo equipas de desenvolvimento e utilizadores finais, visualize e interaja com as capacidades de uma API.

[65] OpenAPI Specification - `https://swagger.io/specification/`

6 Conclusões e Trabalho Futuro

Neste capítulo, são expostas as conclusões finais acerca de ambos os projetos. Depois, são apresentadas algumas publicações nas quais contribuí durante o período de tempo em que estive a colaborar nos vários projetos mencionados na Secção 1.1, e ainda, são enumeradas algumas possíveis melhorias/trabalho futuro no âmbito dos projetos *GesFoGO* e DISME.

6.1 Conclusões

O projeto *GesFoGO* foi um projeto que envolveu a cooperação entre várias organizações, instituições e empresas com capacidade operacional e científico-tecnológica para desenvolver uma rede abrangente de prevenção e gestão de incêndios florestais em tempo real, através de unidades móveis de rápida implantação com sistema auto-georreferenciado, com o objetivo de alcançar uma gestão sustentável em ambientes florestais montanhosos, caraterísticos das regiões envolvidas.

A solução final integra duas partes principais: uma desenvolvida pelos parceiros da ULPGC e a outra pela ARDITI. A ULPGC desenvolveu o *hardware* dos sensores, o respetivo *software* para controlá-los e também os algoritmos de georreferenciação que determinam a localização dos focos de incêndio. Já a ARDITI ficou encarregue de desenvolver a base de dados, a API REST e a interface gráfica do utilizador (GUI) para o *GesFoGO*. Na base de dados são armazenados todos os dados do *GesFoGO*, tanto as informações necessárias para dar suporte à GUI como as configurações e capturas provenientes dos sensores. Os acessos à BD são realizados exclusivamente através da API REST, que disponibiliza os *endpoints* para dar suporte às operações da GUI e para receber as informações geradas pelos sensores. Através da GUI o utilizador pode visualizar o mapa com os sensores, linhas de fogo/sombra, imagens visíveis/térmicas e também, pode enviar comandos para alterar algumas configurações dos sensores, como o campo de visão, o ângulo para onde apontam, a frequência de captura e alternar entre os modos fixo e de varrimento.

O *GesFoGO* foi um projeto desafiante e de grande responsabilidade, pois foi o primeiro projeto desta dimensão em que colaborei, onde aprofundei grande parte dos conhecimentos que fui adquirindo ao longo do meu percurso académico. Para além disso, fiquei a conhecer melhor a logística deste tipo de projetos que envolvem várias entidades e parcerias, nomeadamente através das reuniões regulares para esclarecimento de dúvidas, alteração de requisitos, planeamento, entre outros aspetos. O desenvolvimento da aplicação do *GesFoGO* englobou uma grande variedade de conhecimentos e tarefas, não só no desenvolvimento do *backend* e *frontend*, mas também na elaboração da documentação/relatórios sobre cada componente, o *deploy* da aplicação no servidor do EELab, a integração das partes da ARDITI e ULPGC, os testes no terreno e também as demonstrações.

Apesar de ainda haver possíveis melhorias para o *GesFoGO*, mencionadas na Secção 6.3, considero que, de um modo geral, o projeto foi bem sucedido. Os requisitos funcionais foram cumpridos e toda a aplicação funciona corretamente. Apesar de, nos primeiros testes no terreno terem sido identificados alguns problemas, estes foram corrigidos e nos testes/demonstração final correu tudo bem. O feedback recebido do sistema por parte dos especialistas em prevenção e combate a incêndios foi bastante positivo e não foi identificada informação em falta, por isso considero que o projeto foi concluído com sucesso.

No que diz respeito à integração de mecanismos de API REST na plataforma *low-code* DISME, foi apresentada uma nova abordagem para a rápida gestão e implementação de APIs REST, baseada na metodologia DEMO e numa solução *low-code* orientada por modelos. A nossa solução, integrada

na plataforma DISME, fornece uma forma metódica de criar e gerir APIs. Tirando partido e atualizando o meta modelo de ação do DEMO, tornou-se possível a geração automática ou rápida de APIs de *input* e *output* através de uma interface intuitiva de *drag-and-drop*.

A nossa abordagem de combinar os modelos de ação DEMO com os componentes da plataforma DISME reduz a complexidade do desenvolvimento e gestão de APIs, reduzindo o tempo e o esforço necessários quando comparado com a abordagem tradicional. A geração automática de documentação da API garante maior clareza e facilita a manutenção e utilização do sistema. Alinhado com estes avanços, ainda serão realizados testes mais abrangentes para aferir a usabilidade, integração e desempenho do sistema, para garantir que esta funcionalidade satisfaz as necessidades dos utilizadores eficazmente.

Como o desenvolvimento foi feito com base numa componente específica do DISME, foi possível o desenvolvimento de funcionalidades de *queries* dinâmicas que permitem a parametrização de *queries* de forma dinâmica. Esta melhoria permite ainda aos utilizadores definir os parâmetros das *queries* durante a execução, aumentando a flexibilidade e a utilidade da nossa solução para gestão de APIs.

Para concluir, considero que o desenvolvimento destes projetos me concedeu muita experiência, uma vez que exigiu o domínio de várias competências, que tentei encarar sempre com a maior responsabilidade e empenho possível. Além disso, considero que o desenvolvimento desta funcionalidade no DISME foi uma boa forma de consolidar e pôr à prova os conhecimentos acerca de APIs REST que adquiri com os vários projetos em que colaborei.

6.2 Publicações

Durante o período de realização desta tese colaborei na escrita de quatro artigos científicos, que foram submetidos, aceites e publicados, onde são demonstrados os resultados e conclusões da investigação realizada.

Na fase final do projeto *GesFoGO*, colaborei na escrita do artigo científico "*Ontology for a Georeferencing Mobile System for Real Time Detection and Monitoring of Wildfires*" [5], para a 14ª Conferência Internacional em *Knowledge Engineering and Ontology Development*[66] (KEOD) 2022, que teve como base os desenvolvimentos do projeto *GesFoGO*. Este artigo apresenta a ontologia, de uma forma genérica, que serve como base para a implementação de um sistema móvel de deteção e monitorização de incêndios florestais por georreferenciação, particularmente para territórios montanhosos e íngremes. Isto é, o paper retrata o conjunto de dados que representam os conceitos e as relações entre eles, neste domínio específico dos incêndios florestais, com o intuito de ajudar os desenvolvedores a acelerar o processo de análise de requisitos na projeção de um novo sistema. O conteúdo deste artigo científico incorpora alguma da informação presente no Capítulos 3 e na Secção 4.4.

Contribuí também na escrita de dois artigos científicos para a *12th Enterprise Engineering Working Conference*[67] (EEWC) 2022. Um deles acerca do DISME, intitulado "*The DISME low-code platform - from simple diagram creation to system execution*" [6], onde é apresentada uma visão geral sobre o DISME e são explicados detalhadamente cada um dos seus componentes e como se conectam entre si para criar um sistema dinâmico onde as alterações feitas se refletem de forma

[66] KEOD 2022 - `https://keod.scitevents.org/?y=2022`
[67] EEWC 2022 - `https://ee-institute.org/eewc/eewc-2022/`

imediata na sua execução. O conteúdo deste paper encontra-se, de forma resumida, na Subsecção 5.1.3. Já o outro paper, intitulado "*DEMO Model based Rapid REST API Management in a low code platform*" [7], apresenta um *proof of concept* para a adição de mecanismos de criação e gestão de APIs REST através de plataformas *low-code* com base nos modelos DEMO.

Por fim, no seguimento do artigo "*DEMO Model based Rapid REST API Management in a low code platform*" [7] da EEWC 2022, foi escrito um novo artigo científico para a *13th Enterprise Design and Engineering Working Conference*[68] (EDEWC) 2023, denominado de "*Rapid REST API Management in a DEMO Based Low Code Platform*" [8] onde são demonstrados os avanços relativamente aos mecanismos de criação e gestão de APIs REST para plataformas *low-code* abordados no paper anterior. Apresenta uma versão mais refinada e mais concreta daquilo que foi proposto/abordado no paper anterior com algumas alterações/adaptações e correções conforme foram identificados problemas durante a implementação desta funcionalidade no DISME. O Capítulo 5 engloba o conteúdo destes dois artigos científicos.

Além disso, durante todo o período de desenvolvimento do *GesFoGO* foram publicados no site oficial do projeto (`https://www.gesfogo.ulpgc.es/`), uma série de recursos acerca de todas as atividades realizadas. Estes recursos encontra-se organizados cronologicamente no separador 'ATIVIDADES' do site (`https://www.gesfogo.ulpgc.es/index.php/pt/atividades-pt`). Nesta página do site constam várias fotografias, vídeos e capturas de ecrã acerca do desenvolvimento do sistema (*hardware* e *software*), vídeos demonstrativos, ações de divulgação e artigos científicos relacionados. Além disso, no separador 'NOTÍCIAS' (`https://www.gesfogo.ulpgc.es/index.php/pt/noticias-pt`) encontram-se várias notícias e reportagens publicadas pelos meios de comunicação social das várias regiões envolvidas no projeto.

6.3 Trabalho Futuro

No decorrer do projeto *GesFoGO*, foram identificadas algumas características e funcionalidades que poderiam ser melhoradas e/ou adicionadas que acabaram por não ser implementadas por falta de tempo ou por serem consideradas pouco prioritárias ou pouco adequadas. Nesta secção são enumeradas algumas dessas possíveis melhorias que poderiam ser exploradas para aumentar a utilidade do trabalho desenvolvido.

Como foi referido na Secção 2.1, onde são apresentados os requisitos funcionais para o projeto, é mencionada uma funcionalidade designada por "Assistente de localização", que a partir da localização atual em que um sensor está instalado, gera/determina algumas localizações alternativas que otimizam a sua área de cobertura. Além disso, gera polígonos (semelhantes aos usados para representar as linhas de fogo/sombra) que delimitam a área visível total que o sensor pode capturar, tendo em consideração a topografia circundante, como montanhas que podem reduzir a visibilidade. Na Secção 3.1, onde é explicada a estrutura da base de dados, constam algumas tabelas que foram criadas precisamente para esta funcionalidade, nomeadamente a entidade *Location* para representar as várias localizações alternativas e a entidade *Range_Polygon* para armazenar os polígonos correspondentes à área abrangida pelo sensor naquela localização alternativa. O algoritmo que fazia o calculo das zonas abrangidas pelo sensor em cada localização estava a cargo dos parceiros da ULPGC. Apesar de ter implementado as rotas da API necessárias para dar suporte a esta funcionalidade, devido à sua complexidade percebemos que não teríamos tempo útil suficiente

[68] EDEWC 2023 - `https://ede-network.org/edewc/edewc2023/`

para implementá-la de forma satisfatória, tanto do ponto de vista do algoritmo como da integração na GUI, portanto foi decidido de forma colaborativa que esta funcionalidade não seria realizada.

Em relação às notificações, foi criada uma tabela na BD para isso, idealizada especificamente para complementar a funcionalidade do "Assistente de localização", mencionada acima, que por conseguinte ficou inutilizada. No entanto, um sistema de notificações com alertas quando é detetado algum incêndio ou quando é feita alguma alteração de configurações dos sensores, etc. seria uma boa adição à aplicação do *GesFoGO*.

A GUI *GesFoGO* consiste apenas numa página, mas seria interessante ter pelo menos mais duas páginas para algumas ações complementares, uma página para autenticação e uma para gestão de regiões, campanhas, sensores, etc. Em relação à autenticação, apesar desta ser suportada pela API e por todos os seus *endpoints*, de momento não existe forma de fazer o registo, *login* nem *logout* através da página. No que diz respeito à gestão das regiões, campanhas, sensores, entre outros, seria bom ter uma página separada onde fosse possível criar/gerir as instâncias destas entidades porque, de momento estas instâncias são criadas/atualizadas utilizando uma ferramenta como o *Postman* ou adicionadas diretamente na BD.

Tendo em consideração os testes no terreno e a possibilidade de ser necessário armazenar e servir grandes quantidades de capturas (sobretudo as imagens), estas podem facilmente utilizar grande parte do espaço de armazenamento disponível. As imagens geradas pelos sensores têm uma resolução relativamente elevada, portanto acho que estas seriam algumas possibilidades para reduzir o espaço ocupado pelas imagens: utilizar algum algoritmo de compactação principalmente quando são imagens em que não foi detetado nenhum foco de incêndio e/ou cortar partes da imagem que sejam irrelevantes, por exemplo, a parte do céu.

Por fim, seria importante acrescentar suporte a várias línguas, isto porque o *GesFoGO* foi implementado apenas em inglês.

Quanto à adição de mecanismos para gestão de APIs REST na plataforma DISME, também identificamos alguns aspetos que pretendemos melhorar/acrescentar futuramente, como a integração da capacidade para lidar com pedidos REST com parâmetros dinâmicos, aumentando a capacidade das regras de ação configuradas no DISME e disponibilizadas em *endpoints* locais para sistemas externos. Além disso, atualmente apenas são suportados métodos simples de autenticação através de *tokens* conforme a arquitetura REST e pretendemos acrescentar suporte para outros protocolos de autenticação mais complexos, como o *OAuth* e também o protocolo SOAP, incluindo a gestão dos mesmos. À data de escrita/entrega deste relatório, esta funcionalidade está a ser testada e melhorada em contexto real num projeto piloto com a Câmara Municipal do Funchal (CMF).

Referências

[1] J. San-Miguel-Ayanz, E. Schulte, G. Schmuck, A. Camia, P. Strobl, G. Libertà, C. Giovando, R. Boca, F. Sedano, P. Kempeneers, D. McInerney, C. Withmore, S. Oliveira, M. Rodrigues, T. Durrant, P. Corti, F. Oehler, L. Vilar, and G. Amatulli, *Comprehensive Monitoring of Wildfires in Europe: The European Forest Fire Information System (EFFIS)*, 03 2012.

[2] J. Perez-Mato, V. Araña, and F. Cabrera Almeida, "Real-time autonomous wildfire monitoring and georeferencing using rapidly deployable mobile units," *IEEE Aerospace and Electronic Systems Magazine*, vol. 31, pp. 6–15, 02 2016.

[3] V. Arana-Pulido, F. Cabrera-Almeida, J. Perez-Mato, B. P. Dorta-Naranjo, S. Hernandez-Rodriguez, and E. Jimenez-Yguacel, "Challenges of an autonomous wildfire geolocation system based on synthetic vision technology," *Sensors*, vol. 18, no. 11, 2018. [Online]. Available: https://www.mdpi.com/1424-8220/18/11/3631

[4] J. Dietz and H. Mulder, *Enterprise Ontology: A Human-Centric Approach to Understanding the Essence of Organisation*. Springer Cham, Jan. 2020.

[5] D. Pacheco, D. Aveiro, V. Caires, and D. Pinto, "Ontology for a Georeferencing Mobile System for Real Time Detection and Monitoring of Wildfires," in *Proceedings of the 14th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, vol. 2, Valletta, Malta, 2022, pp. 260–268. [Online]. Available: https://www.scitepress.org/Link.aspx?doi=10.5220/0011592100003335

[6] V. Freitas, D. Pinto, V. Caires, L. Tadeu, and D. Aveiro, "The DISME low-code platform-from simple diagram creation to system execution." in *Proceedings of the 22nd CIAO! Doctoral Consortium, and Enterprise Engineering Working Conference Forum 2022 co-located with 12th Enterprise Engineering Working Conference (EEWC 2022)*, ser. CEUR Workshop Proceedings, S. Guerreiro, C. Griffo, and M. Jacob, Eds. Leusden, Netherlands: CEUR-WS.org, 2022. [Online]. Available: https://ceur-ws.org/Vol-3388/paper4.pdf

[7] D. Aveiro and V. Caires, "DEMO Model based Rapid REST API Management in a low code platform," in *Proceedings of the 22nd CIAO! Doctoral Consortium, and Enterprise Engineering Working Conference Forum 2022 co-located with 12th Enterprise Engineering Working Conference (EEWC 2022)*, ser. CEUR Workshop Proceedings, S. Guerreiro, C. Griffo, and M. Jacob, Eds., vol. 3388. Leusden, Netherlands: CEUR-WS.org, 2022. [Online]. Available: https://ceur-ws.org/Vol-3388/paper2.pdf

[8] V. Caires, J. Vasconcelos, D. Pinto, V. Freitas, and D. Aveiro, "Rapid REST API Management in a DEMO Based Low Code Platform," in *Advances in Enterprise Engineering XVII*, M. Malinova Mandelburger, S. Guerreiro, C. Griffo, D. Aveiro, H. A. Proper, and M. Schnellmann, Eds. Cham: Springer Nature Switzerland, 2024, pp. 73–91.

[9] P. Beynon-Davies, C. Carne, H. Mackay, and D. Tudhope, "Rapid application development (RAD): An empirical review," *European Journal of Information Systems*, vol. 8, Sep. 1999.

[10] "Application Programming Interface," Jan. 2024. [Online]. Available: https://www.techopedia.com/definition/24407/application-programming-interface-api

[11] “What is REST?” [Online]. Available: https://www.codecademy.com/article/what-is-rest

[12] “Aircraft principal axes,” Feb. 2024, page Version ID: 1201908293. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Aircraft_principal_axes&oldid=1201908293

[13] MladjoA, “Polygon - SQL Server,” Feb. 2023. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/spatial/polygon?view=sql-server-ver16

[14] “Directory Structure - Laravel 8.x - The PHP Framework For Web Artisans.” [Online]. Available: https://laravel.com/docs/8.x/structure

[15] “Eloquent: Relationships - Laravel 8.x - The PHP Framework For Web Artisans.” [Online]. Available: https://laravel.com/docs/8.x/eloquent-relationships

[16] “Sanctum - Laravel 8.x - The PHP Framework For Web Artisans.” [Online]. Available: https://laravel.com/docs/8.x/sanctum

[17] Elliott, “ejarnutowski/laravel-api-key,” Nov. 2023, original-date: 2017-11-01T18:46:22Z. [Online]. Available: https://github.com/ejarnutowski/laravel-api-key

[18] “What’s an API Endpoint?” [Online]. Available: https://smartbear.com/learn/performance-monitoring/api-endpoints/

[19] “MDN - HTTP Messages,” Feb. 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

[20] “W3Schools - HTTP Methods.” [Online]. Available: https://www.w3schools.com/tags/ref_httpmethods.asp

[21] “Status codes - HTTP | MDN,” Nov. 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

[22] “Routing - Laravel 8.x - The PHP Framework For Web Artisans.” [Online]. Available: https://laravel.com/docs/8.x/routing#route-groups

[23] “Eager Loading - Laravel 8.x - The PHP Framework For Web Artisans.” [Online]. Available: https://laravel.com/docs/8.x/eloquent-relationships#eager-loading

[24] “Raw Methods - Laravel 8.x - The PHP Framework For Web Artisans.” [Online]. Available: https://laravel.com/docs/8.x/queries#raw-methods

[25] “Validation - Laravel 8.x - The PHP Framework For Web Artisans.” [Online]. Available: https://laravel.com/docs/8.x/validation#quick-writing-the-validation-logic

[26] “Transactions - Laravel 8.x - The PHP Framework For Web Artisans.” [Online]. Available: https://laravel.com/docs/8.x/database#manually-using-transactions

[27] “HTTP Client - Laravel 8.x - The PHP Framework For Web Artisans.” [Online]. Available: https://laravel.com/docs/8.x/http-client#making-requests

[28] “Decimal degrees,” Mar. 2024, page Version ID: 1212356747. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Decimal_degrees&oldid=1212356747

[29] “Introduction | Scribe.” [Online]. Available: https://scribe.knuckles.wtf/laravel/

[30] A. J. Meyghani, “JavaScript Bundlers, a Comparison,” Jun. 2019. [Online]. Available: https://medium.com/@ajmeyghani/javascript-bundlers-a-comparison-e63f01f2a364

[31] "ECMAScript - JavaScript technologies overview - JavaScript | MDN," Nov. 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/JavaScript_technologies_overview

[32] "Production - Webpack." [Online]. Available: https://webpack.js.org/guides/production/

[33] O. F. openjsf.org, "jQuery API Documentation." [Online]. Available: https://api.jquery.com/

[34] R. Briceno, "Relatório de Estágio - Projeto Red GesFoGO," Universidade da Madeira, Tech. Rep., Jul. 2020.

[35] "fetch() global function - Web APIs | MDN," Apr. 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/fetch

[36] "Quick Start Guide - Leaflet - a JavaScript library for interactive maps." [Online]. Available: https://leafletjs.com/examples/quick-start/

[37] "Documentation - Leaflet - a JavaScript library for interactive maps." [Online]. Available: https://leafletjs.com/reference.html

[38] M. AG, "MapTiler Cloud API | Api | MapTiler." [Online]. Available: https://docs.maptiler.com/cloud/api

[39] "Cross-Origin Resource Sharing Explained - AWS." [Online]. Available: https://aws.amazon.com/what-is/cross-origin-resource-sharing/

[40] "Same-origin policy - Security on the web | MDN," Dec. 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[41] "CORS (Cross-Origin Resource Sharing) - Laravel 8.x - The PHP Framework For Web Artisans." [Online]. Available: https://laravel.com/docs/8.x/routing#cors

[42] "Preflight request - MDN Web Docs Glossary: Definitions of Web-related terms | MDN," May 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Preflight_request

[43] "Request: mode property - Web APIs | MDN," Oct. 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Request/mode

[44] H. Jethva, "Host Multiple Websites on a Single Server with Apache on Ubuntu," Jan. 2020. [Online]. Available: https://www.atlantic.net/vps-hosting/host-multiple-websites-on-a-single-server-with-apache-on-ubuntu/

[45] J. Cameron, "Webmin | Documentation." [Online]. Available: https://webmin.com/docs/

[46] M. Andrade, D. Aveiro, and D. Pinto, "Bridging Ontology and Implementation with a New DEMO Action Meta-model and Engine," in *Advances in Enterprise Engineering XIII*, Lisbon, Portugal, Jan. 2020, pp. 66–82.

[47] D. Pinto, D. Aveiro, D. Pacheco, B. Gouveia, and D. Gouveia, "Validation of DEMO's Conciseness Quality and Proposal of Improvements to the Process Model," in *Advances in Enterprise Engineering XIV*, Bozen-Bolzano, Italy, Apr. 2021, pp. 133–152.

[48] B. Gouveia, D. Aveiro, D. Pacheco, D. Pinto, and D. Gouveia, "Fact Model in DEMO - Urban Law Case and Proposal of Representation Improvements," in *Advances in Enterprise Engineering XIV*, Apr. 2021, pp. 173–190.

[49] D. Pacheco, D. Aveiro, D. Pinto, and B. Gouveia, "Towards the X-Theory: An Evaluation of the Perceived Quality and Functionality of DEMO's Process Model," in *Advances in Enterprise Engineering XV*, D. Aveiro, H. A. Proper, S. Guerreiro, and M. de Vries, Eds. Cham: Springer International Publishing, 2022, pp. 129–148.

[50] D. Pacheco, D. Aveiro, B. Gouveia, and D. Pinto, "Evaluation of the Perceived Quality and Functionality of Fact Model Diagrams in DEMO," in *Advances in Enterprise Engineering XV*, D. Aveiro, H. A. Proper, S. Guerreiro, and M. de Vries, Eds. Cham: Springer International Publishing, 2022, pp. 114–128.

[51] D. Aveiro and V. Freitas, "A New Action Meta-model and Grammar for a DEMO Based Low-Code Platform Rules Processing Engine," in *Advances in Enterprise Engineering XVI*, C. Griffo, S. Guerreiro, and M. E. Iacob, Eds. Cham: Springer Nature Switzerland, 2023, pp. 33–52.

[52] D. Aveiro and J. Oliveira, "Towards DEMO Model-Based Automatic Generation of Smart Contracts," in *Advances in Enterprise Engineering XVI*, C. Griffo, S. Guerreiro, and M. E. Iacob, Eds. Cham: Springer Nature Switzerland, 2023, pp. 71–89.

[53] B. Wang, D. Rosenberg, and B. W. Boehm, "Rapid realization of executable domain models via automatic code generation," in *2017 IEEE 28th Annual Software Technology Conference (STC)*, Sep. 2017, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/8234464#citations

[54] A. Mora-Segura, J. Sanchez Cuadrado, and J. Lara, "ODaaS: Towards the Model-Driven Engineering of Open Data Applications as Data Services," in *2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations*, Sep. 2014, pp. 335–339.

[55] R. Gonçalves and I. Azevedo, "RESTful Web Services Development With a Model-Driven Engineering Approach," in *Code Generation, Analysis Tools, and Testing for Quality*, Jan. 2019, pp. 191–228.

[56] S. Hussein, S. Zein, and N. Salleh, "REST API Auto Generation: A Model-Based Approach," in *Knowledge Innovation Through Intelligent Software Methodologies, Tools and Techniques*, Sep. 2020, pp. 246–259.

[57] M. Overeem, S. Jansen, and M. Mathijssen, "API Management Maturity of Low-Code Development Platforms," in *Enterprise, Business-Process and Information Systems Modeling*, A. Augusto, A. Gill, S. Nurcan, I. Reinhartz-Berger, R. Schmidt, and J. Zdravkovic, Eds. Cham: Springer International Publishing, 2021, pp. 380–394.

[58] B. De, *API Management*. Berkeley, CA: Apress, 2017. [Online]. Available: http://link.springer.com/10.1007/978-1-4842-1305-6

[59] M. R. Krouwel and M. Op 't Land, "Business Driven Microservice Design," in *Advances in Enterprise Engineering XV*, D. Aveiro, H. A. Proper, S. Guerreiro, and M. de Vries, Eds. Cham: Springer International Publishing, 2022, pp. 95–113.

[60] "ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF." International Organization for Standardization, 1996.