Plataforma baseada em microserviços para gerenciamento de transformações de modelos

Samuel Possamai* e Gian Ricardo Berkenbrock†

29 de Julho de 2022

Resumo

Neste trabalho é proposta uma plataforma para conversão entre diferentes tipos de arquivos de modelos. Ela busca simplificar o uso de ferramentas de conversão, integrando elas em um único ambiente, sem a necessidade de instalá-las e configurá-las individualmente. Ela também visa suportar a armazenagem de modelos e permitir a organização deles. Dessa forma, facilitando a integração de modelos em diferentes formatos, permitindo que eles sejam armazenados e convertidos em uma mesma aplicação. Para elaborar a plataforma, no presente texto, foi elaborado requisitos para o projeto e definido uma arquitetura e especificação de um API interna e externa visando cumpri-los.

Palavras-chave: microsserviços, modelos, transformação e requisitos.

1 Introdução

O uso de aplicações de *software* já faz parte do cotidiano de grande parte da população mundial, podendo elas estarem sendo executadas diretamente no computador ou celular do usuário final, ou indiretamente por um servidor na nuvem. Elas têm diversas utilidades que abrangem desde o acesso à informação e notícias até entretenimento e até mesmo o desenvolvimento de outras aplicações ou aperfeiçoar a ela mesma.

No desenvolvimento de software são utilizadas diversas ferramentes que incluem a parte física(*hardware*) e virtual(*software*). Uma das categorias de ferramentas essenciais é as linguagens de programação, elas permitem ao programador dar instruções a máquina, porém existem muitas outras ferramentas que cumprem um papel muito importante para diferentes etapas de um projeto de construção de programas. Essas etapas incluem desde o início do projeto até os momentos finais, como a identificação de falhas, o planejamento da

*possamai.sam@gmail.com
†gian.rb@ufsc.br/Orientador

estrutura e fluxo da aplicação e a entrega automática de *software* ao local adequado para entrar em funcionamento.

Antes de iniciar o desenvolvimento de um programa é vital entender quais funções ele deverá ter, quais serão as principais ferramentas usadas e planejar como a estrutura dele deverá ser e como ele se comportará. Se essa etapa não for executada apropriadamente o desempenho do programa, tanto no momento de seu lançamento ou em atualizações posteriores, poderá ser comprometido. A modelagem de sistemas é um método, que por múltiplas ferramentes, permite ajudar em diversas funções no planejamento e entendimento de um projeto. Ela permite criar uma estrutura e estabelecer o comportamento do *software* em um tempo hábil muito menor do que o necessário para montar a aplicação.

Existem múltiplas linguagens e ferramentas para a modelagem de sistemas e diferentes ferramentas que podem ser escolhidas conforme a necessidade do projeto. Naturalmente a diversidade de opções também pode se tornar um empecilho em alguns casos, considerando que diferentes ferramentas podem utilizar diferentes formatos de armazenamento das informações contidas nos modelos, mas existem, no entanto, algumas outras ferramentas que ajudam na conversão de modelos para diversos formatos.

Esse projeto pretende criar uma plataforma que facilite a utilização dessas ferramentas de conversão, permitindo o uso delas em um único ambiente sem precisar lidar com a instalação e configuração das mesmas. Além disso, ele também visa permitir o armazenamento e organização de modelos nesse mesmo ambiente.

Este trabalho foi feito para descrever as decisões e ações feitas na busca de atingir o objetivo do projeto e quais resultados foram alcançados ao término dele. Para isso ele foi dividido em quatro seções. Em uma primeira seção é descrito o embasamento teórico utilizado para justificar as escolhas do projeto. Em seguida é explicitado a metodologia usada, as ferramentas e arquiteturas escolhidas e suas justificativas. Após as seções anteriores é discutido os resultados obtidos e para finalizar a conclusão obtida é sintetizada.

2 Fundamentação teórica

A escolha da arquitetura tem um impacto profundo no desenvolvimento de software dessa forma é preciso analisar qual arquitetura será mais vantajosa de ser usada para cada aplicação. Algumas delas presentes no mercado são as arquiteturas monolíticas, de microsserviços e orientada a serviços (SOA).

Em uma aplicação monolítica um único código é usado para todos os componentes da aplicação e toda a aplicação está baseada em um único *design*, não permitindo a implantação de componentes individuais (PACHGHARE, 2016).

2.1 Microsserviços

O acoplamento de múltiplos componentes de forma mais solta é muito importante para assegurar a flexibilidade do software. Isso permite um sistema leve e com alta manutenibilidade. Microsserviços permite garantir isso e, simultaneamente, facilitar a integração com novas tecnologias (SELVAKUMAR, 2019).

A arquitetura de microsserviços consiste na decomposição da aplicação em componente menores e modulares. Esses componentes trabalham juntos para exercer a função do todo, oferecendo maior escalabilidade (BANICA et al., 2017).

Esse modelo pode tornar aplicação mais flexível. Fritzsch et al. (2019) afirmam que sistemas monolíticos podem se tornar grandes e complexos após anos ou décadas de crescimento e na busca de maior manutenibilidade e escalabilidade, várias empresas migraram para o uso de microsserviços. Por exemplo,

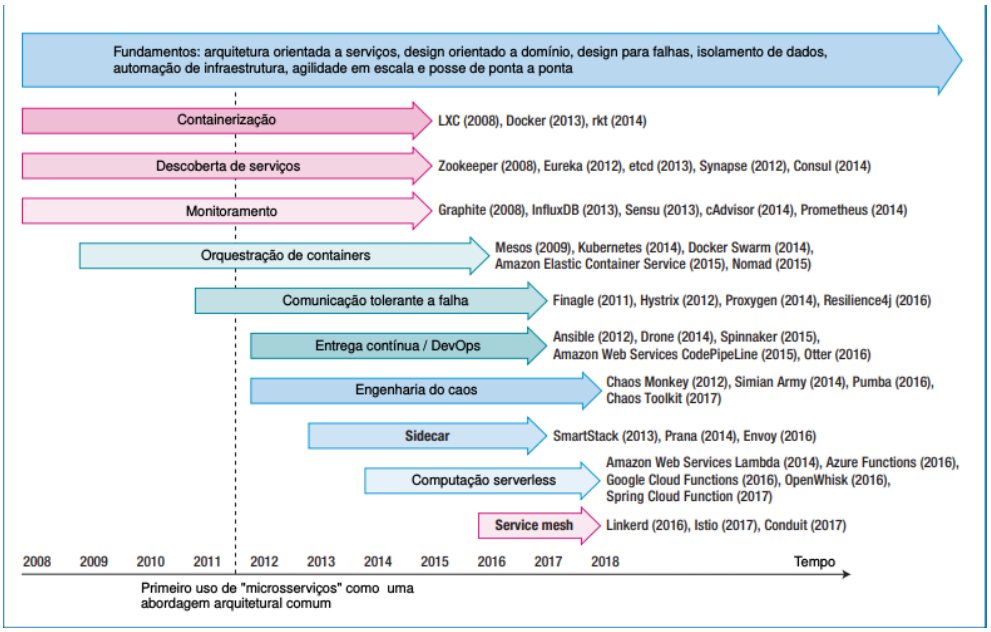
> "Várias corporações proeminentes migraram para microsserviços. Netflix, eBay, Amazon, Twitter, Paypal, Sound cloud são algumas delas. Elas experimentaram imensos benefícios em manutenibilidade dos seus sistemas e desenvolvimento continue e integração" (selvakumarinternet, tradução nossa).

Além da comparação entre uma arquitetura de microsserviços e uma monolítica clássica é interessante constatar que essa não é a única arquitetura que constitui na divisão da aplicação em partes menores.

Cerny, Donahoo e Trnka (2018) explicam que tanto SOA quanto microsserviços propõem a decomposição da aplicação em múltiplos serviços que cooperam para prover a funcionalidade do todo. SOA sugere a decomposição em serviços simples enfatizando o serviço de integração, em contrapartida, microsserviços sugere a decomposição em serviços inteligentes com mecanismos de roteamento simples.

Jamshidi et al. (2018) classifica a evolução tecnológica, que permitiu a criação e difusão de microsserviços, em dez ondas tecnológicas, sendo que as cinco primeiras delas já existiam antes de o termo microsserviço ter sido cunhado. Elas foram as *containers*, serviços de descoberta de dispositivos, serviços de monitoramento, orquestração de *containers*, bibliotecas de comunicação tolerantes a falhas e latência, *continuous-delivery*, engenharia de caos, *Sidecar*, computação *serverless* e *mesh*. Essa evolução está ilustrada na Figura 1.

Figura 1 – Linha do tempo de tecnologias para microsserviços



Fonte: Jamshidi et al. (2018)

2.2 Comunicação entre microsserviços

Para a comunicação entre os serviços é possível escolher entre comunicação síncrona e assíncrona. A primeira consiste em um dos serviços mandar uma requisição para outro e esperar até este processar o resultado e retornar uma resposta. Um exemplo de ferramenta comumente usada em aplicações é REST API. A segunda forma de comunicação é feita através de um *message broker* intermediário que coordenada a comunicação entre os processos, dessa forma o serviço não espera por uma resposta direta (SHAFABAKHSH; LAGERSTRÖM; HACKS, 2020). Existem múltiplas ferramentas que podem ser usadas para comunicação assíncrona entre serviços, entre eles estão MQTT, AMQP e Kafka.

MQTT é um protocolo de transmissão de dados baseado em *publisher* e *consumer*, ele é um protocolo usado na camada de aplicação do conjunto de protocolos TCP/IP. Nele as partes que se comunicam não precisam conhecer a identidade uma da outra. Nesse modelo de comunicação são considerados dois tipos de agentes: os clientes e o *broker*, sendo os clientes chamados *publisher* ou *consumer*. Um *publisher* pode enviar mensagens, enquanto um *consumer* pode solicitar as informações relacionadas as mensagens e o *broker* reconhece e transmite mensagens entre clientes associados a elas (KASHYAP; SHARMA; GUPTA, 2018).

AMQP é um protocolo de comunicação entre máquinas que suporta tanto *request/response* quanto *publish/subscribe*. O protocolo oferece filas confiáveis, *publish/subscribe* baseado em tópicos, roteamento e transações flexíveis. Ele permite a troca de mensagens em várias formas que incluem mensagens diretas, *fan-out* e por tópico. Confiabilidade é uma de suas principais características e oferece dois níveis preliminares de qualidade de serviço (NAIK, 2017).

Kafka é um sistema de mensagens distribuído. Ele permite distribuir fluxos de mensagens entre múltiplos sub fluxos e pode ser escalável permitindo o uso de múltiplos *brokers*. Diferente da maioria, os *Brokers* Kafka são *stateless*, isso significa que a informação se o dado foi consumido ou não por algum serviço não é mantido no *broker*, mas deve ser mantido no serviço que o utiliza, isso traz vantagens e desvantagens, por um lado o *broker* não sabe se todos os destinatários receberam a mensagem, dificultando a remoção, por outro lado, isso traz mais flexibilidade aos consumidores dos dados(KREPS et al., 2011).

2.3 Tipos de servidores

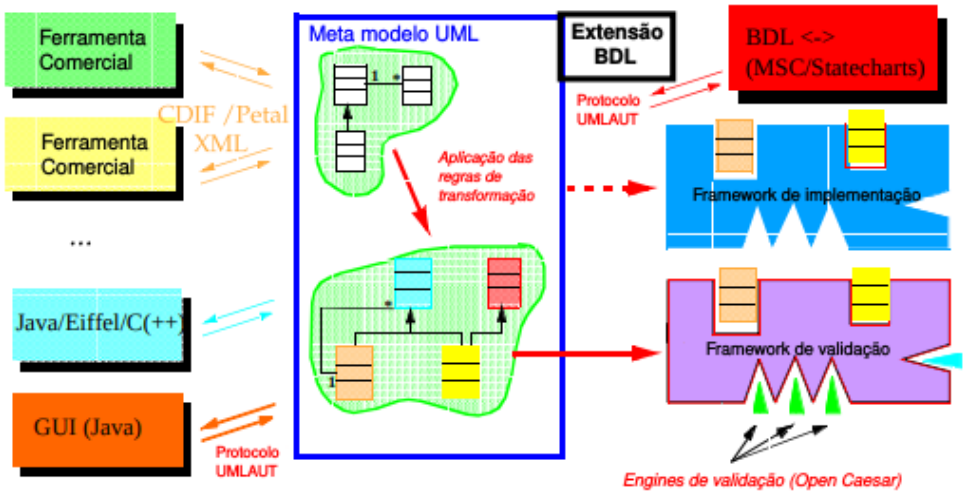
Um servidor pode ser considerado *stateless* ou *stateful*. Segundo Alonso (2015), ser *stateless* significa que cada requisição de um cliente deve conter toda informação necessária para que o servidor consiga interpretar a requisição. Usando esse paradigma dados referentes a sessão devem ser retornados ao cliente e o estado da sessão deve ser mantido por este.

Alonso (2015) afirma também que restringir o sistema a uso de *stateless* melhora a visibilidade, confiabilidade e escalabilidade, porém reduz o desempenho da rede e controle do servidor. Visibilidade é ganha, pois não a necessidade de traçar as requisições anteriores para entender todos os aspectos da requisição atual. Confiabilidade é aperfeiçoada, pois esse protocolo facilita a recuperação de falhas parciais. Não precisar armazenar o estado atual facilita a escalabilidade.

2.4 Trabalhos relacionados

O projeto UMLAUT é um *framework* de transformação de modelos que permite aplicar transformações complexas em modelos UML, na Figura 2 está representado a sua arquitetura. Eles englobam a transformação entre formatos como CDIF e MDL, geração de código em linguagens como Java e Eiffel e transformação de modelos dedicados à validação de sistemas reativos distribuídos (HO et al., 1999).

Figura 2 – Arquitetura do projeto UMLAUT



Fonte: Ho et al. (1999)

A aplicação CROSSMINER busca auxiliar o desenvolvimento de projetos de código aberto através da extensão de um projeto anterior OSSMETER. Nele feito a integração de componentes legados através de uma *API REST* atuando como um *proxy* reverso, essa solução vem de um padrão utilizado em aplicações de microsserviços (BOUDEFFA et al., 2019).

Matthias Wauer1 e Axel-Cyrille Ngonga Ngomo descrevem GEISER, um projeto de uso de dados de sensores e geoespaciais com intuito de uso em múltiplas aplicações que incluem *geomarketing*. Por exemplo, alertando um restaurante local que uma banda estará tocando próximo, estacionamento inteligente, guiando o usuário para lugares com maior probabilidade de vagas livres e manutenção preditiva.

A meta principal desse projeto é permitir o processamento em massa de dados de sensores e geoespaciais permitindo o uso dos mesmos para aplicações diversas. A arquitetura descrita, ilustrada pela Figura 3, utiliza microsserviços e a ferramenta RabbitMQ usando o protocolo AMQP para comunicação entre serviços e com acesso a aplicação usando uma API REST (WAUER; SHERIF; NGOMO, 2018).

3 Materiais e métodos

As ferramentas e estruturas escolhidas para compor uma aplicação podem determinar o cumprimento ou não de qualquer objetivo estabelecido no projeto. Considerando isso é vital o entendimento e planejamento antes de iniciar o desenvolvimento de qualquer sistema. Nessa seção do trabalho é descrito o planejamento e escolhas feitas para a plataforma.

Figura 3 – Arquitetura do projeto Geiser

Fonte: Wauer, Sherif e Ngomo (2018)

3.1 Metodologia

Antes de iniciar o desenvolvimento da aplicação é necessário entender qual o objetivo do software e o que ele teria que ter para cumpri-los. A partir do problema que se busca ser corrigido foi elaborado um conjunto de requisitos funcionais e não funcionais permitindo o melhor entendimento em relação ao que o *software* deverá ou não conseguir fazer.

Após essa etapa foi iniciada a elaboração da arquitetura do software sendo essencial para determinar quais ferramentas e padrões serão utilizados no decorrer do desenvolvimento. Nela, com base nos requisitos dele, foi determinado quais componentes deveriam existir na aplicação e de que forma esses componentes estariam conectados.

Após entender os componentes que devem compor o sistema foi preciso determinar de que forma um usuário poderá se comunicar com eles e de que forma eles devem interagir entre si.

3.2 Definição de problema e requisitos

Na modelagem de sistemas, há a necessidade do uso, intercambiável, de múltiplos formatos de arquivos de modelos em ambientes variados. Um exemplo deles é o ambiente acadêmico, nele a interação entre múltiplos alunos e professores pode resultar em alto número de ferramentas utilizadas, cada uma podendo ter seu próprio formato de armazenamento e representação.

A multitude dos usos de ferramentas de modelagem que se comunicam entre si já pode ser alcançada através do uso de conversores de arquivos, porém isso cria um aumento de complexidade que pode não ser tolerada, levando a instituição de ferramentas e formatos padrões que diminuem a pluralidade e liberdade dos usuários presentes no ambiente em questão.

Esta aplicação busca facilitar esse processo de forma que a diversidade de ferramentas possam ser mantida sem aumentar, significativamente, a complexidade do processo e também busca facilitar o gerenciamento dos modelos, simplificando seu gerenciamento de armazenamento e compartilhamento.

Buscando solucionar os problemas considerados e atingir os objetivos finais foi feito o levantamento dos requisitos funcionais e não funcionais. Dessa forma foi possível entender o que a aplicação, após finalizada, busca atender e permitindo assim decidir quais ferramentas podem melhor acatar as necessidades do projeto.

Os requisitos estabelecidos consideram uma plataforma de conversão de modelos e armazenamento de modelos. Permitindo organização de modelos em grupos e projetos, além de definir permissões de acesso para diferentes usuários.

Os requisitos também incluem a proposta de uma aplicação gráfica que deverá poder se comunicar com a aplicação definida nesse trabalho, porém ela não será tratada nesse trabalho.

3.3 Arquitetura

Como é mostrada na Figura 4, a aplicação disponibiliza uma *API* externa que será usada para a comunicação diretamente com o usuário ou por uma *interface* gráfica externa. Em adição a ela, o sistema é composto de um gerenciador de usuários e permissões, um gerenciador de serviços, um gerenciador de projetos, um gerenciador de modelos, um barramento de mensagens e múltiplos serviços de transformação de modelos.

As setas saindo de *Cliente* para a *Interface* e *API* na imagem representam a possibilidade de um usuário acessar a aplicação através da *API* diretamente ou utilizando alguma outra aplicação como intermediário. Devido a isso, embora o projeto atual não disponibilize uma *interface*

Como pode ser observado na figura a comunicação entre a maioria dos processos é intermediada por um barramento de mensagens, sendo que no caso dos serviços de transformação de modelos, por ter um número variável de aplicações, precisam ser descobertas após o início da aplicação pelo gerenciador de serviços.

Um microsserviço que se destaca na imagem é gerenciador de usuário e permissões, ele estabelece uma comunicação direta com *API*. Ela é utilizada para garantir a validade das requisições que chagam a *API*.

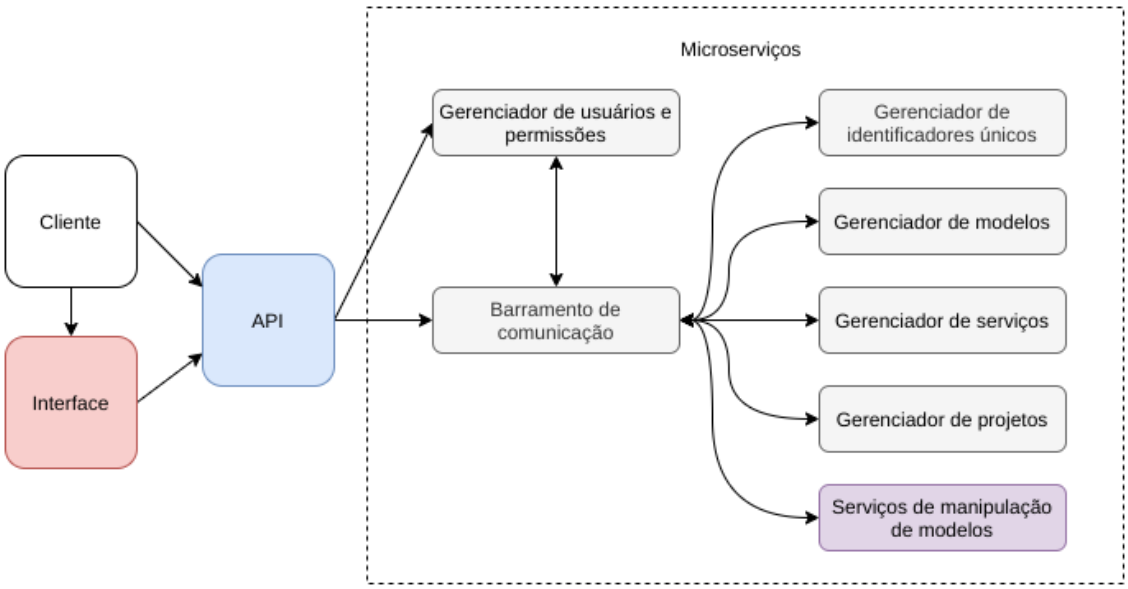
3.3.1 Definição de API

Com base nos requisitos do software foi levantado a lista de quais recursos seriam expostos ao usuário e quais informações estariam contidas neles, utilizando esse conhecimento foi possível definir uma API usando o paradigma REST e o formato JSON para representar todos os recursos que não são arquivos.

O gerenciamento de sessão é feito através de um *token* de autenticação utilizando o formato JWT. Ele poderá ser obtido através de uma rota de autenticação e deverá ser preservado no computador do usuário e enviado junto ao cabeçalho de todas as requisições.

Para descrever a API foi utilizado a especificação OpenAPI na versão 3.0, com ela é possível descrever as rotas de forma padronizada independente da linguagem utilizada

Figura 4 – Arquitetura



Fonte: elaborado pelo autor (2022)

Tabela 1 – Especificação dos métodos de requisição HTTP

Nome do método	Descrição
GET	Transfere a representação atual do recurso alvo
HEAD	Mesmo que GET, porém não transfere o conteúdo da resposta
POST	Executa processamento específico do recurso no conteúdo da requisição
PUT	Substitui toda representação atual do recurso alvo pelo conteúdo da requisição
DELETE	Remove todas as representações do recurso alvo
CONNECT	Estabelece um túnel ao servidor identificado pelo recurso alvo
OPTIONS	Descreve as opções de comunicação para o recurso alvo
TRACE	Executa um teste de retorno pelo caminho até recurso alvo

Fonte: adaptado de Nottingham (2022)

na implementação, além de poder ser utilizada para geração da documentação que pode ser disponibilizada ao usuário.

O protocolo HTTP possui múltiplos métodos de requisição, como pode ser observado na Figura 1 que atribuem diferentes sentidos semânticos a requisição.

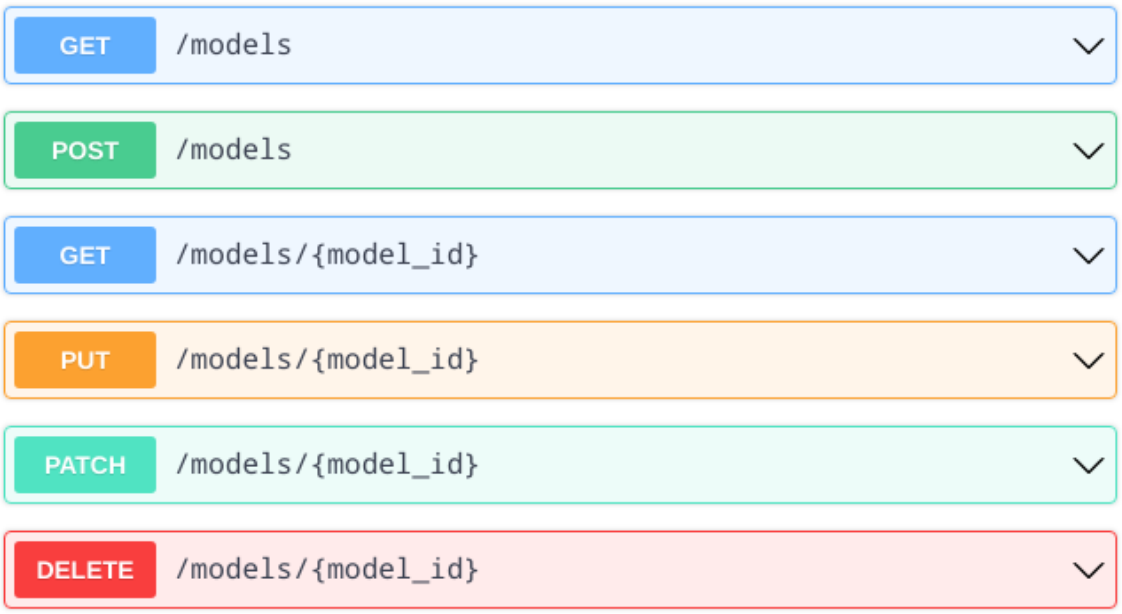
Foram atribuídos, a eles, diferentes significados no escopo da utilização da API. Foi definido que o método *GET* representa requisição de informação, *POST* representando a inserção de recurso, *PUT* indica a substituição total, *PATCH* serve para a atualização parcial dos dados e *DELETE* seria usado para excluir.

Baseando no padrão REST as rotas da *interface* são organizadas com base no conceito de recursos, por exemplo, um modelo é tratado como um recurso e duas rotas são associadas a ele. A primeira delas, utilizando o formato */models/*, contém a representação de coleção de modelos, dessa forma o método *GET* resulta no retorno de múltiplos e o método *POST* na criação de um único. A segunda, utilizando o formato */models/<model_id>*,

representa um único modelo e o método de *GET* retorna informações dele, o *PUT* substitui todos os dados dele, o *PATCH* substitui parte das informações dele e o *DELETE* o deleta.

A Figura 5 mostra a visualização das rotas de modelos através de documentação gerada pelo software Swagger. A Figura 6 mostra os parâmetros para o rota */models* conforme documentação gerada.

Figura 5 – Rotas de modelos na API



Fonte: elaborado pelo autor (2022)

Além dos recursos, foi utilizado também a representação de ações, isso foi utilizado para as transformações. Nesse cenário a ação é tratada de forma similar a um recurso sendo definido que deve-se usar o método *PUT* caso a ação não altere o estado da aplicação e *POST* caso ela o faça.

A Figura 7 ilustra um exemplo de uso da *API*. Se usuário quiser obter as informações de um modelo, ele deverá, conforme descrito anteriormente, utilizar a rota */models/<model_id>* e fazer uma requisição com método *GET*, substituindo *<model_id>* pela identificação do modelo desejado.

Para que a requisição tenha exito é preciso ter acesso ao modelo, caso o modelo não seja público isso irá requerer a aquisição de um *token JWT*, como mostrado na Figura 7. Para obter o *token* é preciso uma requisição *POST* a rota */auth*. A utilização de *POST* para essa rota implica na criação de um novo *token*. O *token* deve ser encaminhado junto as requisições através do cabeçalho *HTTP Authorization*, passando o valor *Bearer <token>*, substituindo *<token>* pelo valor correspondente.

Na Figura 7 foi mostrada uma requisição a rota */models* antes de acessar */models/<model_id>* com intuito de representar que o *id* associado ao modelo poderá ser encontrado utilizando requisições para rotas que retornem modelos.

Figura 6 – Parâmetro do método *GET* de */models*

Fonte: elaborado pelo autor (2022)

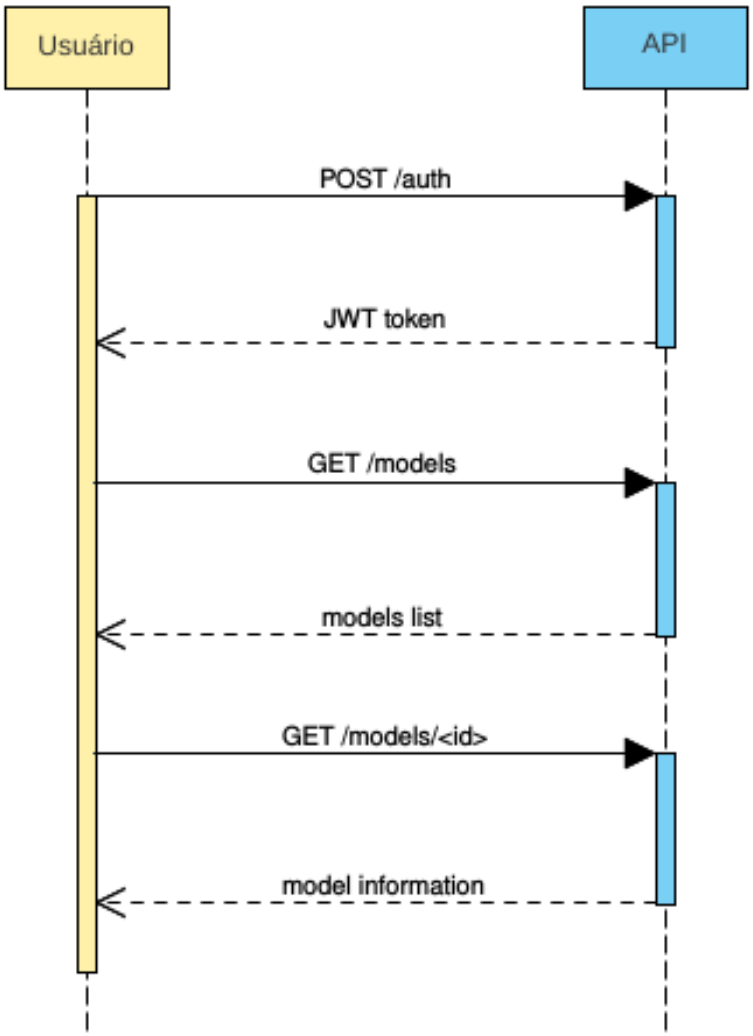
3.3.2 Serviços

A arquitetura de microsserviços foi escolhida para essa aplicação, dessa forma ela abrigará diversos serviços. Para os serviços criados para ela foi escolhida a linguagem de programação Python. Alguns serviços também são utilizados, entre eles MySQL como um banco de dados *SQL*, OpenStack Swift usado para armazenamento de arquivos e RabbitMQ como um barramento de comunicação.

O gerenciador de *tokens* de registros desempenha um papel simples, nele é feita a criação de *tokens* de identificação que visam ser usados para identificar registros tanto para aplicações internas quanto para outros serviços que fazem parte da aplicação, usando um identificador único para aquele recurso que não colide com nenhum outro presente no sistema.

O gerenciador de usuários e permissões é responsável pelo armazenamento das informações associadas aos usuários, através dele é feita a geração e checagem de autenti-

Figura 7 – Exemplo de uso da API



Fonte: elaborado pelo autor (2022)

cidade do *token* JWT. Esse serviço usa um esquema no MySQL dedicado a ela, porém não utiliza o barramento para comunicação, disponibilizando apenas comunicação síncrona através de uma API interna.

Está presente um gerenciador de arquivos servindo de ponte entre o serviço de armazenamento de arquivos escolhido e outros serviços do sistema. Ele usa armazenamento SQL para lidar com a indexação dos arquivos e lidar com a contagem de referências.

O gerenciador de projetos é responsável pelo gerenciamento de todos os dados da estrutura dos grupos, projetos e modelos dos usuários. Nele são geridas a relação de pertencimento desses dados, por exemplo, qual grupo contém o projeto e qual o dono do grupo. Esse serviço também gerencia permissões de nível de grupo, barrando requisições de usuários que não possam ter acesso a um grupo, projeto ou modelo especifico com base nas configurações dos mesmos e das informações advindas do gerenciador de usuários e

permissões. As informações são guardadas através do serviço SQL escolhido.

O serviço de *API* serve como uma *interface* para usuários e aplicações externas, servindo como entrada e saída para a aplicação total. Qualquer agente externo deverá poder apenas se comunicar com esse serviço, não sendo possível o acesso direto a qualquer outro.

Cada função de transformação de modelos é lançada como um serviço separado que pode ser integrado ao restante das aplicações através do barramento de comunicação. Elas são organizadas através do gerenciador de serviços que irá conter a fila de requisições de transformação e irá organizar e repassar para o serviço de transformação correspondente qual arquivo ele deverá transformar.

3.3.3 Comunicação entre serviços

Os serviços se comunicam entre si indiretamente por um *broker* RabbitMQ. Esta ferramenta suporta múltiplas formas de comunicação, dentre elas AMQP e MQTT, sendo a escolhida AMQP. Apenas a comunicação entre a API e o gerenciador de usuários e permissões utiliza outro meio de comunicação, o HTTP.

A escolha do protocolo se deve primeiramente pela opção do uso de comunicação assíncrona para a comunicação entre os diversos serviços. Essa escolha foi feita para lidar mais facilmente com a multitude de serviços que compõem a aplicação, considerando a necessidade de extensão do número de serviços conforme o aumento do número de opções de transformação e ter que lidar com a possibilidade de falha deles.

4 Resultados e discussões

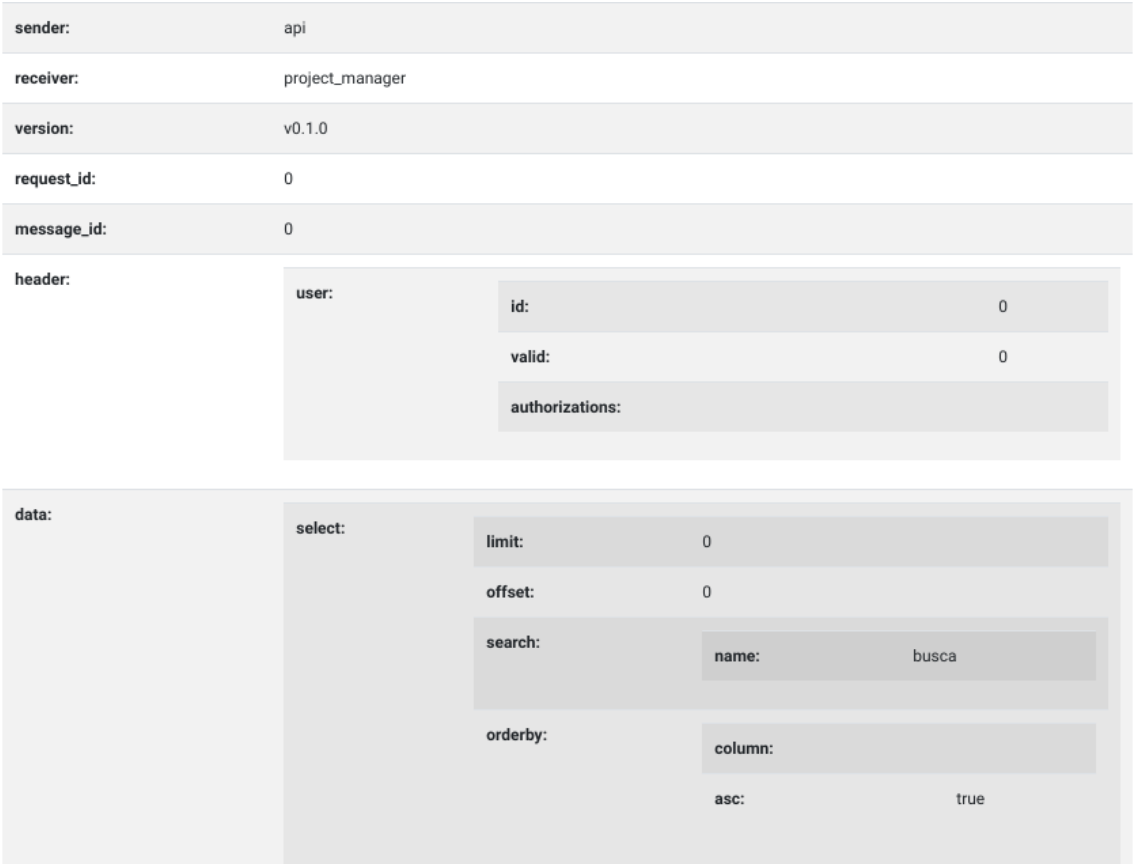
Após o planejamento a primeira etapa no desenvolvimento do projeto foi encontrar e testar o serviço de comunicação. Isso foi possível através de uma imagem *docker* (RabbitMQ), que pode ser utilizada através do comando descrito na Figura 9, dessa forma simplificando essa etapa do trabalho, ela também fornece uma *interface* que auxilia em testes durante o desenvolvimento.

A conexão com o servidor de banco de dados foi feita utilizando a biblioteca *SQLAlchemy* em *Python*, usando ela é possível descrever as tabelas utilizadas utilizando o conceito *ORM*, exemplificado na Figura 10. Ele consiste em uma técnica de abstração do *SQL*, uma linguagem utilizada para realizar operações em bancos de dados relacionais, tornando mais simples a interação com os registros. Essa descrição da estrutura das tabelas também é usada pelo sistema de versionamento do banco de dados utilizado, chamado *alembic*, que permite a criação e atualização delas.

A estruturação do código, ilustrada na Figura 11, seguiu uma organização similar para todos os serviços, cada um deles é separado um diretório, dentro dele a pasta chamada *src* contém o código-fonte do programa, o arquivo *requirements.txt* contém as dependências que podem ser instaladas através do gerenciador de pacotes *Pip*, o arquivo *config.py* guarda as configurações e a pasta *migrations* armazena as informações necessárias para o gerenciador de versão do banco de dados.

Na pasta *src* é possível encontrar uma maior diversidade de conteúdo que varia entre os microsserviços, porém é comum estar presente um arquivo *___main___.py* usado para inicializar o programa e uma pasta *models* onde os modelos *ORM* usados estão

Figura 8 – Estrutura da mensagem entre serviços



Fonte: elaborado pelo autor (2022)

Figura 9 – Comando utilizado para usar o docker do RabbitMQ

```
docker pull rabbitmq:3.9-management
docker run -d -it --restart unless-stopped --name rabbitmq -p 5672:5672 \
    -p 15672:15672 rabbitmq:3.9-management
```

Fonte: elaborado pelo autor (2022)

estruturados. Assim pode-se encontrar o diretório *routes* para os serviços que exponham rotas HTTP, nele são contidos os códigos chamados quando essas rotas são requisitadas.

A arquitetura de microsserviços permitiu que o desenvolvimento de cada serviço fosse criado de forma individual sem a necessidade do funcionamento completo do todo. E por meio da interação com o barramento de comunicação foi possível emular o envio de mensagens de outros sistemas facilitando o desenvolvimento ao impedir que o mau funcionamento de um deles afete aquele que está sendo desenvolvido e permitindo buscar individualmente em cada um deles caso uma falha no sistema seja detectada. O mesmo comportamento também poderia ser aplicado para as comunicações síncronas que fazem

Figura 10 – Exemplo de representação *ORM*

```python
class File(Base):
    __tablename__ = 'files'

    id = Column(Integer(), primary_key=True)
    filename = Column(String(64), nullable=False)
    content_type = Column(String(255), nullable=False)
    id_token = Column(String(20), nullable=False)
    count = Column(Integer(), nullable=False, server_default=text('1'))
    deleted = Column(Enum('true', 'false'), nullable=False,
                     server_default=text("'false'"))
    created_at = Column(DateTime, nullable=False,
                        server_default=text('current_timestamp()'))
    updated_at = Column(DateTime, onupdate=text('current_timestamp()'))
    storage_idx = Column(ForeignKey('storages.id'), index=True, nullable=True)

    storage = relationship('Storage')

    def put(self, content):
        self.storage.getInterface().put(self.filename, content)

    def read(self):
        return self.storage.getInterface().get(self.filename)
```

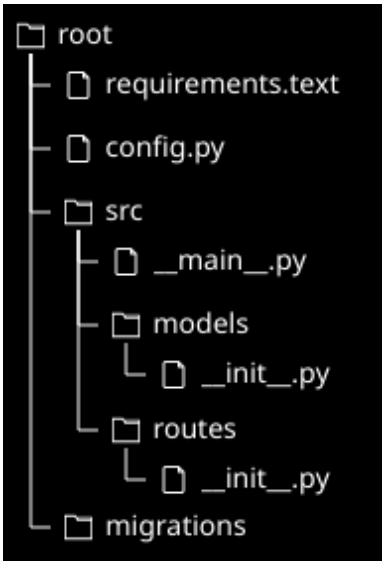
Fonte: elaborado pelo autor (2022)

parte do projeto através de um servidor *dummy*, porém isso foi julgado não ser necessário para a aplicação.

O gerenciador de *tokens* de registros, que está localizado na pasta *unique_tokens*, se mostrou um serviço com menor complexidade de comunicação, através dele é possível fazer apenas duas operações que são: através das informações do registro buscar o *token* correspondente, nesse caso será criado caso ele não exista, e através do *token* obter os dados do registro. Eles são gerados de forma aleatória utilizando a biblioteca *secrets* e para garantir a unicidade deles é verificado se já está em uso, caso já esteja, um novo é gerado até ser encontrado um que esteja livre. As informações de um registro utilizadas pelo gerador são o dono que representa qual serviço detém o registro, o tipo que é um identificador de qual categoria de conteúdo está atrelado a ele, que é, em sua maioria, uma indicação da tabela onde ele está e um identificador local que contém como o serviço o identifica, sendo na prática, normalmente, o *id* no banco de dados.

O diretório *storage* contém o gerenciador de arquivos. Ele gerencia a comunicação com o serviço de armazenamento de arquivos, para isso ele contém uma tabela com as configurações para realizar a comunicação com eles, essa configuração é utilizada para selecionar uma classe de *interface* para com o serviço de armazenamento adequado e

Figura 11 – Estrutura padrão de diretórios



Fonte: elaborado pelo autor (2022)

os parâmetros passados a ela. No momento são suportadas apenas o uso de *Swift* e o sistema de arquivos local, sendo o primeiro a forma recomendada para o uso em produção. As operações permitidas são de inserção, leitura e deleção de arquivos, não permitindo modificar um arquivo já existente.

Através do gerenciador de permissões é possível criar, deletar e alterar usuários, além de gerar e verificar *tokens JWT* usados para validação de autenticidade de requisições. Para manipulá-los, foi utilizado a biblioteca *PyJWT*, a criação é ilustrada na Figura 12. Para isso ele expõe uma *API* de uso interno seguindo o padrão *REST*. Para garantir a segurança do sistema é importante que, ao disponibilizar em produção, a comunicação entre a *API* externa sejam seguras e ele responda apenas requisições provenientes dela, alguns fatores que podem ajudar nisso é garantir que somente a máquina que contém ela possa fazer requisições a ele ou estarem localizados na mesma máquina e que a chave e usuário da autenticação *Basic* utilizada entre os dois tenha as devidas permissões.

A implementação dos diferentes componentes foi iniciada de forma separada, porém a integração deles no atual estado de desenvolvimento é extremamente limitada, dessa forma, atualmente, não é possível utilizar das funções do software, por exceção da autenticação, através da API.

4.1 Limitações

Uma limitação do projeto atual é não lidar de forma otimizada com a atualização de modelos já existentes. É possível, no entanto, em uma versão futura, por codificação delta minimizar esse problema, porém não foi considerado em nenhuma etapa de planejamento ou desenvolvimento do trabalho.

Segundo a arquitetura atual, para adicionar novas conversões de arquivos é necessário ter acesso direto ao barramento de comunicação. Devido a isso a adição de novos serviços fica a cargo de usuários ou ferramentas com acesso ao barramento de comunicação, sem ser possível fazer a adição por meio do próprio serviço.

Figura 12 – Código de criação de *token JWT*

```
class File(Base):
    __tablename__ = 'files'

    id = Column(Integer(), primary_key=True)
    filename = Column(String(64), nullable=False)
    content_type = Column(String(255), nullable=False)
    id_token = Column(String(20), nullable=False)
    count = Column(Integer(), nullable=False, server_default=text('1'))
    deleted = Column(Enum('true', 'false'), nullable=False,
                     server_default=text("'false'"))
    created_at = Column(DateTime, nullable=False,
                        server_default=text('current_timestamp()'))
    updated_at = Column(DateTime, onupdate=text('current_timestamp()'))
    storage_idx = Column(ForeignKey('storages.id'), index=True, nullable=True)

    storage = relationship('Storage')

    def put(self, content):
        self.storage.getInterface().put(self.filename, content)

    def read(self):
        return self.storage.getInterface().get(self.filename)
```

Fonte: elaborado pelo autor (2022)

5 Conclusão

No desenvolvimento de software, principalmente em grandes projetos, o planejamento é essencial para obter resultados satisfatórios, devido a isso o maior enfoque no desenvolvimento do presente trabalho foi buscar entender a melhor forma de atender os objetivos estabelecidos.

Para isso foi necessário estabelecer primeiramente quais requisitos a aplicação deveria cumprir para então determinar quais meios e ferramentas poderiam ser utilizadas para assegurar o cumprimento deles.

Também foi estabelecido a definição da API, estabelecendo uma via de entrada e saída da aplicação que pode ser usada para extensão da mesma para disponibilizar uma interface gráfica com o usuário. No período estabelecido foi possível também iniciar a execução da implementação do sistema, estabelecendo a estrutura base do projeto e experimentação de parte das ferramentes consideradas.

Dessa forma, o desenvolvimento desse trabalho resultou na realização de um estudo das necessidades da plataforma e quais ferramentas e métodos podem ser usados para atendê-las, na elaboração de um padrão da *interface* de comunicação externa e interna e

no início do desenvolvimento dos serviços.

Referências

ALONSO, F. S. Development of a restful api: hateoas & driven api. Turun ammattikorkeakoulu, 2015. Citado na página 4.

BANICA, L. et al. Leveraging the microservice architecture for next-generation iot applications. *Scientific Bulletin–Economic Sciences*, v. 16, n. 2, p. 26–32, 2017. Citado na página 2.

BOUDEFFA, A. et al. Integrating and deploying heterogeneous components by means of a microservices architecture in the crossminer project. In: *STAF (Co-Located Events)*. [S.l.: s.n.], 2019. p. 67–72. Citado na página 5.

CERNY, T.; DONAHOO, M. J.; TRNKA, M. Contextual understanding of microservice architecture: current and future directions. *ACM SIGAPP Applied Computing Review*, ACM New York, NY, USA, v. 17, n. 4, p. 29–45, 2018. Citado na página 3.

FRITZSCH, J. et al. Microservices migration in industry: intentions, strategies, and challenges. In: IEEE. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2019. p. 481–490. Citado na página 3.

HO, W. M. et al. Umlaut: an extendible uml transformation framework. In: IEEE. *14th IEEE International Conference on Automated Software Engineering*. [S.l.], 1999. p. 275–278. Citado na página 5.

JAMSHIDI, P. et al. Microservices: The journey so far and challenges ahead. *IEEE Software*, v. 35, n. 3, p. 24–35, May 2018. ISSN 1937-4194. Citado na página 3.

KASHYAP, M.; SHARMA, V.; GUPTA, N. Taking mqtt and nodemcu to iot: communication in internet of things. *Procedia computer science*, Elsevier, v. 132, p. 1611–1618, 2018. Citado na página 4.

KREPS, J. et al. Kafka: A distributed messaging system for log processing. In: *Proceedings of the NetDB*. [S.l.: s.n.], 2011. v. 11, p. 1–7. Citado na página 4.

NAIK, N. Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. In: IEEE. *2017 IEEE international systems engineering symposium (ISSE)*. [S.l.], 2017. p. 1–7. Citado na página 4.

NOTTINGHAM, M. Http semantics. 2022. Citado na página 8.

PACHGHARE, V. K. Microservices architecture for cloud computing. *architecture*, v. 3, p. 4, 2016. Citado na página 2.

SELVAKUMAR, G. Internet of things and microservices architecture for modern supply chains. 2019. Citado na página 2.

SHAFABAKHSH, B.; LAGERSTRÖM, R.; HACKS, S. Evaluating the impact of inter process communication in microservice architectures. In: *QuASoQ@ APSEC*. [S.l.: s.n.], 2020. p. 55–63. Citado na página 4.

WAUER, M.; SHERIF, M. A.; NGOMO, A.-C. N. Towards a semantic message-driven microservice platform for geospatial and sensor data. In: *GeoLD-QuWeDa@ ESWC*. [S.l.: s.n.], 2018. p. 47–58. Citado 2 vezes nas páginas 5 e 6.

APÊNDICE A - DOCUMENTO DE REQUISITOS DE SOFTWARE

SOFTWARE PARA CONVERSÃO DE MODELOS
DEFINIÇÃO DE REQUISITOS

Versão *1.0*

27/05/2022

HISTÓRICO DE VERSÕES

Versão	Implementada por	Data da Revisão	Motivo da Revisão
1.0	Samuel Possamai	19/10/2021	Definição Inicial de Requisitos
1.0	Danilo Silva	29/11/2021	Padronização Inicial de Requisitos
1.0	Danilo Silva	27/05/2022	Revisão dos Requisitos e Inserção da Introdução

Sumário

1. INTRODUÇÃO

1.1. PROPÓSITO DA DEFINIÇÃO DE REQUISITOS FUNCIONAIS

Requisitos funcionais tem o propósito de listar os fatores básicos de atendimento do escopo do produto e suas ramificações imediatas, buscando visualizar o essencial para o funcionamento legal, seguro, viável, e útil do produto.

1.2. PROPÓSITO DA DEFINIÇÃO DE REQUISITOS NÃO-FUNCIONAIS

Requisitos não-funcionais tem o propósito de: expandir as ramificações dos requisitos funcionais, definindo os fatores constituintes e/ou apoiadores do essencial; definir a moldura geral de fatores secundários que possuem como objetivo expandir a gama de funcionalidades do produto; e construir a base da existência do funcionamento mínimo do produto.

2. RESUMO DE REQUISITOS DE PRODUTO

As características gerais do produto estão baseadas em um software capaz de cadastrar usuários suportando diferentes níveis de acesso, onde a aplicação deve ser capaz de suportar o carregamento, download, exclusão e conversão de diferentes tipos de modelos. Para ser possível a conversão de modelos o usuário deve utilizar sequências de conversão, de modo que, seja indicado o tipo de modelo de entrada e o tipo de modelo de saída que é suportado na plataforma. Modelos e Sequências de conversão são agrupadas dentro de um projeto, sendo que um projeto, modelo ou sequência de conversão pode ser criado, editado, excluído ou copiado. Projetos são públicos ou privados e podem ser agrupados dentro de grupos onde grupos podem ser compartilhados com outros usuários cadastrados na plataforma.

3. REQUISITOS FUNCIONAIS

3.1. REQUISITOS FUNCIONAIS DE USO

- **RFUS-1.1 -** Armazenar modelos no servidor;
- **RFUS-1.2 -** Permitir transformação entre os modelos suportados;
- **RFUS-1.3 -** Modelos podem ser agrupados para diferentes grupos;
- **RFUS-1.4 -** Permitir grupos dentro de grupos;
- **RFUS-1.5 -** O sistema permite registro de "Usuário individual";
- **RFUS-1.6 -** As restrições de armazenamento e transformação são por projeto;
- **RFUS-1.7 -** Um projeto é um conjunto de modelos pertencente a um grupo ou usuário restrito as operações permitidas;
- **RFUS-1.8 -** Um projeto pode ser criado a partir de outro projeto desde que o usuário ou grupo tenha acesso a ele;
- **RFUS-1.9 -** Um projeto pode ser público ou privado, se for privado e pertencer ao um grupo ele poderá ser acessado pelos membros do grupo e dos subgrupos desde que o usuário responsável permita;
- **RFUS-1.10 -** Projetos públicos podem ser consultados por qualquer usuário;
- **RFUS-1.11 -** Um projeto pode ser manipulado pelos usuários que tem permissão;
- **RFUS-1.12 -** Um projeto tem um usuário responsável;
- **RFUS-1.13 -** Permitir visualização de modelos;
- **RFUS-1.14 -** Permitir edição de modelos.

3.2. REQUISITOS FUNCIONAIS DE ACESSO

- **RFAC-2.1 -** Suportar diferentes tipos de usuários:
    - **RFAC-2.1.1 - Usuário não registrado**: pode consultar o repositório de modelos públicos;
    - **RFAC-2.1.2 - Usuário individual:** com as mesmas atribuições do RFAC-2.1.1, porém com acesso ao serviço de armazenamento e transformação;
    - **RFAC-2.1.3 - Usuário membro de grupo**: com as mesmas atribuições do RFAC-2.1.1, porém com acesso ao serviço de armazenamento e transformação que o "usuário responsável por grupo" permitir;
    - **RFAC-2.1.4 - Usuário responsável por grupo**: com as mesmas atribuições do RFAC-2.1.2, porém pode definir permissão para usuários do grupo e gerenciar usuários do grupo e grupos;
    - **RFAC-2.1.5 - Usuário administrador:** com as atribuições dos usuários anteriores, porém pode gerenciar todos os usuários e gerenciar o uso do sistema.
- **RFAC-2.2 - Usuário membro de grupo** poderá escolher uma ou mais sequência de regras de transformações que o responsável pelo grupo permitiu;
- **RFAC-2.3 -** Gerenciar usuário consiste em poder criar/alterar/excluir/bloquear e desbloquear usuários;
- **RFAC-2.4 -** Gerenciar uso do sistema consiste em verificar a utilização dos serviços de transformação e disponibilizar novos serviços de transformação, cancelar serviços de transformação e acompanhar como os usuários usam o sistema;

- **RFAC-2.5 -** Mais de um usuário pode manipular o projeto, desde que tenha permissão;
- **RFAC-2.6 -** Um usuário pode convidar outro usuário para participar do projeto;
- **RFAC-2.7 -** O usuário responsável pelo grupo tem acesso irrestrito a todos os projetos do grupo;
- **RFAC-2.8 -** O usuário responsável por um projeto:
    - **RFAC-2.8.1 -** pode definir a permissão de leitura ou leitura/escrita para um usuário convidado;
    - **RFAC-2.8.2 -** pode remover a permissão de um usuário convidado;
    - **RFAC-2.8.3 -** pode transferir um projeto para outro usuário com concordância do usuário destino;
    - **RFAC-2.8.4 -** pode excluir o projeto;
    - **RFAC-2.8.5 -** pode gerenciar um projeto, como nome do projeto e serviços que o projeto vai usar;
    - **RFAC-2.8.6 -** vai poder escolher uma ou uma sequência de regras de transformações a ser aplicados nos modelos dos projetos.
- **RFAC-2.9 -** Número de membros em um grupo pode ser alterado pelo administrador.

3.3. REQUISITOS FUNCIONAIS DE INTERFACE

- **RFIT-3.1 –** Suporte para outros idiomas;
- **RFIT-3.2 –** Suporte para tema no modo claro e escuro;
- **RFIT-3.2 –** Aplicação responsiva em Desktop e Tablet;
- **RFIT-3.3 –** Painel de gestão com estatistas para uso do administrador.

4. REQUISITOS NÃO FUNCIONAIS

4.1. REQUISITOS NÃO FUNCIONAIS DE USO

- **RNFUS-1.1 -** Manter a privacidade entre os dados dos usuários;
- **RNFUS-1.2 -** Funções devem ser utilizáveis através do site e através de uma API Rest;
- **RNFUS-1.3 -** A disponibilização de novos serviços deve ser via interface de administração do sistema;
- **RNFUS-1.4 -** Limite de modelos no armazenamento de usuário pode ser imposto por um administrador;
- **RNFUS-1.5 -** O usuário responsável pelo grupo pode aplicar permissões a membros, grupo de membros ou a todos os membros;
- **RNFUS-1.6 -** O sistema tem uma cota padrão por usuário, gerenciado pelo administrador de 5MB;
- **RNFUS-1.7 -** Um grupo pode ter no máximo 250 membros;
- **RNFUS-1.8 -** Um grupo pode ter no máximo de 50 subgrupos.
- **RNFUS-1.9 -** Evitar perdas na transformação de modelos quando possível;
- **RNFUS-1.10 -** Suportar três dos principais formatos de modelos;

- **RNFUS-1.11 -** Compatibilidade com as últimas versões dos browsers "Google Chrome" e "Mozilla Firefox";
- **RNFUS-1.12 -** Deve ser usado TLS.

APÊNDICE B - DOCUMENTO DE ESPECIFICAÇÂO DA API

```yaml
openapi: 3.0.0
info:
  title: API
  description: Descrição.
  version: 0.1.0
paths:
  /auth:
    post:
      description: Get the authorization token
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                email:
                  type: string
                  description: User email
                password:
                  type: string
                  description: User password
      responses:
        '200':
          description: Successfully return a list of the models.
          content:
            application/json:
              schema:
                type: object
                properties:
                  auth_token:
                    type: string
                    description: JWT token
        '400':
          description: Invalid authentication.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
  /profile:
    get:
      description: Returns token user information
      responses:
        '200':
          description: Successfully return users information.
          content:
            application/json:
              schema:
                type: object
                properties:
                  email:
                    type: string
                    description: User email
                  first_name:
                    type: string
                    description: User first name
                  last_name:
                    type: string
                    description: User last name
                  locale:
                    type: string
                    description: User idiom locale
```

```
                theme:
                  type: string
                  description: Interface theme
      '401':
        description: Invalid credentials.
        content:
          application/json:
              schema:
                $ref: '#/components/schemas/error'
    security:
      - bearerAuth: []
/models:
  get:
    description: Returns all models
    parameters:
      - in: query
        name: public
        description: Filter for public or private models
        schema:
          type: string
          enum:
            - 'true'
            - 'false'
        required: false
      - in: query
        name: name
        description: Filter models using their names
        schema:
          type: string
        required: false
      - in: query
        name: limit
        description: Maximum amount of models returned
        schema:
          type: string
        required: false
      - in: query
        name: offset
        description: Offset for pagination
        schema:
          type: string
        required: false
      - in: query
        name: order_by_column
        description: Columns used in sorting
        schema:
          type: string
        required: false
      - in: query
        name: order_by_asc
        description: Sort using ascending order if true
        schema:
          type: string
          enum:
            - 'true'
            - 'false'
        required: false
    responses:
      '200':
        description: Successfully return a list of the models.
        content:
          application/json:
            schema:
              type: object
              properties:
                total_quantity:
                  type: integer
                  description: Total quanity of models
                filtered_quantity:
```

```
                  type: integer
                  description: Quantity of models after filtering
                data:
                  type: array
                  items:
                    $ref: '#/components/schemas/model'
      '401':
        description: Invalid credentials.
        content:
          application/json:
              schema:
                $ref: '#/components/schemas/error'
    security:
      - bearerAuth: []
  post:
    description: Create a new model
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/model_create'
    responses:
      '201':
        description: Successfully created the model.
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/model'
      '400':
        description: Invalid parameters.
        content:
          application/json:
              schema:
                $ref: '#/components/schemas/error'
      '401':
        description: Invalid credentials.
        content:
          application/json:
              schema:
                $ref: '#/components/schemas/error'
    security:
      - bearerAuth: []
/models/{model_id}:
  get:
    description: Returns a single model
    parameters:
      - in: path
        name: model_id
        description: Model id
        schema:
          type: string
        required: true
    responses:
      '200':
        description: Successfully return the model.
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/model'
      '401':
        description: Invalid credentials.
        content:
          application/json:
              schema:
                $ref: '#/components/schemas/error'
      '404':
        description: User not found.
        content:
```

```
          application/json:
              schema:
                $ref: '#/components/schemas/error'
  put:
    description: Update or create an model
    parameters:
      - in: path
        name: model_id
        description: Model id
        schema:
          type: string
        required: true
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/model_create'
    responses:
      '200':
        description: Successfully updated the model.
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/model'
      '201':
        description: Successfully created the model.
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/model'
      '401':
        description: Invalid credentials.
        content:
          application/json:
              schema:
                $ref: '#/components/schemas/error'
      '404':
        description: User not found.
        content:
          application/json:
              schema:
                $ref: '#/components/schemas/error'
  patch:
    description: Partially update an model
    parameters:
      - in: path
        name: model_id
        description: Model id
        schema:
          type: string
        required: true
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/model_create'
    responses:
      '204':
        description: Successfully updated the model.
      '401':
        description: Invalid credentials.
        content:
          application/json:
              schema:
                $ref: '#/components/schemas/error'
      '404':
        description: Model not found.
```

```
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
    delete:
      description: Delete a model
      parameters:
        - in: path
          name: model_id
          description: Model id
          schema:
            type: string
          required: true
      responses:
        '204':
          description: Successfully deleted the user.
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '404':
          description: Model not found.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
  /models/{model_id}/content:
    get:
      description: Get the content information of a model
      parameters:
        - in: path
          name: model_id
          description: Model id
          schema:
            type: integer
          required: true
      responses:
        '200':
          description: Successfully returned the model's content information.
          content:
            application/json:
              schema:
                type: object
                properties:
                  files:
                    type: array
                    items:
                      type: object
                      properties:
                        id:
                          type: string
                          description: Model content file id
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '404':
          description: Model not found.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
    post:
      description: Insert a new file to a model
      parameters:
```

```
        - in: path
          name: model_id
          description: Model id
          schema:
            type: integer
          required: true
      requestBody:
        required: true
        content:
          application/zip: {}
      responses:
        '204':
          description: Successfully created the model file.
        '400':
          description: Invalid content.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '404':
          description: Model not found.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
  /models/{model_id}/content/{file_id}:
    get:
      description: Get the content of file of a model
      parameters:
        - in: path
          name: model_id
          description: Model id
          schema:
            type: integer
          required: true
        - in: path
          name: file_id
          description: Model content file id
          schema:
            type: string
          required: true
      responses:
        '200':
          description: Successfully returned the model's content.
          content:
            application/xml: {}
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '404':
          description: Model or file not found.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
    put:
      description: Update the content of file of a model
      parameters:
        - in: path
          name: model_id
```

```
          description: Model id
          schema:
            type: integer
          required: true
        - in: path
          name: file_id
          description: Model content file id
          schema:
            type: string
          required: true
      requestBody:
        required: true
        content:
          application/xml: {}
      responses:
        '204':
          description: Successfully updated the model content.
        '400':
          description: Invalid content.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '404':
          description: Model not found.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
  /projects:
    get:
      description: Returns all projects
      parameters:
        - in: query
          name: public
          description: Filter for public or private projects
          schema:
            type: string
            enum:
              - 'true'
              - 'false'
          required: false
        - in: query
          name: name
          description: Filter projects using their names
          schema:
            type: string
          required: false
        - in: query
          name: limit
          description: Maximum amount of projects returned
          schema:
            type: string
          required: false
        - in: query
          name: offset
          description: Offset for pagination
          schema:
            type: string
          required: false
        - in: query
          name: order_by_column
          description: Columns used in sorting
```

```
          schema:
            type: string
          required: false
        - in: query
          name: order_by_asc
          description: Sort using ascending order if true
          schema:
            type: string
            enum:
              - 'true'
              - 'false'
          required: false
      responses:
        '200':
          description: Successfully return a list of the projects.
          content:
            application/json:
              schema:
                type: object
                properties:
                  total_quantity:
                    type: integer
                    description: Total quanity of models
                  filtered_quantity:
                    type: integer
                    description: Quantity of models after filtering
                  data:
                    type: array
                    items:
                      $ref: '#/components/schemas/project'
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
      security:
        - bearerAuth: []
    post:
      description: Create a new project
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/project_create'
      responses:
        '201':
          description: Successfully created the project.
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/project'
        '400':
          description: Invalid parameters.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
      security:
        - bearerAuth: []
  /projects/{project_id}:
    get:
```

```
    description: Returns a single project
    parameters:
      - in: path
        name: project_id
        description: Project id
        schema:
          type: string
        required: true
    responses:
      '200':
        description: Successfully return the project.
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/project'
      '401':
        description: Invalid credentials.
        content:
          application/json:
              schema:
                $ref: '#/components/schemas/error'
      '404':
        description: User not found.
        content:
          application/json:
              schema:
                $ref: '#/components/schemas/error'
    security:
      - bearerAuth: []
  put:
    description: Update or create an project
    parameters:
      - in: path
        name: project_id
        description: Project id
        schema:
          type: string
        required: true
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/project_create'
    responses:
      '200':
        description: Successfully updated the project.
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/project'
      '201':
        description: Successfully created the project.
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/project'
      '401':
        description: Invalid credentials.
        content:
          application/json:
              schema:
                $ref: '#/components/schemas/error'
      '404':
        description: User not found.
        content:
          application/json:
              schema:
                $ref: '#/components/schemas/error'
```

```yaml
    patch:
      description: Partially update an project
      parameters:
        - in: path
          name: project_id
          description: Project id
          schema:
            type: string
          required: true
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/project_create'
      responses:
        '204':
          description: Successfully updated the project.
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '404':
          description: Project not found.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
    delete:
      description: Delete a project
      parameters:
        - in: path
          name: project_id
          description: Project id
          schema:
            type: string
          required: true
      responses:
        '204':
          description: Successfully deleted the group.
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '404':
          description: Project not found.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
  /projects/{project_id}/content:
    get:
      description: Get the zipped content of a project
      parameters:
        - in: path
          name: project_id
          description: Project id
          schema:
            type: integer
          required: true
      responses:
        '200':
          description: Successfully returned the project's content.
          content:
            application/zip: {}
```

```
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '404':
          description: Project not found.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'

  /projects/{project_id}/models:
    get:
      description: Returns models from this project
      parameters:
        - in: path
          name: project_id
          description: Project id
          schema:
            type: string
          required: true
        - in: query
          name: public
          description: Filter for public or private models
          schema:
            type: string
            enum:
              - 'true'
              - 'false'
          required: false
        - in: query
          name: name
          description: Filter models using their names
          schema:
            type: string
          required: false
        - in: query
          name: limit
          description: Maximum amount of models returned
          schema:
            type: string
          required: false
        - in: query
          name: offset
          description: Offset for pagination
          schema:
            type: string
          required: false
        - in: query
          name: order_by_column
          description: Columns used in sorting
          schema:
            type: string
          required: false
        - in: query
          name: order_by_asc
          description: Sort using ascending order if true
          schema:
            type: string
            enum:
              - 'true'
              - 'false'
          required: false
      responses:
        '200':
          description: Successfully return a list of the models.
          content:
```

```
            application/json:
              schema:
                type: object
                properties:
                  total_quantity:
                    type: integer
                    description: Total quanity of models
                  filtered_quantity:
                    type: integer
                    description: Quantity of models after filtering
                  data:
                    type: array
                    items:
                      $ref: '#/components/schemas/model'
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
      security:
        - bearerAuth: []
  /projects/{project_id}/sequences:
    get:
      description: Returns sequences from this project
      parameters:
        - in: path
          name: project_id
          description: Project id
          schema:
            type: string
          required: true
      responses:
        '200':
          description: Successfully return a list of the sequences.
          content:
            application/json:
              schema:
                type: object
                properties:
                  total_quantity:
                    type: integer
                    description: Total quanity of models
                  filtered_quantity:
                    type: integer
                    description: Quantity of models after filtering
                  data:
                    type: array
                    items:
                      $ref: '#/components/schemas/sequence'
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
      security:
          - bearerAuth: []
  /sequences:
    get:
      description: Returns all sequences
      parameters:
        - in: query
          name: public
          description: Filter for public or private projects
          schema:
            type: string
            enum:
              - 'true'
```

```yaml
              - 'false'
          required: false
        - in: query
          name: limit
          description: Maximum amount of sequences returned
          schema:
            type: string
          required: false
        - in: query
          name: offset
          description: Offset for pagination
          schema:
            type: string
          required: false
        - in: query
          name: order_by_column
          description: Columns used in sorting
          schema:
            type: string
          required: false
        - in: query
          name: order_by_asc
          description: Sort using ascending order if true
          schema:
            type: string
            enum:
              - 'true'
              - 'false'
          required: false
      responses:
        '200':
          description: Successfully return a list of the sequences.
          content:
            application/json:
              schema:
                type: object
                properties:
                  total_quantity:
                    type: integer
                    description: Total quantity of sequences
                  filtered_quantity:
                    type: integer
                    description: Quantity of sequences after filtering
                  data:
                    type: array
                    items:
                      $ref: '#/components/schemas/sequence'
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
      security:
        - bearerAuth: []
    post:
      description: Create a new sequence
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/sequence_create'
      responses:
        '201':
          description: Successfully created the sequence.
          content:
            application/json:
              schema:
```

```
                $ref: '#/components/schemas/sequence'
        '400':
          description: Invalid parameters.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
      security:
        - bearerAuth: []
  /sequences/{sequence_id}:
    get:
      description: Returns a single sequence
      parameters:
        - in: path
          name: sequence_id
          description: Sequence id
          schema:
            type: string
          required: true
      responses:
        '200':
          description: Successfully return the sequence.
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/sequence'
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '404':
          description: User not found.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
    put:
      description: Update or create an sequence
      parameters:
        - in: path
          name: sequence_id
          description: Sequence id
          schema:
            type: string
          required: true
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/sequence_create'
      responses:
        '200':
          description: Successfully updated the sequence.
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/sequence'
        '201':
          description: Successfully created the sequence.
          content:
```

```
            application/json:
              schema:
                $ref: '#/components/schemas/sequence'
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '404':
          description: User not found.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
    patch:
      description: Partially update an sequence
      parameters:
        - in: path
          name: sequence_id
          description: Sequence id
          schema:
            type: string
          required: true
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/sequence_create'
      responses:
        '204':
          description: Successfully updated the sequence.
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '404':
          description: Sequence not found.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
    delete:
      description: Delete a sequence
      parameters:
        - in: path
          name: sequence_id
          description: Sequence id
          schema:
            type: string
          required: true
      responses:
        '204':
          description: Successfully deleted the user.
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '404':
          description: Sequence not found.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
```

```yaml
/sequences/{sequence_id}/transformations:
  get:
    description: Returns all transformations from the sequence
    parameters:
      - in: path
        name: sequence_id
        description: Sequence id
        schema:
          type: string
        required: true
    responses:
      '200':
        description: Successfully return a list of the transformations.
        content:
          application/json:
            schema:
              type: object
              properties:
                total_quantity:
                  type: integer
                  description: Total quantity of transformations
                filtered_quantity:
                  type: integer
                  description: Quantity of transformations after filtering
                data:
                  type: array
                  items:
                    type: object
                    properties:
                      id:
                        type: string
                        description: Transformation id
      '401':
        description: Invalid credentials.
        content:
          application/json:
              schema:
                $ref: '#/components/schemas/error'
    security:
      - bearerAuth: []
  put:
    description: Set the sequence transformations
    parameters:
      - in: path
        name: sequence_id
        description: Sequence id
        schema:
          type: string
        required: true
    requestBody:
      required: true
      content:
        application/json:
          schema:
            type: array
            items:
              type: object
```

```
                properties:
                  id:
                    type: string
                    description: Transformation id
      responses:
        '204':
          description: Successfully set the transformations sequence.
        '400':
          description: Invalid parameters.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
      security:
        - bearerAuth: []
  /transformations:
    get:
      description: Returns all transformations
      responses:
        '200':
          description: Successfully return a list of the transformations.
          content:
            application/json:
              schema:
                type: object
                properties:
                  total_quantity:
                    type: integer
                    description: Total quantity of transformations
                  filtered_quantity:
                    type: integer
                    description: Quantity of transformations after filtering
                  data:
                    type: array
                    items:
                      type: object
                      properties:
                        id:
                          type: string
                          description: Transformations id
                        from:
                          type: string
                          description: Original model kind
                        to:
                          type: string
                          description: Model kind of the new model to be created
        '401':
          description: Invalid credentials.
          content:
            application/json:
                schema:
                  $ref: '#/components/schemas/error'
      security:
        - bearerAuth: []
components:
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
  schemas:
    model:
      type: object
```

```
      properties:
        id:
          description: Model id
          type: string
        name:
          description: Model name
          type: string
        description:
          description: Model description
          type: string
        kind:
          description: Model content kind
          type: string
        empty:
          description: Returns if it's empty
          type: boolean
        tags:
          description: Model tags
          type: array
          items:
            type: object
            properties:
              name:
                type: string
        project:
          description: The models project id
          type: string
        owner:
          description: Owner of this model
          type: string
        creation:
          description: Creation date timestamp
          type: integer
    sequence:
      type: object
      properties:
        id:
          description: Sequence id
          type: string
        name:
          description: Sequence name
          type: string
        description:
          description: Sequence description
          type: string
        project:
          description: The sequences project id
          type: string
        owner:
          description: Owner of this sequence
          type: string
        creation:
          description: Creation date timestamp
          type: integer
    project:
      type: object
      properties:
        id:
          description: Project id
          type: string
        name:
          description: Project name
          type: string
        description:
          description: Project description
          type: string
        owner:
          description: Owner of this project
          type: string
```

```
      tags:
        description: Project tags
        type: array
        items:
          type: object
          properties:
            name:
              type: string
      creation:
        description: Creation date timestamp
        type: integer
  error:
    type: object
    properties:
      success:
        type: boolean
        description: Always false.
      error:
        type: string
        description: Error kind.
      error_message:
        type: string
        description: The error message.
  model_create:
    type: object
    properties:
      project:
        description: The models project id
      name:
        description: Model name
        type: string
      description:
        description: Model description
        type: string
      kind:
        description: Model content kind
        type: string
      tags:
        description: Model tags
        type: array
        items:
          type: object
          properties:
            name:
              type: string
    required:
      - name
      - kind
  sequence_create:
    type: object
    properties:
      project:
        description: The sequences project id
      name:
        description: Sequence name
        type: string
      description:
        description: Sequence description
        type: string
    required:
      - name

  project_create:
      type: object
      properties:
        name:
          description: Project name
          type: string
        description:
```

```
            description: Project description
            type: string
          tags:
            description: Project tags
            type: array
            items:
              type: object
              properties:
                name:
                  type: string
        required:
          - name
```