UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS I – CAMPINA GRANDE
CENTRO DE CIÊNCIA E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE GRADUAÇÃO EM COMPUTAÇÃO

PABLO MONTEIRO SANTOS

CRIANDO WEB SERVICE PARA AUTOMATIZAR TESTES DE USABILIDADE A PARTIR DO MÉTODO POR INSPEÇÃO A PADRÕES

CAMPINA GRANDE

2023

PABLO MONTEIRO SANTOS

CRIANDO WEB SERVICE PARA AUTOMATIZAR TESTES DE USABILIDADE A PARTIR DO MÉTODO POR INSPEÇÃO A PADRÕES

Trabalho de Conclusão de Curso de Graduação em Ciência da Computação da Universidade Estadual da Paraíba, como requisito à obtenção do título de Bacharel em Ciência da Computação.

Área de concentração: Usabilidade e Fatores Humanos

Orientador: Prof. Dr. Daniel Scherer

CAMPINA GRANDE

2023

S237c Santos, Pablo Monteiro.
Criando *web service* para automatizar testes de usabilidade a partir do método por inspeção a padrões [manuscrito] / Pablo Monteiro Santos. - 2023.
96 p. : il. colorido.

Digitado.
Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Universidade Estadual da Paraíba, Centro de Ciências e Tecnologia, 2023.
"Orientação : Prof. Dr. Daniel Scherer, Coordenação do Curso de Computação - CCT. "

1. Usabilidade. 2. Interface de Programação de Aplicação - API. 3. Experiência do usuário. I. Título

21. ed. CDD 005.3

Elaborada por Talita M. A. Tavares - CRB - CRB 15/971 BC/UEPB

PABLO MONTEIRO SANTOS

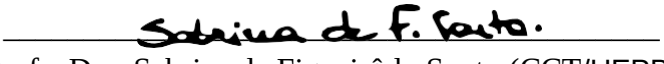
CRIANDO WEB SERVICE PARA AUTOMATIZAR TESTES DE USABILIDADE A PARTIR DO MÉTODO POR INSPEÇÃO A PADRÕES

Trabalho de Conclusão de Curso de Graduação em Ciência da Computação da Universidade Estadual da Paraíba, como requisito à obtenção do título de Bacharel em Ciência da Computação.

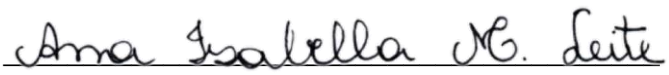
Aprovada em: 23/08/2023

BANCA EXAMINADORA

Prof. Dr. Daniel Scherer (CCT/UEPB)
Orientador(a)



Profa. Dra. Sabrina de Figueirêdo Souto (CCT/UEPB)
Examinador(a)



Profa. Me. Ana Isabella Muniz Leite (CCT/UEPB)
Examinador(a)

RESUMO

Os testes de usabilidade são uma etapa fundamental no desenvolvimento de qualquer tipo de produto. No entanto, a realização desses testes pode ser demorada e requerer muitos recursos. Com a crescente popularidade de serviços web e APIs, tornou-se necessário projetar e implementar esses sistemas de maneira que os mesmos facilitem a automação das avaliações de usabilidade. O método API *First* é uma metodologia que enfatiza o projeto da API antes da implementação do serviço propriamente dito. Este trabalho apresenta um *web service* REST criado através do método API *First* para automatizar o processo de testes de usabilidade de qualquer tipo de produto, usando o método de inspeção a padrões. O objetivo desse serviço é proporcionar uma solução eficiente e escalável para testar a usabilidade de qualquer produto e melhorar a experiência do usuário. Ao automatizar o processo de testes de usabilidade, este *web service* busca facilitar a identificação e correção de problemas de usabilidade, resultando em melhores produtos para os usuários.

Palavras-Chave: usabilidade; apis; api first; web service; rest.

ABSTRACT

Usability tests are a fundamental step in the development of any type of product. However, performing these tests can be time-consuming and resource-intensive. With the growing popularity of web services and APIs, it has become necessary to design and implement these systems in a way that facilitates the automation of usability assessments. The "API First" method is a methodology that emphasizes the design of the API before the implementation of the service itself. This work presents a web service that uses the API First method to automate the usability testing process of any type of product, using the pattern inspection method. The purpose of this service is to provide an efficient and scalable solution to test the usability of any product and improve the user experience. By automating the usability testing process, this web service seeks to facilitate the identification and correction of usability problems, resulting in better products for users.

Keywords: usability; apis; api first; web service; rest.

LISTA DE ILUSTRAÇÕES

Figura 1 – Ilustração do diagrama de atividades..... 25
Figura 2 – Ilustração da atividade Receber Produto..... 27
Figura 3 – Ilustração da atividade Emitir Relatório..... 28
Figura 4 – Ilustração da atividade Executar Avaliação..... 30
Figura 5 – Ilustração da atividade Executar Avaliação II..... 31
Figura 6 – Principais módulos do framework..... 32
Figura 7 – Exemplo de arquivo de configuração de acesso a base..... 34
Figura 8 – -Ilustração da Clean Architecture..... 37
Figura 9 – Ilustração da Simple Clean Architecture..... 38
Figura 10 – Estrutura de pacotes da aplicação..... 39
Figura 11 – Código mermaid responsável por gerar o diagrama de entidade e relacionamento..... 42
Figura 12 – Diagrama de entidade e relacionamento gerado via mermaid..... 44
Figura 13– Swagger UI do web service REST…….. 48
Figura 14 – Comando curl para Cadastro de cliente..... 49
Figura 15 – Tela do Postman com o Resultado da consulta de criação de um cliente válido..... 50
Figura 16 – Comando curl para Cadastro de cliente inválido..... 50
Figura 17 – Tela do Postman com o Resultado da consulta de cadastro de cliente inválido..... 51
Figura 18 – Comando curl para Cadastro de produto válido..... 51
Figura 19 – Tela do Postman com o Resultado da consulta de cadastro de produto válido..... 52
Figura 20 – Comando curl para Cadastro de produto inválido..... 52
Figura 21 – Tela do Postman com o Resultado da consulta de cadastro de produto inválido..... 53
Figura 22 – Comando curl para Cadastro de requisição de avaliação de um produto previamente cadastrado..... 53
Figura 23 – Tela do Postman com o Resultado da consulta de cadastro de requisição de avaliação..... 54
Figura 24– Tela do Postman com o Resultado da consulta de cadastro de 54

requisição de avaliação.....
Figura 25 – Tela do Postman com o Resultado da consulta de cadastro de requisição de avaliação para um produto inválido..... 55
Figura 26 – Comando curl para Cadastro de uma questão na base de dados da api 55
Figura 27 – Tela do Postman com o Resultado da consulta de cadastro de questão 56
Figura 28 – Comando curl para Cadastro de um questionário na base de dados da api..... 56
Figura 29 – Tela do Postman com o Resultado da consulta de cadastro de questionário..... 57
Figura 30 – Comando curl para Cadastro de um relatório na base de dados da api.. 58
Figura 31 – Tela do Postman com o Resultado da consulta de cadastro de relatório..... 59

LISTA DE QUADROS

Quadro 1 - Requisitos funcionais para cada perfil de acesso..... 24
Quadro 2 - Símbolos de cardinalidade do mermaid e seus significados..... 43

SUMÁRIO

1 INTRODUÇÃO 11
1.1 Objetivos 12
1.1.1 Objetivo Geral 12
1.1.2 Objetivos Específicos 12
2 REFERENCIAL TEÓRICO 13
2.1 Internet 13
2.2 WWW (World Wide Web) 14
2.3 HTTP (Hypertext Transfer Protocol) 14
2.4 Web Services 15
2.5 REST (Representational State Transfer) 16
2.6 XML-RPC x RESTful 17
2.7 REST x SOAP 18
2.8 Open API 18
2.9 Usabilidade 19
2.9.1 Inspeção a Padrões 20
3 METODOLOGIA 22
4 DESENVOLVIMENO 24
4.1 Requisitos Funcionais Definidos 24
4.1.1 Receber Produto 26
4.1.2 Emitir Relatório 27
4.1.3 Executar avaliação 29
4.1.4 Configurar SW 31
4.2 Tecnologias e Padrões Utilizadas no Desenvolvimento do Web Service 32
4.2.1 Spring Framework 32
4.2.2 Spring boot 33
4.2.3 Mapeamento do Banco de Dados com JPA 34
4.3 Clean architecture 35
4.3.1 Implementação 37
4.4 Modelagem dos schemas da api 39
4.5 Modelagem dos endpoints da api 40
4.6 Criação do documento open api 41
4.7 Modelagem da base de dados 41

4.7.1 Modelo de entidade e relacionamento 41
4.7.2 Mermaid 42
4.7.3 Mermaid live editor 42
4.7.4 Entidade product 45
4.7.5 Entidade evaluation-request 45
4.7.6 Entidade report 45
4.7.7 Entidade customer 45
4.7.8 Entidade question 46
4.8 Code generator 46
4.9 Documentação 47
5 RESULTADOS 49
5.1 Endpoint Customer 49
5.1.1 Cadastro de cliente válido 49
5.1.2 Cadastro de cliente inválido 50
5.2 ***Endpoint Product*** 51
5.2.1 Cadastro de produto válido 51
5.2.2 Cadastro de produto inválido 52
5.3 Endpoint Request Evaluations 53
5.3.1 Cadastro de Requisição de avaliação para um produto válido 53
5.3.2 Cadastro de Requisição de avaliação para um produto inválido 54
5.4 Endpoint Question 55
5.5 Endpoint Questionnaires 56
5.6 Endpoint Reports 58
6 CONCLUSÃO 60
6.1 Trabalhos futuros 60
REFERÊNCIAS 62
APÊNDICE A – DOCUMENTO OPEN API 63

1 INTRODUÇÃO

Cada vez mais as empresas consomem e disponibilizam APIs para terceiros e isso tem criado várias oportunidades de negócios nos mais diversos segmentos. O bom design de APIs se torna um dos principais fatores que possibilita grandes empresas como a Uber a criarem excelentes experiências para os seus usuários. A empresa de mobilidade urbana agrega à estrutura de seu aplicativo a API do Google Maps, os quais já tem uma ótima precisão e todas as informações de geolocalização necessárias para o seu ideal funcionamento. Munidos de exemplos como esses e de acordo com a RapidAPI (2020), a indústria de desenvolvimento de software vem adotando cada vez mais a utilização de APIs na construção e integração de seus sistemas.

Atualmente existem duas abordagens principais para a construção de APIs *Rest*, *Code First* e Api *First*. A primeira é uma abordagem mais tradicional, onde o desenvolvimento do código acontece logo após a definição dos requisitos, eventualmente gerando a documentação do código juntamente do contrato da API. Logo, primeiro as regras de negócio do sistema são codificadas em alguma linguagem de programação pela equipe de desenvolvimento e, a partir disso, são criados os *endpoints* da API que vão expor essas funcionalidades criadas.

Na segunda abordagem, o foco inicial é na construção da modelagem da api e não na implementação das regras de negócios do sistema, ou seja, o Api *First* defende o design do contrato da API antes de escrever qualquer código. Assim inicia-se o desenvolvimento pela construção de um contrato legível por humanos e máquinas, descrito em um formato aberto, como o próprio *Swagger* e o *OpenAPI*. Sendo assim, uma vez que a API é projetada, ela se torna uma especificação codificada que deve ser usada como base para a construção do *backend*, *frontend* e base de dados da aplicação.

Na abordagem *Code First* os desenvolvedores podem começar a implementar a API mais rapidamente, pois eles começam a codificar a API diretamente do documento de requisitos. Conforme Michael Ochs (2021), que destaca a agilidade e a rapidez como vantagens da abordagem *Code First*, em situações onde é dada uma importância muito grande para a velocidade e agilidade de entrada no mercado essa abordagem pode ser consistente com tais objetivos.

Já no Api *first*, para qualquer projeto de desenvolvimento, as APIs recebem tratamento especial, o que ajuda a evitar o negligenciamento de funções necessárias ou gastar muito tempo e energia em recursos indesejados. Semelhante ao design de produto centrado no

cliente, em que os designers renderizam conceitos e solicitam feedback do consumidor antes que as especificações técnicas sejam detalhadas.

Porém a necessidade de estabelecer um contrato envolve gastar mais tempo pensando no design de uma API. Também envolve planejamento e colaborações adicionais com os *stakeholders*, para que seja possível coletar feedbacks sobre o design da API antes que qualquer código seja escrito.

Além disso, segundo Eisenberg (2019) o método API *first* permite a reutilização de código e aumenta a escalabilidade do sistema. Como a API é desenvolvida de forma independente do aplicativo, ela pode ser facilmente utilizada por outros sistemas e aplicativos, o que aumenta a eficiência e reduz os custos de desenvolvimento.

Devido a essas vantagens, optou-se no desenvolvimento deste *web service* pela utilização da abordagem centrada no design da API.

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo deste trabalho de conclusão de curso é o desenvolvimento de um *web service* com as operações necessárias para a realização de avaliação de usabilidade a partir do método por inspeção a padrões, utilizando a arquitetura *REST* e a abordagem API *First*

1.1.2 Objetivos Específicos

Elaborar um serviço web simples e prático utilizando uma API *REST.* Verificar a praticidade da metodologia API *First* para o desenvolvimento de APIs *RESTs.* Projetar uma arquitetura de software escalável e de fácil manutenibilidade

O trabalho está estruturado da seguinte forma: A Seção 2 apresenta o referencial teórico com o intuito de possibilitar um entendimento sobre a problemática desse trabalho; na Seção 3 foi apresentando o delineamento metodológico seguido por esse TCC; a Seção 4 explica como foi feito o desenvolvimento desta pesquisa, nele é retratado as tecnologias utilizadas e o modelo de arquitetura seguido; e na Seção 5 são apresentados os resultados obtidos, seguidamente é apresentado na seção 6 às considerações finais e os trabalhos futuros.

2 REFERENCIAL TEÓRICO

2.1 Internet

A história da internet começa com a criação do ARPANET *(Advanced Research Projects Agency Network)* em 1969 pelo Departamento de Defesa dos Estados Unidos. Segundo Kurose e Ross (2017), "o objetivo era conectar as redes militares e universitárias para que os cientistas pudessem compartilhar recursos e trabalhar juntos em projetos de pesquisa".

Em 1972, a ARPANET adotou o protocolo TCP/IP *(Transmission Control Protocol/Internet Protocol)*, que permite a comunicação entre redes diferentes. Segundo Kurose e Ross (2017), "o TCP/IP tornou-se o padrão para as redes que se expandiram além da ARPANET e é a base para a comunicação na internet de hoje".

Em 1983, a ARPANET foi dividida em duas redes: a MILNET (Military Network) e a internet. A internet começou a ser utilizada comercialmente e rapidamente se expandiu para se tornar a principal forma de comunicação e acesso à informação. Segundo Kurose e Ross (2017), "atualmente, milhões de redes privadas e públicas se conectam à internet através de provedores de serviços de internet (ISP) e ela é a base para uma ampla variedade de serviços e aplicações online".

Hoje a internet é uma rede global de computadores que possibilita a comunicação entre dispositivos de diferentes locais. Mas somente a partir dos anos 80, a internet começou a ser utilizada comercialmente, expandindo para se tornar a principal forma de comunicação e acesso à informação.

A internet é baseada em protocolos de comunicação, que são conjuntos de regras e procedimentos que permitem a comunicação entre dispositivos. O protocolo mais importante é o TCP/IP *(Transmission Control Protocol/Internet Protocol),* que define como os pacotes de dados devem ser transmitidos e como os endereços IP devem ser utilizados. Outros protocolos importantes incluem o HTTP (*Hypertext Transfer Protocol*), que é utilizado para transmitir páginas web, e o FTP (*File Transfer Protocol*), que é utilizado para transferir arquivos.

A internet é composta por milhões de redes privadas e públicas, que se conectam através de provedores de serviços de internet (ISP, na sigla em inglês). Os provedores de serviços de internet fornecem acesso à internet aos usuários finais através de diferentes tecnologias, como linhas telefônicas, cabos de fibra óptica e redes sem fio. A internet também é interconectada com outras redes, como redes de comunicação móvel e redes de televisão por

cabo.

Além de permitir a comunicação entre dispositivos, a internet também é a base para a maioria dos serviços e aplicações online, como e-mails, mensagens instantâneas, redes sociais, comércio eletrônico, jogos online e muito mais.

2.2 WWW (World Wide Web)

A World Wide Web (WWW ou Web) é um sistema de documentos hipertexto interligados e acessíveis através da internet. Ela foi criada em 1989 pelo cientista britânico Tim Berners-Lee, com o objetivo de facilitar a troca de informações científicas entre pesquisadores.

A Web funciona através de um protocolo de comunicação chamado HTTP (*Hypertext Transfer Protocol*), que permite a transferência de documentos através da internet. Os documentos são escritos em um formato de marcação chamado HTML *(Hypertext Markup Language*), que permite a criação de links entre documentos.

A Web também utiliza outros protocolos e tecnologias, como o URL *(Uniform Resource Locator*), que permite a identificação de documentos na Web, e o DNS *(Domain Name System*), que permite a tradução de nomes de domínio em endereços IP. Segundo Tanenbaum (2007) "o DNS é essencial para que as pessoas possam digitar nomes de domínio fáceis de lembrar, como www.example.com, ao invés de endereços IP complexos como 192.0.2.1, para acessar sites na Web".

A Web também é responsável por popularizar a utilização de navegadores web, como o Chrome, Firefox, e Safari, que permitem a visualização e interação com documentos na Web. Segundo Kurose e Ross (2017), "os navegadores tornam a Web acessível para um público amplo e permitem a criação de aplicações ricas e interativas, como lojas virtuais, redes sociais e sistemas de comunicação".

Em resumo, a World Wide Web é um sistema fundamental para a comunicação e acesso à informação na atualidade, ele é baseado em protocolos de comunicação, linguagem de marcação e tecnologias que possibilitam a interação entre usuários e documentos na internet, tornando a internet mais acessível e fácil de usar.

2.3 HTTP (Hypertext Tranfser Protocol)

O HTTP (*Hypertext Transfer Protocol*) é um protocolo de comunicação que permite a

transferência de documentos através da internet. Ele foi criado em 1989 por Tim Berners-Lee e é utilizado para acessar documentos na World Wide Web (WWW).

O funcionamento do HTTP baseia-se em requisições e respostas. Quando um usuário acessa um documento na Web através de um navegador, ele envia uma requisição HTTP para o servidor que armazena o documento. O servidor, por sua vez, envia uma resposta HTTP com o documento solicitado. As requisições HTTP são geralmente enviadas através do método GET, que solicita ao servidor que forneça o conteúdo de um recurso específico, enquanto as respostas são geralmente enviadas com o código de status HTTP 200 OK, indicando que a requisição foi bem-sucedida.

O HTTP é um protocolo de camada de aplicação, que significa que ele opera na camada superior da pilha de protocolos de comunicação da internet. Ele é baseado em TCP (*Transmission Control Protocol*), que é responsável por garantir a confiabilidade da comunicação através da internet. Segundo Tanenbaum (2007), "o TCP garante que os pacotes de dados sejam transmitidos corretamente e na ordem correta, e que retransmissões sejam realizadas quando necessário."

O HTTP também é amplamente utilizado para a comunicação entre sistemas, como aplicações web e APIs (*Application Programming Interface*). A versão mais recente do HTTP é o HTTP/2, que introduziu melhorias na performance e novos recursos, como multiplexação de requisições e compressão de cabeçalhos.

2.4 Web Services

Um *web service* é uma aplicação que oferece serviços através da internet, usando um protocolo de comunicação padrão, como o HTTP *(Hypertext Transfer Protocol*). Ele permite que diferentes aplicações se comuniquem e troquem dados sem a necessidade de se preocupar com as diferenças de plataforma e linguagem de programação.

Web services são frequentemente implementados usando protocolos baseados em XML, como SOAP *(Simple Object Access Protocol*) ou REST (*Representational State Transfer*). O SOAP é um protocolo mais antigo e é usado para trocar mensagens estruturadas usando o XML, enquanto o REST é um protocolo mais recente e é baseado em princípios de arquitetura de software, que usa o HTTP para trocar informações não estruturadas.

Um *web service* pode ser exposto através de uma API (*Application Programming Interface*), que define como os clientes podem acessar e utilizar os recursos disponibilizados pelo *web service*. Essa API pode ser documentada através de um arquivo no formato de *OpenAPI*, por exemplo.

As vantagens de utilizar *web services* incluem:

1. Integração: permite a integração de diferentes sistemas e aplicações, independentemente de sua plataforma ou linguagem de programação.
2. Reutilização de recursos: os recursos disponibilizados por um *web service* podem ser utilizados por vários clientes, o que aumenta a eficiência e reduz a duplicação de esforços.
3. Flexibilidade: os recursos de um *web service* podem ser acessados remotamente através da internet, o que permite acesso a recursos e dados de qualquer lugar e a qualquer momento.
4. Escalabilidade: é possível escalar o número de requisições e usuários que acessam um *web service* sem afetar sua funcionalidade e performance.
5. Segurança: os *web services* podem ser protegidos por mecanismos de autenticação e autorização, garantindo a segurança dos dados e recursos disponibilizados.
6. Manutenibilidade: os recursos disponibilizados por um *web service* podem ser atualizados e melhorados sem afetar os clientes que os utilizam.
7. Interoperabilidade: os *web services* seguem padrões e protocolos abertos, o que garante a interoperabilidade entre diferentes sistemas e aplicações.

2.5 REST (Representational State Transfer)

REST (*Representational State Transfer*) é um modelo de arquitetura de software que segue princípios simples e flexíveis para criar sistemas escaláveis e distribuídos. Ele utiliza o protocolo HTTP para comunicação entre cliente e servidor e se baseia em recursos representados por URI (*Uniform Resource Identifier*).

O modelo REST foi formalizado por Roy Fielding em sua tese de doutorado de 2000, intitulada "*Architectural Styles and the Design of Network-based Software Architectures*". Segundo Fielding, um sistema é considerado RESTful quando ele segue os seguintes princípios:

"*REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.* (Fieldng,2000)". Pro Fielding, isso significa que um sistema RESTful deve permitir a identificação de recursos através de URI, a manipulação desses recursos através de representações (como JSON ou XML), o envio de

mensagens auto-descritivas e o uso de hipermídia como mecanismo para controlar o estado da aplicação.

Fielding também destaca a importância da independência entre cliente e servidor, ou seja, o cliente deve ser capaz de acessar recursos sem conhecer a implementação do servidor e o servidor deve ser capaz de lidar com vários clientes sem conhecer suas implementações.

"*REST is not a protocol, but rather the architecture of the Web, which is based on a set of principles that include client-server, statelessness, cacheability, and a uniform interface.* (Fieldng,2000)"

Existem seis restrições que foram pensadas para constituir o estilo arquitetural. As restrições (*constraints*) do REST são:

1. Arquitetura cliente-servidor: O cliente e o servidor são independentes entre si e podem ser substituídos e evoluídos independentemente. Isso significa que o cliente não precisa conhecer a implementação do servidor e vice-versa.
2. Sem estado (*Stateless*): as requisições são independentes entre si, ou seja, o servidor não mantém nenhum estado relacionado às requisições anteriores. Isso torna o sistema escalável e facilita a recuperação de falhas.
3. Cacheável: as respostas podem ser armazenadas em cache para diminuir a carga no servidor e melhorar a performance.
4. Interface uniforme: as operações disponíveis são as mesmas para todos os recursos e são identificadas por verbos HTTP padrões (GET, POST, PUT, DELETE, etc.).
5. Sistema de mensagens: as requisições e respostas são mensagens auto-descritivas, ou seja, contém toda a informação necessária para serem processadas.
6. Hipermídia como mecanismo de estado: os recursos são representados como hipermídia e incluem links para acessar outros recursos relacionados. Isso permite ao cliente navegar pelos recursos sem precisar conhecer a estrutura da aplicação.

Logo, o REST é uma abordagem de estilo arquitetural para criar sistemas distribuídos e escaláveis na web, baseado em princípios simples e flexíveis, e que utiliza o protocolo HTTP para comunicação. A tese de doutorado de Roy Fielding é considerada uma referência importante para entender e aplicar essa abordagem.

2.6 XML-RPC x RESTful

XML-RPC e REST são duas abordagens diferentes para criar web services.

XML-RPC é um protocolo baseado em XML para chamadas remotas de

procedimentos (RPC, do inglês Remote Procedure Call). Ele permite que um cliente envie uma requisição a um servidor para executar uma determinada ação, passando parâmetros e obtendo uma resposta. A comunicação é realizada através de uma estrutura de dados XML que define a ação e os parâmetros.

Enquanto o XML-RPC segue uma abordagem mais procedimental, o RESTful segue uma abordagem mais orientada a recursos. Além disso, o XML-RPC tem uma estrutura de dados mais rígida e complexa, enquanto o RESTful utiliza formatos de dados mais simples, como JSON ou XML

2.7 REST x SOAP

O protocolo SOAP (*Simple Object Access Protocol*) foi desenvolvido pela Microsoft, IBM e DevelopMentor em meados dos anos 90 com o objetivo de prover uma forma de comunicação entre aplicações distribuídas independente de plataforma. Ele foi baseado no uso de documentos XML para definir a estrutura das mensagens e ainda forneceu mecanismos para segurança, autenticação e outros aspectos de mensagem. A primeira versão do SOAP foi lançada em 1999 e logo ganhou popularidade como forma de implementar Web Services.

Em 2000, o SOAP foi submetido ao W3C *(World Wide Web Consortium*) como proposta para padronização, e em meados de 2003, a primeira versão do SOAP 1.2 foi publicada como recomendação. O SOAP foi amplamente utilizado na criação de Web Services, em especial em aplicações empresariais, devido a sua capacidade de lidar com a segurança e garantia de entrega das mensagens, além de suportar vários protocolos de transporte, como HTTP, SMTP e JMS.

A principal diferença entre as arquiteturas SOAP e REST é que a primeira é orientada a procedimentos (ações) e possui uma estrutura de dados mais rígida, enquanto a segunda é orientada a recursos, os quais as ações (verbos HTTP) manipulam de modo direto os recursos.

2.8 Open API

O OpenAPI é um framework de especificação de API que foi originalmente desenvolvido pela Reverb Technologies e lançado em 2010 como Swagger. Em 2015, a Reverb Technologies foi adquirida pela SmartBear e o projeto Swagger foi renomeado para OpenAPI. Desde então, o OpenAPI tem sido amplamente utilizado para documentar APIs e fornecer aos desenvolvedores uma maneira fácil e padronizada de descrever e compartilhar

informações sobre APIs. A versão mais recente da especificação OpenAPI é a 3.0, que foi lançada em 2017 e oferece uma série de novos recursos e melhorias em comparação com as versões anteriores. Com a sua popularidade crescente, o OpenAPI se tornou um padrão de facto para a documentação de APIs em muitas empresas e organizações.

O OpenAPI permite que os desenvolvedores descrevam suas APIs, incluindo informações sobre recursos, métodos de requisição, parâmetros, respostas e possíveis códigos de erro. Além disso, a especificação também inclui recursos para testar suas APIs e garantir que estejam funcionando como o esperado. O OpenAPI é amplamente adotado devido a sua capacidade de fornecer uma descrição clara e precisa da API, além de ser compatível com uma ampla variedade de linguagens de programação. Além disso, a especificação é mantida por uma comunidade ativa e é apoiada por uma série de ferramentas, incluindo geradores de código, documentação e ferramentas de teste. Outra vantagem do OpenAPI é sua flexibilidade. Ele pode ser usado em projetos de qualquer tamanho, desde pequenos projetos de *start-up* até grandes aplicativos corporativos. Além disso, ele permite que os desenvolvedores trabalhem com as APIs em suas próprias linguagens de programação, facilitando a integração e a colaboração entre equipes.

O OpenAPI hoje é uma ferramenta essencial para qualquer equipe de desenvolvimento de API. Principalmente por permitir que as APIs sejam documentadas de forma clara e padronizada, geradas de forma automática e personalizadas para atender às necessidades específicas de cada projeto.

2.9 Usabilidade

A usabilidade é uma medida da eficiência, efetividade e satisfação com que os usuários podem interagir com um sistema, produto ou serviço. É uma preocupação fundamental para a indústria de tecnologia, pois a facilidade de uso é um fator decisivo na escolha dos usuários. Segundo Jakob Nielsen, um dos principais especialistas em usabilidade, "A usabilidade é um atributo importante do produto, pois afeta diretamente a satisfação do usuário, a eficiência de trabalho e a segurança dos dados" (Nielsen, 1993).

Para garantir a usabilidade de um sistema, é necessário realizar testes de usabilidade para identificar problemas e oportunidades de melhoria. No entanto, realizar testes manuais pode ser demorado e dispendioso, especialmente em sistemas complexos e em constante evolução. A automação de testes de usabilidade pode ajudar a superar esses desafios, pois permite que os testes sejam realizados de forma mais rápida, precisa e consistente.

De acordo com um estudo de Lim et al. (2019), a automação de testes de usabilidade pode aumentar a eficiência e a eficácia dos testes, permitindo uma melhor cobertura dos testes e a identificação de problemas de usabilidade mais rapidamente. Além disso, a automação de testes permite que os testes sejam realizados de forma consistente, o que é importante para garantir a qualidade do sistema ao longo do tempo.

Realizar testes de usabilidade é essencial para identificar problemas e melhorar a usabilidade, mas realizar testes manuais pode ser demorado e dispendioso. A automação de testes de usabilidade pode ajudar a superar esses desafios, permitindo testes mais rápidos, precisos e consistentes.

2.9.1 Inspeção a padrões

Dentro do domínio da usabilidade, a inspeção a padrões destaca-se como uma técnica especializada voltada para avaliar a usabilidade dos produtos. Sua premissa fundamental repousa na análise comparativa entre uma interface específica e padrões de design, normalmente advindos de normas tal como a ISO 9241, ou práticas amplamente reconhecidas na literatura de usabilidade

O primeiro passo crucial dessa técnica envolve a definição rigorosa dos padrões que nortearão a avaliação. O objetivo é garantir que as interfaces sejam desenvolvidas de forma a maximizar a satisfação do usuário, minimizando erros e otimizando a eficiência.

Outro passo importante é a definição dos avaliadores, pois estes desempenham um papel central neste processo. Profissionais com conhecimento em usabilidade são fundamentais para garantir que a análise seja tanto precisa quanto útil. Através de suas experiencias, desvios dos padrões estabelecidos são identificados, documentados e posteriormente discutidos em um fórum colaborativo. O produto final deste exercício é um relatório detalhado que não só realça as incongruências, mas também propõe soluções alinhadas às práticas recomendadas na usabilidade.

Para ilustrar de forma prática como ocorre uma inspeção baseada em padrões, é útil imaginar um cenário específico de aplicação. Considere-se o contexto de avaliação da interface de um aplicativo de e-commerce recém-lançado. Munido de um checklist de padrões reconhecidos no design de interfaces para comércio eletrônico, tais como "facilidade de busca de produtos", "clareza nas informações do produto" e "processo de checkout simplificado" (Nielsen, 1994; Preece, Rogers & Sharp, 2015), a avaliação se inicia pela página inicial. A presença e eficácia da barra de busca são inspecionadas por meio de testes

com a inserção de termos genéricos e específicos. As páginas de produtos são examinadas para verificar a qualidade das imagens, a clareza das descrições e a visibilidade dos preços. Durante o processo de checkout, observa-se o número de etapas requeridas, a clareza das instruções e o feedback proporcionado ao usuário após cada interação. Para cada padrão estabelecido no checklist, é documentado se a interface adere, desvia parcialmente ou não atende ao padrão proposto. Ao término desta avaliação, é possível ter uma compreensão detalhada dos pontos de conformidade e das áreas que necessitam de aprimoramento.

Embora a inspeção a padrões ofereça uma série de benefícios, como interfaces mais alinhadas às expectativas dos usuários, a técnica manual apresenta desafios, tais como:

- definição do checklist a ser aplicado;
- consolidação dos resultados obtidos, através da comparação de respostas entre os especialistas para definição de consenso;
- dificuldades em reuso de checklist para produtos equivalentes; bem como
- dificuldades para análise comparativa entre produtos equivalentes.

Desta forma, um ferramental que permita automatizar os processos de reuso de checklist, consolidação preliminar dos resultados, geração preliminar de relatório e possibilidade de comparativo entre resultados; pode ser um acréscimo valioso para a área de usabilidade.

3 METODOLOGIA

A presente pesquisa trata-se de um estudo exploratório, sendo assim foi necessário uma maior proximidade com o tema investigado, essa aproximação aconteceu através de uma análise bibliográfica e leitura de artigos científicos, bem como foi importante estudar diversas tecnologias, *frameworks* e conceitos de boas práticas de modelagem de APIs para que fosse possível conseguir atingir o objetivo desejado.

A metodologia utilizada para o desenvolvimento do software foi baseada na criação de *endpoints* REST seguindo a metodologia API FIRST para gerenciar as avaliações de usabilidade baseadas em padrões. O OpenAPI foi utilizado para modelar a API e especificar as operações disponíveis, como criar, ler, atualizar e excluir avaliações de usabilidade. O desenvolvimento deste trabalho precisou ser dividido nas seguintes etapas:

Definição dos requisitos: Nesta etapa procurou-se compreender e definir os requisitos como também o contexto de uso e as tarefas. A definição de requisitos é o processo de identificar as necessidades e expectativas do sistema, produto ou processo que será desenvolvido. O contexto de uso descreve as circunstâncias em que o sistema será utilizado, incluindo informações sobre o local, ambiente e objetivos. As tarefas são as atividades que o usuário deseja realizar com o sistema. Ambos são elementos cruciais para garantir que um sistema seja projetado de maneira adequada e atenda às expectativas dos stakeholders.

Definição da arquitetura e tecnologias: Nesta etapa foram escolhidas as soluções técnicas que serão utilizadas para desenvolver o sistema. Definiu-se a arquitetura do sistema, incluindo sua estrutura, componentes e interações. Como também as tecnologias e plataformas que serão utilizadas para implementar a arquitetura. A escolha da arquitetura e das tecnologias foi feita com cuidado para garantir que elas atendam às necessidades do sistema e sejam adequadas ao contexto de uso e aos requisitos definidos anteriormente.

Modelagem dos schemas da API: Foi definido os formatos e regras que serão utilizados para a comunicação entre o sistema e seus usuários. Os *schemas* da API descrevem as estruturas de dados que serão enviadas e recebidas pela API, incluindo os tipos de dados, os campos obrigatórios e opcionais, e as regras de validação.

Modelagem dos endpoints da API: É nesta fase que foram definidas as URLs e métodos HTTP que serão utilizados para acessar os recursos do sistema através da API. Os *endpoints* da API descrevem as ações que podem ser realizadas no sistema, como listar, criar, atualizar e excluir dados.

Criação do documento OPEN API: A partir da definição dos *schemas* e dos *endpoints* foi

possível construir o documento OpenAPI, que é uma representação em formato de arquivo dos *schemas* da API, incluindo a descrição dos *endpoints*, parâmetros de entrada, códigos de resposta e exemplos de requisições e respostas. A criação do documento OpenAPI é importante porque permite que desenvolvedores e usuários da API tenham uma visão geral da estrutura e funcionalidades da API. Além disso, o documento OpenAPI pode ser usado para gerar código automaticamente, documentação e outros recursos para ajudar na integração da API com outras aplicações.

Modelagem da base de dados: Nesta fase foram definidos os esquemas de banco de dados que serão utilizados para armazenar e recuperar os dados da aplicação. Garantindo assim que as informações fossem armazenadas de forma clara, precisa e estruturada. Além disso, a modelagem da base de dados permite que as relações entre as entidades do sistema sejam representadas de forma clara, facilitando a implementação das regras de negócio.

Criação do *Backend*: Neste momento foi implementado o código fonte da API usando as regras de negócios, tecnologias e arquitetura definidas anteriormente.

4 DESENVOLVIMENTO

Para a implementação de uma ferramenta, é necessário fazer-se um levantamento minucioso de requisitos, sendo assim, nessa etapa o estudo se concentrou em entender o contexto de uso do sistema, isso inclui o cenário onde a aplicação será utilizada, quais expectativas os usuários esperam encontrar e qual experiência é oferecido a cada interação com o sistema, logo procurou-se entender e definir os principais requisitos funcionais que deverão ser disponibilizados aos 3 perfis de usuários que terão acesso a aplicação:

Quadro 1 - Requisitos funcionais para cada perfil de acesso

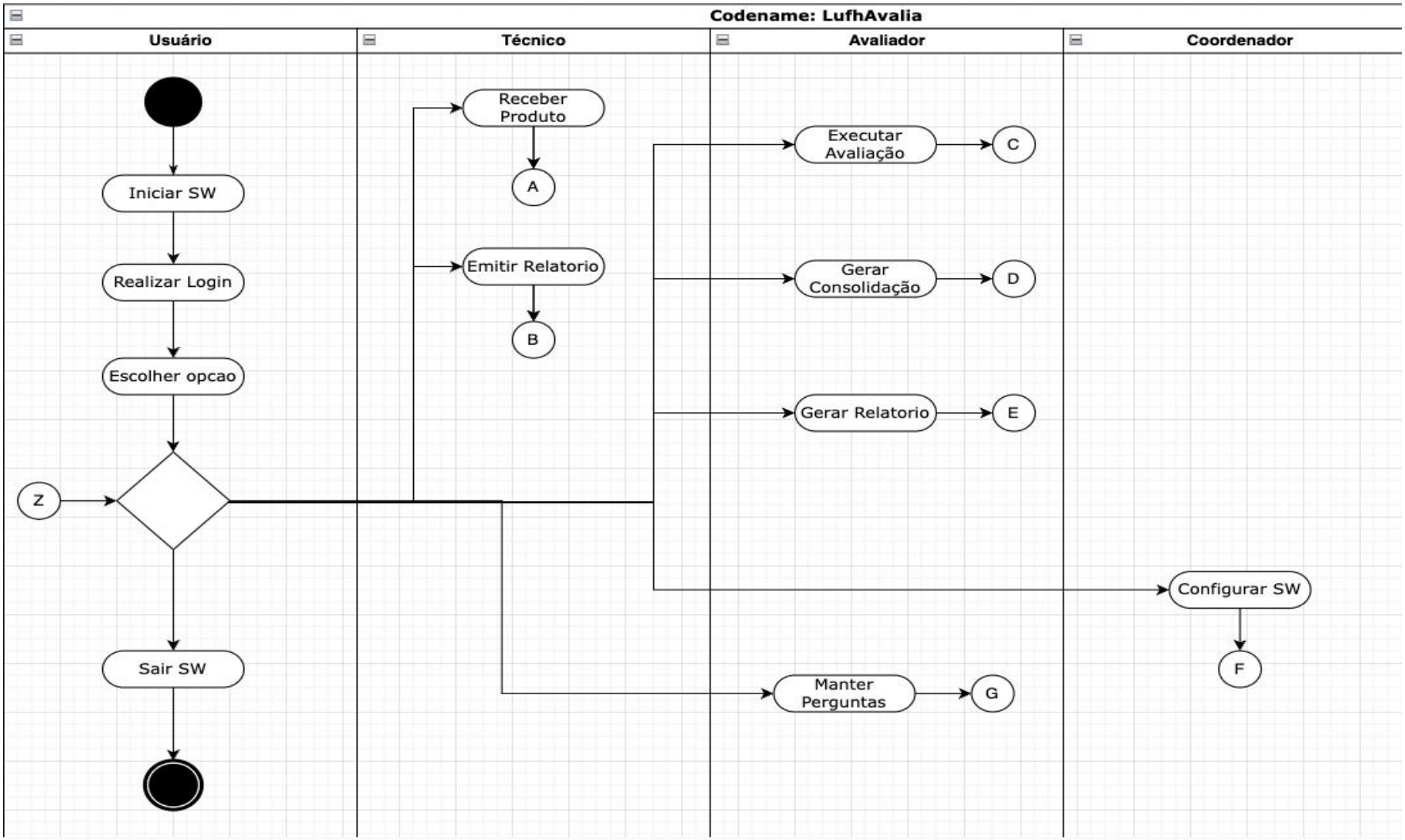
Perfil de acesso	**Requisitos**
Técnico	Receber Produto e Emitir relatório
Avaliador	Executar avaliação, Gerar consolidação, Gerar relatório e Manter perguntas
Coordenador	Configurar Sistema

Fonte: Elaborada pelo autor, 2023.

4.1 Requisitos Funcionais Definidos

Os requisitos funcionais são especificações que descrevem as funcionalidades que uma determinada solução de software deve oferecer. Eles são a base para a concepção e implementação de qualquer sistema, pois determinam o que o software precisa fazer para atender às necessidades dos usuários. Na construção da API, os requisitos funcionais foram utilizados para definir as funcionalidades que a API deveria oferecer. Eles foram uma referência para o projeto da arquitetura da API, guiando a escolha das tecnologias e ferramentas a serem utilizadas. O seguinte diagrama de atividades, representado na figura 1, foi construído para demonstrar os principais requisitos que cada um dos 3 perfis de usuário do sistema poderá interagir:

Figura 1 - Ilustração do diagrama de atividades



Fonte: Elaborada pelo autor, 2023.

Um diagrama de atividades é uma representação visual que descreve a sequência de atividades realizadas por um sistema ou processo. Ele é composto por elementos como atividades, decisões, mergulhos e ramificações, que representam, respectivamente, as ações realizadas, escolhas a serem feitas, pontos de entrada e saída de um processo e alternativas a serem tomadas.

Foi utilizado este tipo de diagrama para representar os requisitos funcionais do sistema, pois ele permite que sejam visualizadas as ações que o sistema deve realizar, bem como as escolhas que precisam ser feitas em cada etapa. Isso facilita a compreensão desses requisitos por parte de todos os envolvidos no projeto, além de permitir a identificação de possíveis problemas ou otimizações desde o início do desenvolvimento.

Ao construir o diagrama de atividades, cada requisito funcional foi representado por uma atividade no diagrama. As atividades foram ligadas entre si para representar a sequência de operações necessárias para realizar cada requisito funcional. Isso permitiu que os requisitos funcionais fossem visualizados de forma clara, facilitando a compreensão de como a API deverá funcionar. As "raias de natação", representadas pelas linhas horizontais, foram usadas para separar as atividades em diferentes áreas ou camadas de responsabilidade. Em relação à descrição de requisitos funcionais de cada ator no sistema, elas ajudam a identificar claramente quais atividades estão associadas a cada ator, ou seja, quem é responsável por realizar as atividades necessárias para atender aos requisitos funcionais. Ajudando também a destacar as interações entre os atores e as dependências entre as atividades.

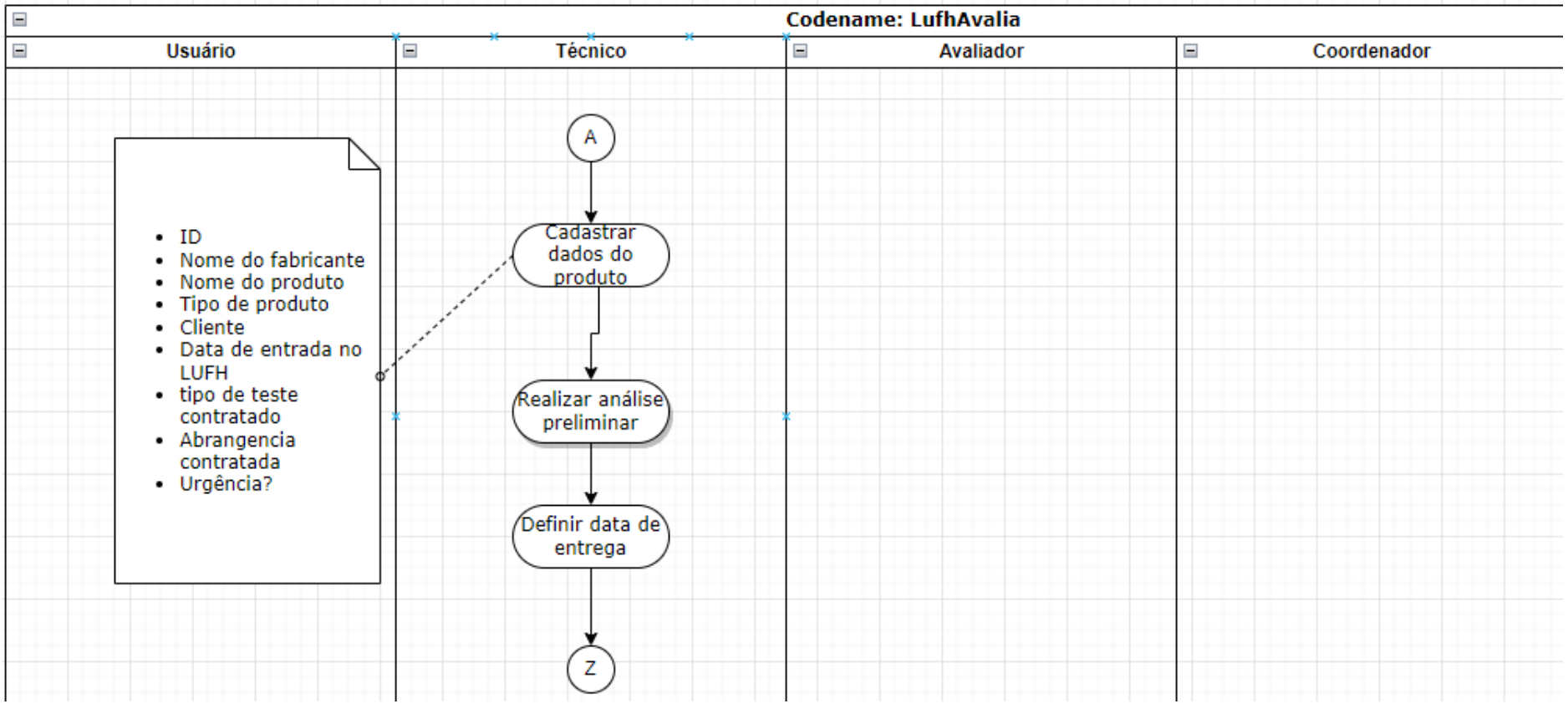
Algumas atividades do diagrama são compostas por subatividade que são elementos menores do diagrama de atividades que representam ações mais específicas que compõem uma atividade maior. Elas são usadas para descrever os requisitos funcionais de uma maneira mais detalhada, ajudando a clarificar a funcionalidade desejada. As próximas subseções apresentam as atividades que possuem ações mais específicas representadas por suas subatividades.

4.1.1 Receber Produto

A atividade Receber produto foi decomposta nas seguintes subatividades:

- Cadastrar dados do produto
- Realizar análise preliminar
- Definir data de entrega

Figura 2 - Ilustração da atividade receber produto



Fonte: Elaborada pelo autor, 2023.

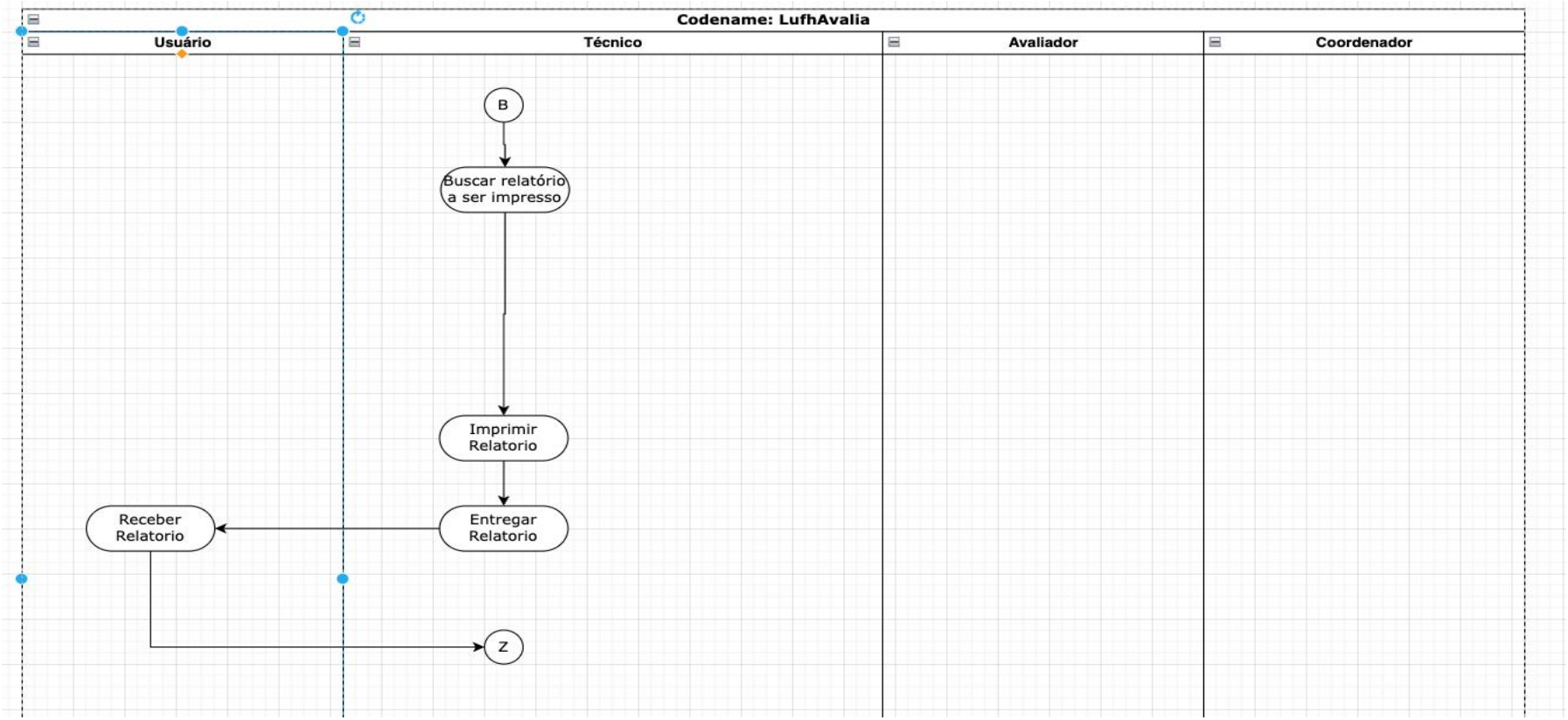
4.1.2 Emitir Relatório

A atividade Emitir Relatório foi decomposta nas seguintes subatividades:

- Buscar relatório a ser impresso

- Imprimir relatório
- Entregar relatório
- Receber relatório

Figura 3 - Ilustração da atividade emitir relatório



Fonte: Elaborada pelo autor, 2023.

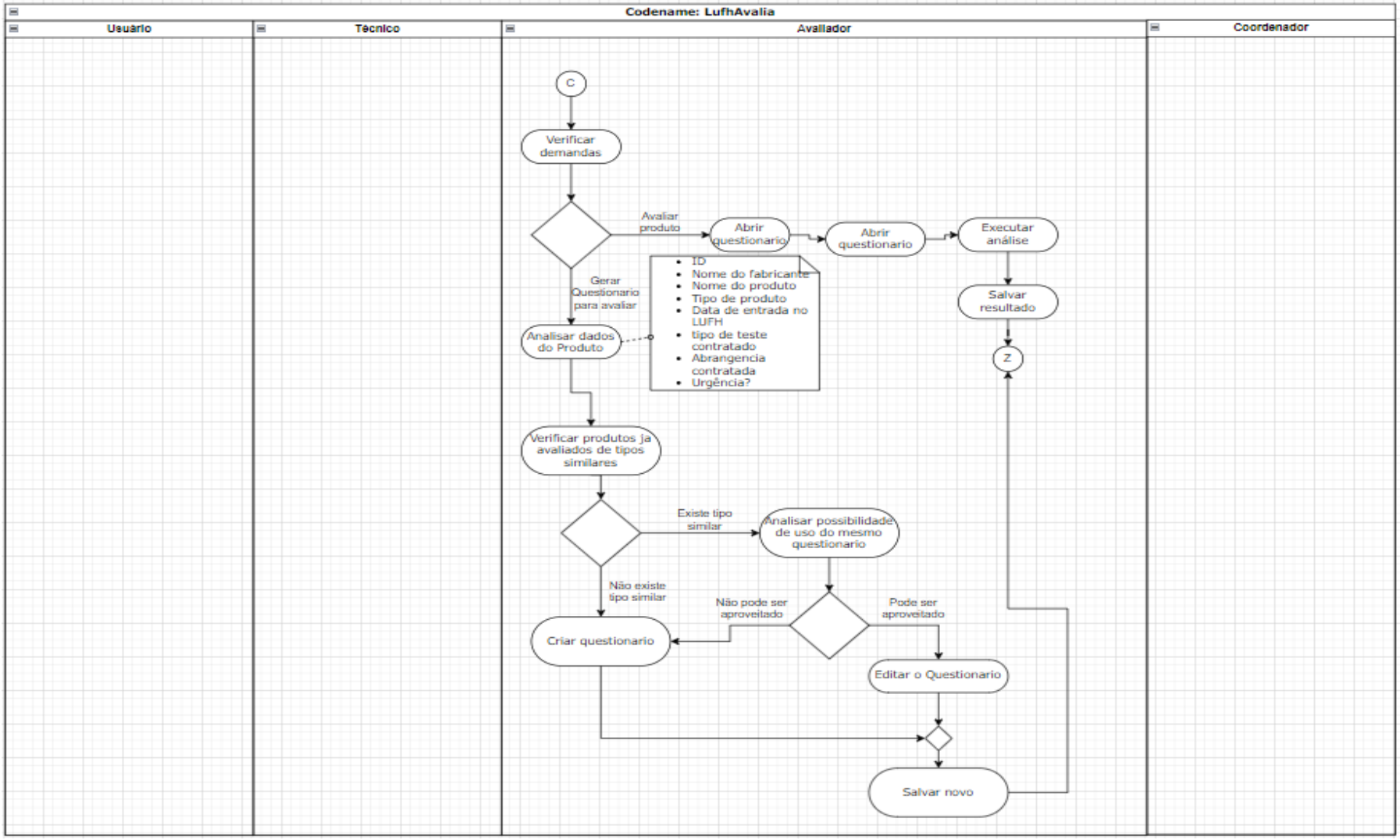
4.1.3 Executar avaliação

A atividade Executar Avaliação foi decomposta nas seguintes subatividades:

- Verificar demandas
- Abrir questionário
- Executar análise
- Salvar rascunho
- Analisar dados do produto
- Verificar produto já avaliados de tipos similares
- Criar questionário
- Analisar possibilidade de uso de um questionário já criado
- Editar questionário

  Salvar novo questionário

Figura 4 - Ilustração da atividade executar avaliação



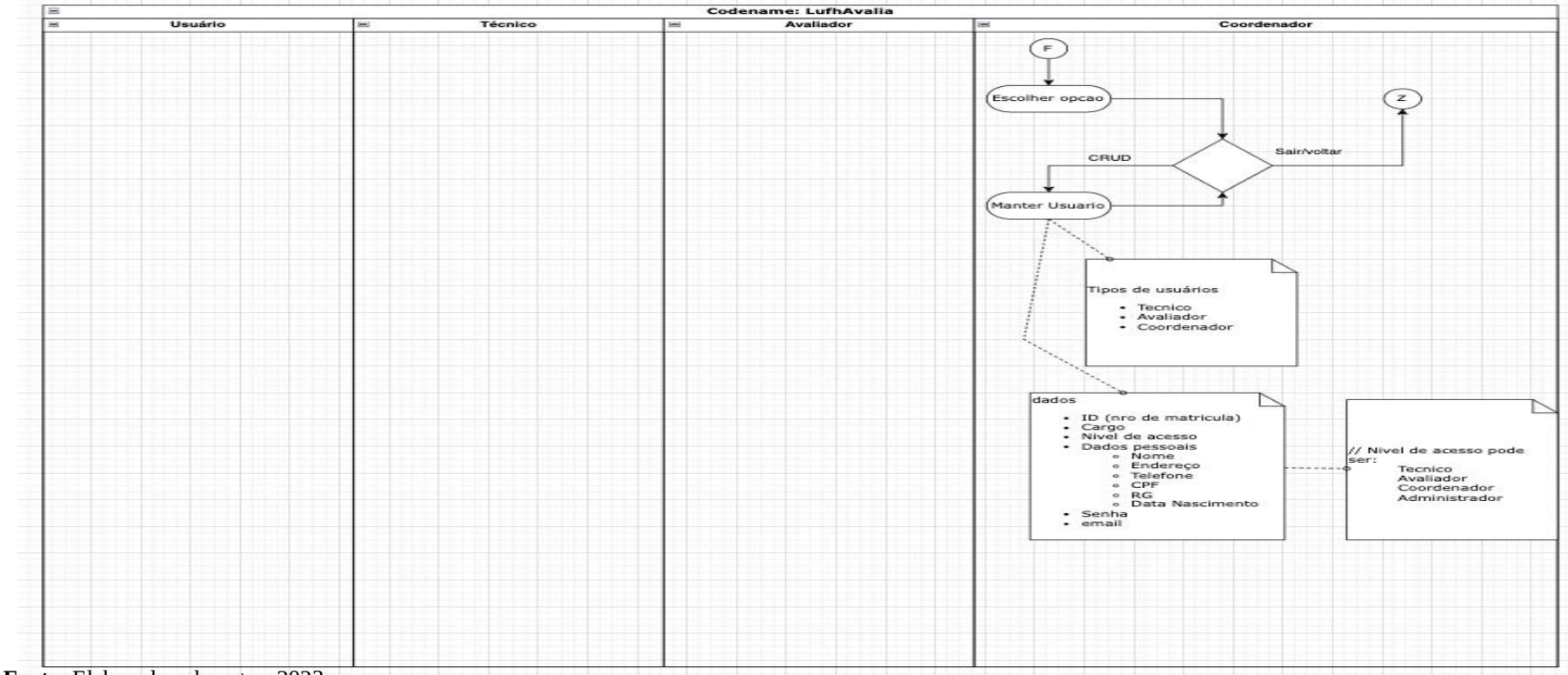
Fonte: Elaborada pelo autor, 2023.

4.1.4 Configurar SW

A atividade Configurar SW foi decomposta nas seguintes subatividades:

- Escolher opção
- Manter usuário

Figura 5 - Ilustração da atividade configurar sw



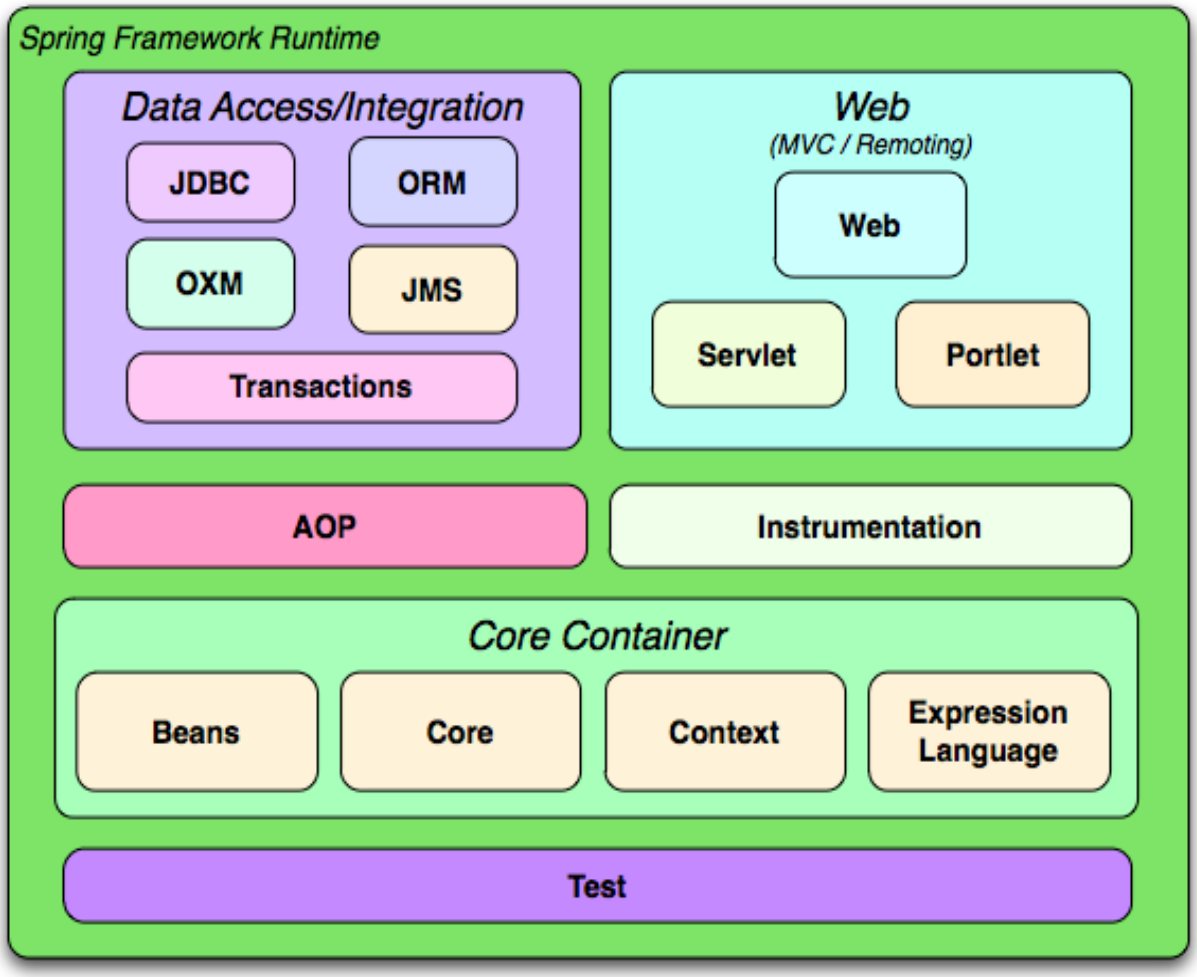
Fonte: Elaborada pelo autor, 2023.

4.2 Tecnologias e Padrões Utilizadas no Desenvolvimento do Web Service

Como o escopo deste trabalho limitou-se a modelagem e construção de um *backend* e espera-se que futuramente possa vir a existir uma ou mais aplicações clientes que deverão consumir as operações disponibilizadas por ele, optou-se pela criação de um **web service** utilizando o modelo arquitetural REST, que possui como uma de suas principais características a fácil interoperabilidade entre suas APIs e os diferentes sistemas que as utilizam. Visando também obter uma maior independência entre o *backend* e as suas futuras aplicações clientes, decidiu-se iniciar o desenvolvimento pela modelagem da API, seguindo a abordagem API *First*, sendo assim qualquer aplicação cliente deverá apenas usar o contrato estabelecido pela API ficando assim independente do *backend* desenvolvido.

4.2.1 Spring Framework

Figura 6 - Principais módulos do framework



Fonte: https://docs.spring.io/spring-framework/docs/3.0.0.M4/reference/html/ch01s02.html, 2022.

O Spring é um *framework open-sourc*e baseado em Java para a construção de aplicações. Ele fornece um conjunto abrangente de ferramentas para criar aplicações robustas e escaláveis, tornando-o um dos frameworks mais populares para desenvolvedores Java. Neste projeto, o framework Spring foi escolhido por vários motivos. Em primeiro lugar, o Spring fornece uma maneira simples e direta de construir *web services* RESTful. Ele abstrai muitos dos detalhes de baixo nível envolvidos na configuração de uma API RESTful, permitindo que os desenvolvedores se concentrem em escrever a lógica de negócios. Isso é particularmente útil neste projeto em que o objetivo é implementar rapidamente uma API funcional para automatizar o teste de usabilidade. Além disso, a estrutura Spring fornece recursos poderosos de acesso a dados, incluindo suporte para uma ampla variedade de bancos de dados e a capacidade de integração com vários frameworks ORM. Isso foi essencial para este projeto, pois a API exige a persistência dos dados sobre produtos, tipos de produtos, perguntas, grupos de perguntas e relatórios. Além disso, o framework Spring integra-se perfeitamente com o *code generator* do *OpenAPI*, tornando assim possível gerar código integrado com *spring* para o *back-end* da API automaticamente com base na descrição do documento *OpenAPI*. Esta foi uma vantagem significativa para este projeto, pois permitiu gerar rapidamente uma base de código que adere à especificação da API. Isso não apenas economizou tempo, mas também reduziu as chances de introduzir erros ou inconsistências no *codebase*.

A figura 2 apresenta alguns dos módulos mais importantes do Spring, entre eles podemos citar o Spring Core, que fornece suporte para o inversão de controle e injeção de dependência, o Spring Web, que fornece suporte para desenvolvimento de aplicativos web, e o Spring Data, que fornece suporte para acesso a dados. Além disso, o Spring Framework também oferece módulos para segurança, validação de dados, envio de e-mails, agendamento de tarefas, testes e etc.

O framework Spring foi uma excelente escolha para este projeto devido à sua simplicidade, poderosos recursos de acesso a dados e integração com o *code generator* do *OpenAPI*. Esses recursos permitiram o desenvolvimento de forma rápida e eficiente do *back-end* da API, garantindo que fosse robusto, escalável e aderisse à especificação da API definida no documento *OpenAPI*.

4.2.2 Spring boot

O Spring Boot é um dos módulos Spring mais utilizados do Framework, ele é projetado para ser mais fácil de usar e configurar. Isso o torna uma escolha popular para desenvolvedores que procuram simplificar a criação de aplicações Java

O Spring Boot elimina a necessidade das várias configurações manuais que são

necessárias para a utilização do Spring. Um exemplo de configuração manual do Spring que já é feita automaticamente pelo Spring boot é a configuração de um *DataSource*. No *Spring*, é necessário criar uma classe de configuração, configurar o driver e a URL, usuário e senha manualmente, através da criação de vários arquivos. No Spring Boot, essa configuração é feita automaticamente com base nas propriedades fornecidas por um único arquivo de configuração. Assim o Spring Boot permite que os desenvolvedores se concentrem em escrever as regras de negócios das aplicações ao invés de focar por exemplo nas criações das configurações de acesso a base de dados.

Figura 7 - Exemplo de arquivo de configuração de acesso a base

```
# Ajuste as configurações do servidor de aplicação
server.port=8080
server.servlet.context-path=/api

# Configurações da base de dados
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=user
spring.datasource.password=password
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

# Habilita o modo debug da aplicação
debug=true
```

Fonte: Elaborada pelo autor, 2023.

O arquivo de configuração de propriedades do Spring Boot é usado para especificar configurações para a aplicação. Ele pode conter informações sobre conexões de banco de dados, configurações de segurança, configurações de servidor, entre outras.

Algumas das propriedades mais comuns incluem o nome da aplicação, o endereço do servidor, o nome do usuário e a senha do banco de dados, e as configurações de segurança. Estas propriedades podem ser acessadas por meio de código Java durante a execução da aplicação.

O arquivo de configuração de propriedades é um componente importante para garantir que a aplicação se comporte como esperado e possa ser facilmente configurada de acordo com as necessidades do projeto. A figura 7 apresenta um exemplo de configuração de propriedades de acesso à base.

4.2.3 Mapeamento do Banco de Dados com JPA

JPA (Java *Persistence* API) é uma especificação Java que define como persistir objetos

Java em um banco de dados relacional. Ela fornece uma abstração para a camada de acesso a dados, o que permite ao desenvolvedor se concentrar em implementar a lógica de negócios da aplicação sem se preocupar com detalhes de implementação de banco de dados.

JPA funciona por meio da definição de entidades, que são classes Java mapeadas para tabelas do banco de dados. Cada atributo da classe é mapeado para uma coluna na tabela e cada instância da classe é mapeada para uma linha na tabela. JPA também fornece suporte para relacionamentos entre entidades, tais como relacionamentos um-para-um, um-para-muitos e muitos-para-muitos.

Para utilizar JPA no *spring*, é necessário especificar as configurações do banco de dados e as entidades mapeadas em um arquivo de configuração ou em anotações Java. JPA também fornece uma API para realizar operações de CRUD (*Create, Read, Update, Delete*) em entidades e para executar consultas em banco de dados.

JPA é uma solução popular para persistência de dados em aplicações Java, pois fornece uma maneira padronizada e fácil de realizar operações de banco de dados, como também permite a mudança do provedor de persistência sem afetar a lógica de negócios da aplicação, o que aumenta a flexibilidade e portabilidade da solução de persistência de dados.

Neste trabalho utilizou-se do JPA para implementar o modelo de entidade e relacionamento apresentado na subseção 4.7, ou seja, as anotações e configurações do JPA foram usadas para definir os mapeamentos de entidade, que foram usados para gerar o esquema do banco de dados do web service desenvolvido.

4.3 Clean architecture

A *Clean Architecture* é um padrão de arquitetura de software que visa separar as domínios de uma aplicação em diferentes camadas, garantindo a separação de responsabilidades e uma melhor organização do código.

Foi criada por Robert C. Martin em seu livro "Clean Architecture: A Craftsman's Guide to Software Structure and Design". A ideia por trás da *Clean Architecture* é criar um design de software flexível e escalável que possa ser facilmente mantido e expandido no futuro. Esse padrão de arquitetura é baseado nos princípios SOLID, que são cinco princípios de design fundamentais para escrever software fácil de manter, escalável e flexível. Com esse padrão, é possível isolar a lógica de negócios dos detalhes técnicos, como a interface do usuário, a camada de acesso a dados ou a infraestrutura. Essa separação de domínios torna mais fácil entender o sistema, testar o código e fazer alterações sem afetar outras partes do sistema.

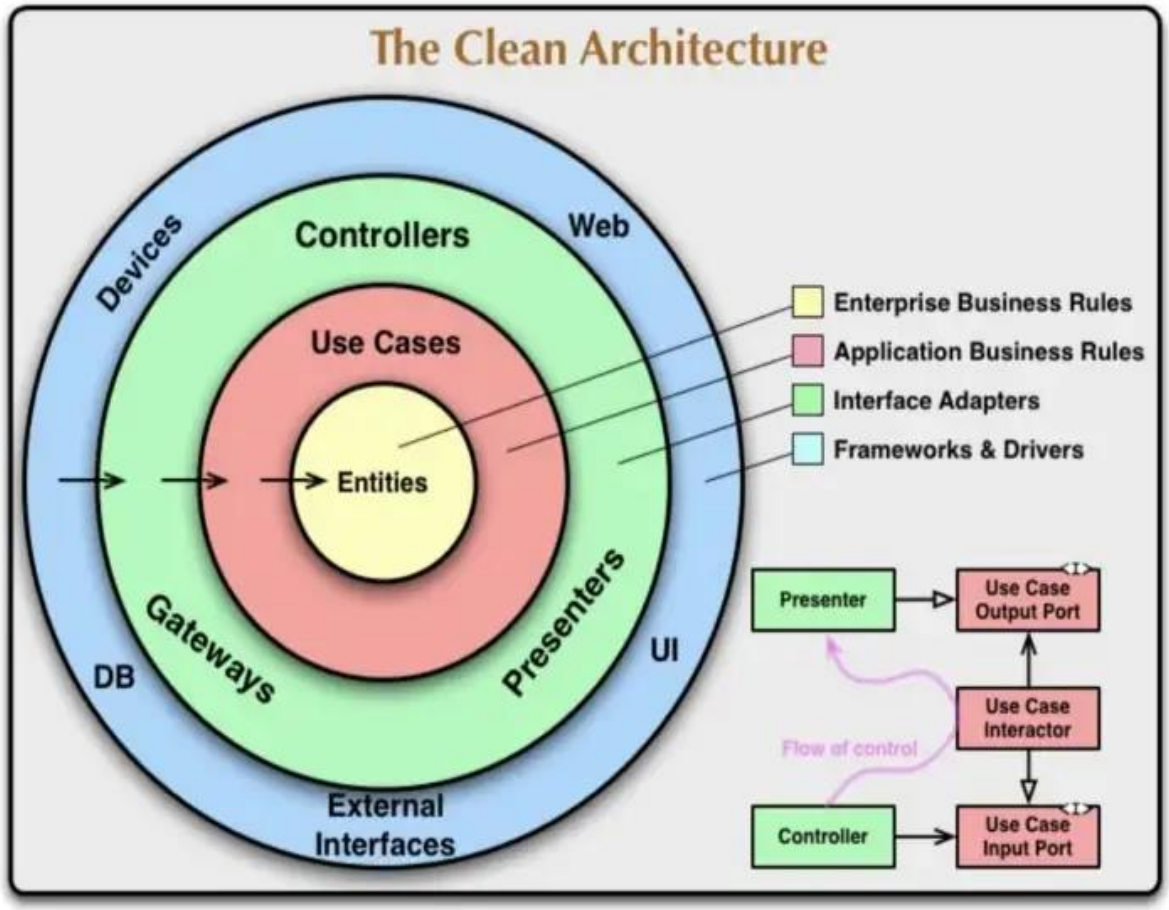
Outra vantagem *da Clean Architecture* é que ela é independente de tecnologia. A arquitetura não depende de nenhuma tecnologia ou plataforma específica, o que significa que

pode ser utilizada com qualquer *stack*. Isso torna a arquitetura reutilizável e adaptável, pois a lógica de negócios pode ser facilmente portada para diferentes plataformas e tecnologias.

A figura 8 apresenta a ilustração criada por Robert Martin da clean *architecture*. Nela podemos ver 4 camadas, cada uma com uma cor, função e responsabilidade específicas.

- Camada amarela: Representa a camada das entidades, como as entidades *de Product, Product Type, Questions* e todas as outras definidas na seção 4.7. Essas entidades definem as principais regras de negócios do sistema e são independentes de quaisquer fatores externos, como estruturas ou bancos de dados
- Camada vermelha: Representa a camada de Domínio. Essa camada é onde a lógica e as regras de negócios gerais com as interações entre as entidades são definidas, contém os casos de uso da aplicação e é independente de qualquer tecnologia ou estrutura. Essa camada se comunica com a camada de entidades e a camada de adaptadores de interface, mas não se comunica diretamente com a camada de *frameworks* e *drivers*
- Camada verde: Representa a camada de Adaptadores de Interface, ela é responsável por orquestrar as atividades das diferentes partes do sistema, incluindo as interações com a camada de apresentação e a camada de acesso a dados. Esta camada adapta os casos de uso e entidades aos frameworks e drivers que são requeridos pelo sistema.
- Camada azul: Representa a camada de apresentação ou *Frameworks* e *Drivers*, ela é responsável por receber as entradas do usuário e apresentar as saídas. Geralmente é implementado usando componentes de interface do usuário e APIs. Esta camada contém os componentes de baixo nível que fornecem ao sistema a infraestrutura necessária, como bancos de dados, servidores web e sistemas operacionais. As camadas de casos de uso e entidades não devem depender dessa camada, e as interfaces devem ser abstratas para que os componentes subjacentes possam ser trocados sem afetar o restante do sistema.

Figura 8 - Ilustração da Clean Architecture



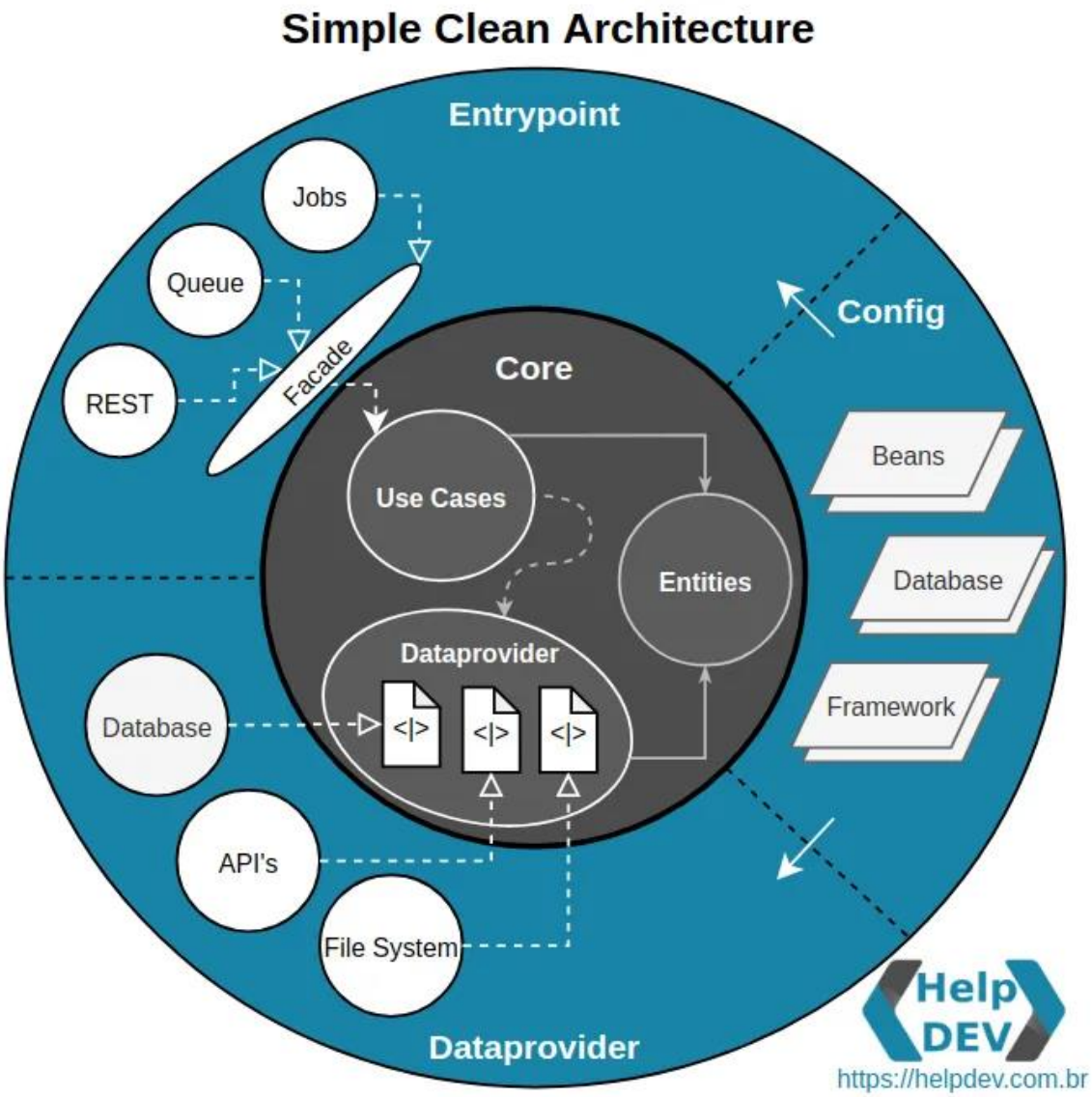
Fonte: https://betterprogramming.pub/the-clean-architecture-beginners-guide-e4b7058c1165, 2022.

Este padrão foi utilizado na organização e arquitetura interna do código fonte do *web service* visando obter classes com baixo acoplamento e melhor manutenibilidade do código.

4.3.1 Implementação

Para implementar a *Clean Architecture*, optou-se por organizar o código fonte dividindo-o em quatro diretórios: *core*, *dataprovider*, *entrypoint* e *configuration*, conforme o modelo *The Simple Clean Architecture* mostrado na figura 9

Figura 9 - Ilustração da Simple Clean Architecture



Fonte: https://helpdev.com.br/

A camada core é a camada mais interna e contém as regras de negócio do sistema. Essa camada é independente de qualquer tecnologia ou ferramenta utilizada e deve ser projetada para ser facilmente testável. Essa camada contém as classes de entidades, casos de uso, interfaces e exceções do sistema.

A camada *dataprovider* é responsável por fornecer os mecanismos de persistência e recuperação de dados para o sistema. Essa camada é responsável pela comunicação com o banco de dados ou outros mecanismos de armazenamento de dados e é composta por interfaces que definem as operações de leitura e escrita de dados e suas implementações.

A camada *entrypoint* é responsável por fornecer a interface de entrada para o sistema. Essa camada é responsável por expor as funcionalidades do sistema para os usuários ou outros

sistemas externos. Essa camada pode ser implementada como uma API REST, uma interface web, um serviço de mensagens ou outro mecanismo de comunicação.

A camada *configuration* é responsável por configurar e integrar todas as outras camadas do sistema. Essa camada contém as configurações de inicialização do sistema, as definições de injeção de dependência e outras configurações que permitem que as diferentes camadas sejam integradas de forma adequada.

A figura 10 apresenta o desfecho da organização interna do *web service*, uma vez que as separações por camadas mencionadas foram aplicadas na estrutura de diretórios da aplicação.

Figura 10 - Estrutura de pacotes da aplicação

Fonte: Elaborada pelo autor, 2023.

4.4 Modelagem dos *schemas* da api

Os *schemas* são uma parte fundamental da API, eles são usados para descrever a estrutura de dados que uma API espera receber ou retornar. Eles também podem ser usados para descrever as propriedades de objetos específicos, como parâmetros de solicitação ou respostas de API.

Os *schemas* do *OpenAPI* incluem suporte para vários tipos de dados, como números, *strings*, objetos e matrizes, e também podem incluir regras de validação, como tamanho mínimo e máximo, tipo de dado esperado e expressões regulares. Além disso, os *schemas* podem ser usados para definir relacionamentos entre objetos, como uma propriedade de objeto que deve ser um outro objeto específico.

Os *Schemas* são nomeados de acordo com a convenção de nomenclatura camelCase.

Nesta etapa foram modelados os seguintes *schemas*:

- *Product*: Representa o produto a ser avaliado
- *ProductType*: Representa o tipo do produto
- *Question*: Representa a questão que será respondida no momento da avaliação
- *Class*: Representa uma classe de uma pergunta, cada classe é uma heurística de Nilsen
- *Reference*: Representa o nome do material de onde a questão foi retirada
- *Answer*: Representa os dados de resposta para uma determinada pergunta
- *Questionnaire*: Representa um agrupamento de questões que devem ser respondidas na análise de um produto.
- *EvaluationRequest*: Representa uma solicitação para avaliar um produto.
- *Customer*: Representa um cliente que solicitou uma revisão do produto
- *Report*: Representa o resultado da avaliação

4.5 Modelagem dos *endpoints* da api

Os *endpointss* descrevem as diferentes operações que uma API pode realizar e os parâmetros que ela espera receber ou retornar. Eles são usados para mapear as requisições HTTP para as operações da API e podem ser usados para descrever as operações de leitura, escrita, atualização e exclusão de dados.

Os *endpoints* são definidos como uma coleção de caminhos e operações, cada um dos quais corresponde a uma URL específica na API. Cada operação é associada a uma requisição HTTP específica, como *GET, POST, PUT* ou *DELETE*, e pode especificar parâmetros de solicitação, como *query params* ou *path params*, e os tipos de respostas esperadas.

Além disso, os *endpoints* podem ser usados para descrever a autenticação e autorização necessárias para acessar determinadas operações, bem como os *schemas* de dados esperados para as solicitações e respostas. Isso ajuda a garantir que as requisições sejam precisas e completas antes de serem processadas pelo aplicativo.

Segue abaixo as urls dos *endpoints* que foram modelados nesta etapa:

- /questions: Disponibiliza operações sobre as questões.
- /questionnaires: Disponibiliza operações sobre os questionários utilizado para avaliar o produto.
- /request-evaluations: Disponibiliza operações sobre as requisições de avaliação.
- /reports: Disponibiliza operações sobre os relatórios de uma avaliação.
- /products: Disponibiliza operações sobre os produtos a serem avaliados.
- /customers: Disponibiliza operações sobre os clientes que solicitaram uma avaliação de

um determinado produto.

4.6 Criação do documento open api

O OpenAPI é uma especificação para descrever e documentar APIs RESTful. Através dessa especificação é possível criar um documento que descreve as funcionalidades e a estrutura das APIs de forma clara e precisa. Esse documento fornece uma maneira estandardizada de descrever as operações da API, incluindo *endpoints*, parâmetros de requisição e resposta, autenticação e autorização. Isso permite que os desenvolvedores de aplicativos consumidores entendam facilmente como usar a API e garante que as requisições sejam precisas e completas antes de serem processadas pelo aplicativo.

Além disso, o *OpenAPI* permite que os desenvolvedores criem ferramentas automatizadas para testar e validar as requisições e respostas da API, o que ajuda a garantir que a API esteja funcionando corretamente. Ele também permite que os desenvolvedores gerarem automaticamente a documentação da API, incluindo exemplos de código e modelos de requisição e resposta, o que facilita a compreensão e a implementação da API.

Outra vantagem do *OpenAPI* é que ele é independente de linguagem e plataforma, o que significa que as APIs podem ser documentadas usando essa especificação independentemente da linguagem ou plataforma de desenvolvimento utilizada. Isso permite que desenvolvedores de diferentes linguagens e plataformas possam entender e trabalhar com a API de forma eficiente.

O documento *OpenAPI* desenvolvido neste trabalho, encontra-se disponível no Apêndice A.

4.7 Modelagem da base de dados

4.7.1 Modelo de entidade e relacionamento

Foi utilizado o Modelo de Entidade e Relacionamento (ER) para definir e representar de forma gráfica a estrutura da base de dados modelada. O modelo ER é composto por entidades, que representam as tabelas, e relacionamentos, que representam as ligações entre as tabelas. De acordo com Peter Chen, considerado o "pai" do ER, "o modelo ER é um modelo conceitual que permite representar a estrutura lógica da base de dados de uma forma simples e intuitiva, independentemente da tecnologia de banco de dados utilizada." (Chen, 1976). O modelo ER é amplamente utilizado em diversas áreas, como engenharia de software,

gerenciamento de banco de dados, e sistemas de informação, e é considerado uma ferramenta fundamental para a modelagem de sistemas de informação. Além de Chen, outros autores como Barker, Umanath, e Silberschatz, também contribuíram para o desenvolvimento e evolução do modelo ER.

4.7.2 Mermaid

Para a geração do diagrama do modelo de entidade e relacionamento (ER) foi utilizado o mermaid, que é uma ferramenta de código aberto que permite criar diagramas de fluxo, gantt, entidade-relacionamento e outros tipos de diagramas de forma simples e intuitiva, utilizando uma linguagem de marcação de texto simples. Isso permite aos desenvolvedores incluir diagramas dentro dos arquivos de código e documentação, mantendo-os sempre atualizados e sincronizados com o código fonte.

A sintaxe do Mermaid é baseada em *Markdown*, o que facilita a escrita e a leitura dos códigos. Ele suporta a geração de imagens PNG e SVG, além de ser integrado com ferramentas de documentação como GitBook e Confluence. Ele é uma ferramenta gratuita e de fácil utilização, é uma boa opção para documentar e apresentar informações de forma visual.

4.7.3 Mermaid live editor

Utilizou-se do *Mermaid Live Editor*, uma website que permite a visualização em tempo real dos diagramas escritos pelas definições de texto inspiradas em *markdown* do mermaid. O *Mermaid Live Editor* possui outros recursos além do preview em tempo real, como exportação para imagem e integração com diversas plataformas, tornando-se uma opção útil para a visualização de dados e processos em projetos e documentações. A figura 11 mostra o código escrito *no mermaid live editor* para gerar o diagrama de entidade e relacionamento da figura 12.

Figura 11 - Código mermaid responsável por gerar o diagrama de entidade e relacionamento

```
erDiagram
    EVALUATION-REQUEST |{ -- || PRODUCT: evaluated
    EVALUATION-REQUEST |{ -- || CUSTOMER : request
    QUESTION |{ -- }| QUESTIONNAIRE: has
    REPORT |{ -- || QUESTIONNAIRE: has
    EVALUATION-REQUEST ||-- |{ REPORT: has
```

Fonte: Elaborada pelo autor, 2023.

Cada *statment* da figura 11 consiste nas seguintes partes:

<primeira-entidade> [<relacionamento> <segunda-entidade> : <rótulo-relação>]

- primeira-entidade: Nome de uma entidade. Os nomes devem começar com um caractere alfabético e também podem conter dígitos, hífens e sublinhados.
- relacionamento: Descreve a maneira como ambas as entidades se inter-relacionam.
- segunda-entidade: Nome da outra entidade.
- rótulo-relação: Descreve o relacionamento da perspectiva da primeira entidade.

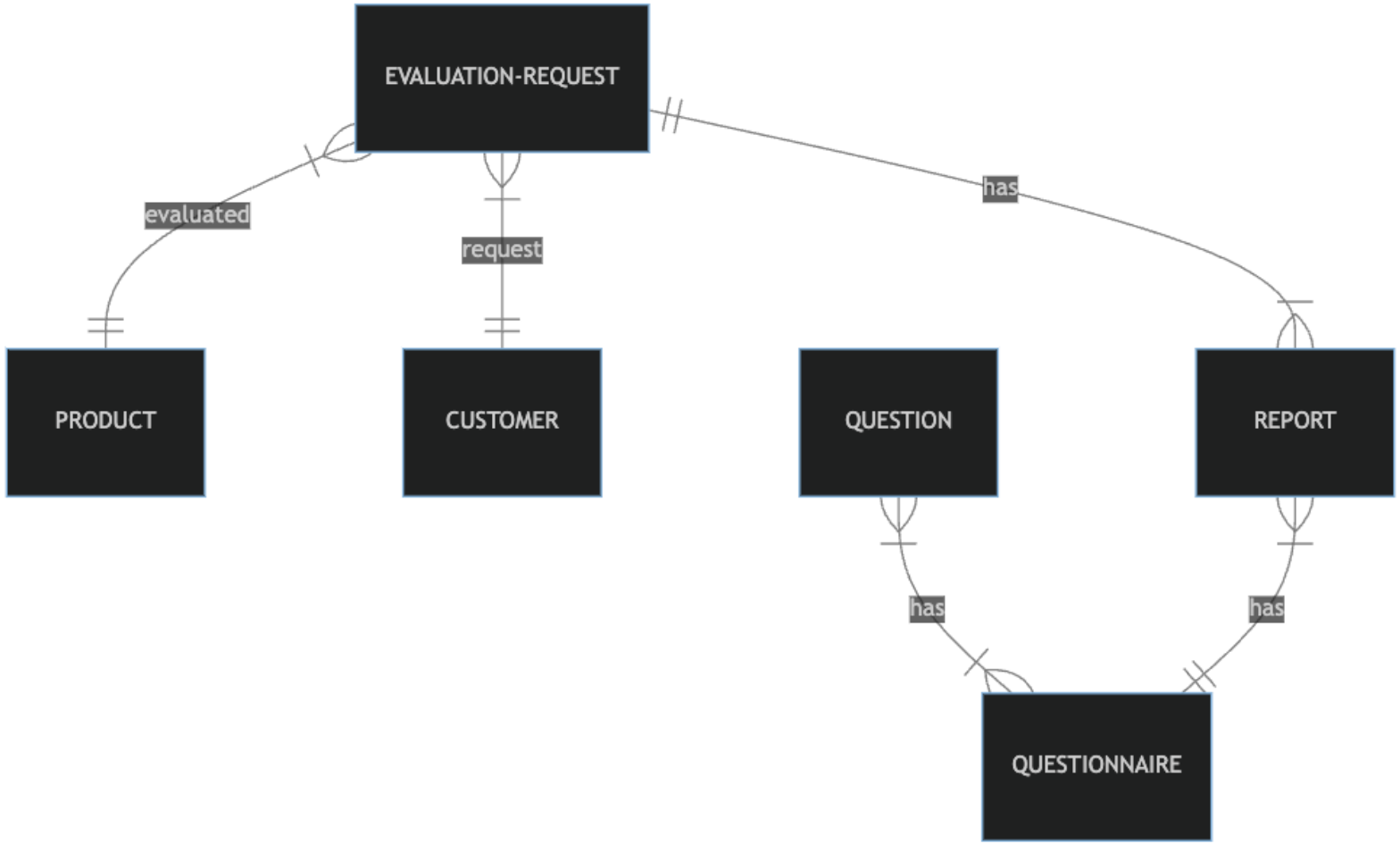
Há sempre dois caracteres no marcador de cardinalidade. O caractere mais externo representa um valor máximo e o caractere mais interno representa um valor mínimo. O quadro 2 abaixo resume as possíveis cardinalidades

Quadro 2 - Símbolos de cardinalidade do *mermaid* e seus significados

Valor (Esquerda)	**Valor (Direita)**	**Significado**
|O	O|	Zero ou um
||	||	Exatamente um
}O	O{	Zero ou mais
}|	|{	Um ou mais

Fonte: Elaborada pelo autor, 2023.

Figura 12 - Diagrama de entidade e relacionamento gerado via mermaid



Fonte: Elaborada pelo autor, 2023.

A figuras 12 apresenta todas as entidades e relações de cardinalidade entre elas. As entidades foram modeladas de acordo com os *schemas* definidos previamente na etapa de modelagem da api. Nas próximas subseções é apresentado os detalhes de cada entidade modelada e seus respectivos relacionamentos.

4.7.4 Entidade product

A entidade *Product* representa a tabela do produto a ser avaliado pela aplicação e possui uma relação de 1:N com a entidade *Evaluation-request*

4.7.5 Entidade evaluation-request

A entidade *Evaluation-request* representa a tabela de requisição de avaliação de um produto

- Relação de cardinalidade N:1 com as entidades *Product* e *Customer*, onde cada *Request Evaluation* pode ser solicitada para um único produto e por um único *Customer*, mas um mesmo produto pode estar associado a vários *Request Evaluations* assim como um único *Customer* pode solicitar várias *Request Evaluations.*
- Relação de cardinalidade 1:N com a entidade *Report*, onde cada Request Evaluation pode possuir vários *Reports*, mas um *Report* pode pertencer a apenas um Request Evaluation.

4.7.6 Entidade report

A entidade *Report* representa a tabela de relatório de avaliação da aplicação

- Relação de cardinalidade N:1 com as entidades *Request Evaluation* e *Questionnaire*, onde cada *Report* pode ser associado a um único *Request Evaluation* e a um único *Questionnaire*, mas um mesmo *Request Evaluation* pode estar associado a vários *Reports* assim como um único *Questionnaire* pode possuir vários *Reports.*

4.7.7 Entidade customer

A entidade *Customer* representa a tabela de clientes da aplicação

- Relação de cardinalidade 1:N entre *Customer* e *Evaluation-request*, onde um cliente pode solicitar N avaliações

4.7.8 Entidade question

A entidade *Question* representa a tabela de questões da aplicação

- Relação de cardinalidade N:N entre *Question* e *Questionnaire*, onde uma questão pode pertencer a vários grupos de questões e um grupo pode conter várias questões.

4.8 Code generator

Neste projeto, o *code generator* do *OpenAPI* foi utilizado para gerar os arquivos iniciais do projeto. Ele é uma ferramenta poderosa que permite aos desenvolvedores gerar rapidamente o esqueleto de uma API REST com base em um documento de especificação *OpenAPI*. Ao usar o gerador de código, os desenvolvedores podem economizar tempo e esforço que, de outra forma, seriam gastos escrevendo código manualmente.

Ele pode ser usado para gerar código para diferentes linguagens de programação, como Java, JavaScript, Python, Ruby, entre outras. Esse gerador automatiza a criação de código para acessar uma API, tornando mais fácil e rápido para os desenvolvedores implementarem suas funcionalidades. Ele gera códigos para o cliente e para o servidor, incluindo classes de modelo para cada entidade da API, código para validação de entrada e geração de respostas, além de exemplos de código para autenticação e outras operações comuns. Isso permite que os desenvolvedores se concentrem em escrever o código específico da aplicação, enquanto o gerador de código cuida de todos os aspectos de acesso à API.

O gerador de código *OpenAPI* dá suporte para uma variedade de tecnologias de servidor. Algumas das tecnologias de servidor mais comumente suportadas incluem Java com Spring, Python com Flask e Node.js com Express. Neste trabalho optou-se por utilizar o suporte do Java com Spring, que é uma das tecnologias de servidor mais populares do setor.

Utilizar o *code generator* do *OpenAPI* tem várias vantagens. Primeiro, permite a separação clara entre a especificação da API e a sua implementação. Isso facilita a manutenção da API, pois a especificação pode ser modificada sem afetar o código fonte. Além disso, o *code generator* gera código de alta qualidade, pois segue as boas práticas recomendadas pela comunidade *OpenAPI*.

Os seguintes artefatos de código foram gerados pelo *Code Generator*:

- Classes de modelo: Classes responsáveis por representar os schemas de dados definidos no arquivo de especificação *OpenAPI*.
- Interface API: Classes de interface que definem os *endpoints* RESTful com base no

arquivo de especificação *OpenAPI*.

- Controladores: Classes que implementam a interface API e fornecem uma implementação padrão para os *endpoints* RESTful.
- Classes de *Request* e *Response*: Classes que representam os *payloads* de *Request* e *Response* para cada *endpoint* RESTful
- Arquivos de configuração: Classes que são necessários para integrar o *OpenAPI* com o *Spring*

4.9 Documentação

Robert C. Martin, é um dos autores mais renomados no campo da engenharia de software. Em seu livro "Clean Code: A Handbook of Agile Software Craftsmanship", ele enfatiza a importância de escrever um código claro e bem documentado. Martin argumenta que a documentação é importante não apenas para ajudar outros a entender o código, mas também para ajudar o próprio desenvolvedor a lembrar por que ele escreveu o código de uma determinada maneira.

Fica claro que a documentação de software é uma parte essencial do desenvolvimento de um projeto, pois ajuda a garantir que o código seja compreendido e mantido com eficiência ao longo do tempo. Durante todo o desenvolvimento deste trabalho foram utilizadas diversas ferramentas de documentação.

Neste projeto, a utilização de várias ferramentas de documentação foi possível graças à adoção da abordagem API *First*, que tem como suas principais características começar o desenvolvimento pelo documento de contrato da API. Dessa forma, foi possível documentar o projeto desde o início do processo de desenvolvimento.

Para documentar a API utilizou-se o Swagger UI, que é uma interface de usuário para visualizar e testar APIs que são documentadas no formato *OpenAPI*. Ele fornece uma documentação interativa que permite explorar *endpoints*, enviar solicitações e ver as respostas correspondentes. O Swagger UI é útil para testar rapidamente uma API e ver como ela funciona em tempo real.

Para documentar as regras de negócios, foi utilizado o diagrama de atividades, que ajudou a descrever as diferentes etapas e fluxos do processo de negócios.

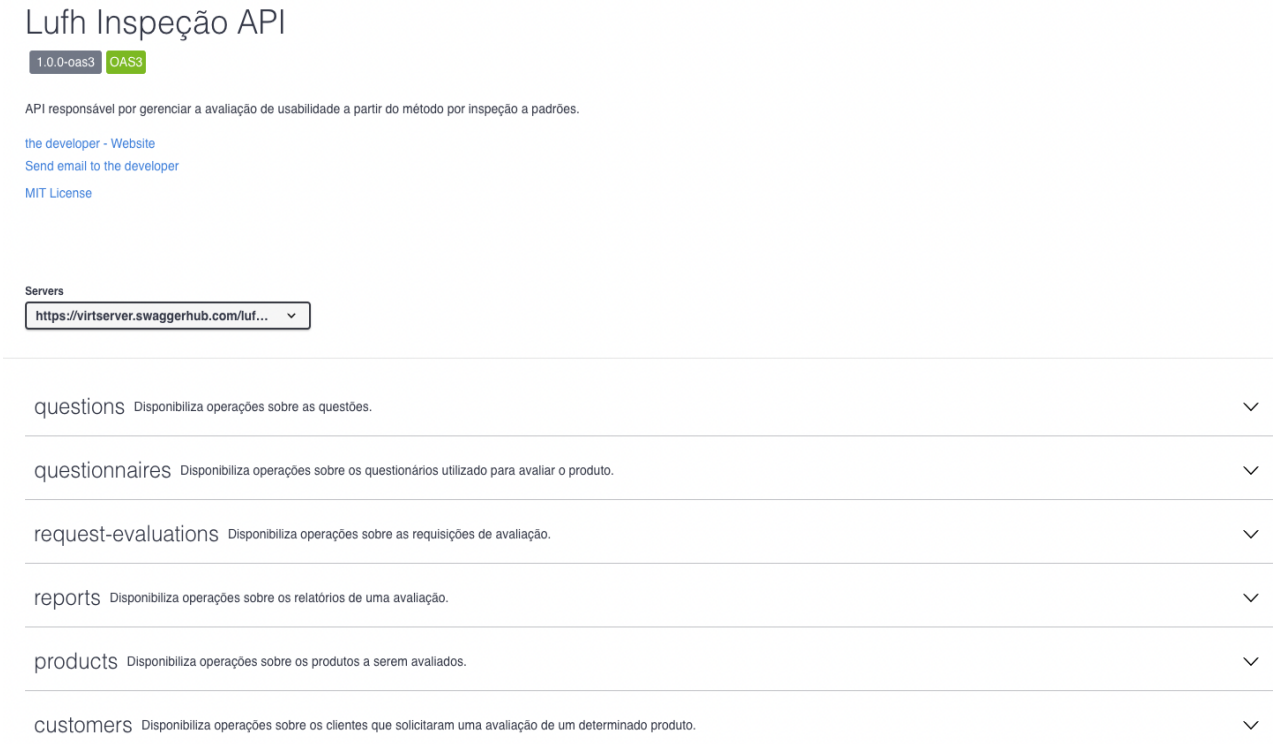
Além disso, para documentar a modelagem da base de dados, foi utilizado o diagrama de entidade e relacionamento (ER), que ajudou a descrever as entidades do sistema e os relacionamentos entre elas. O ER permitiu a visualização da estrutura da base de dados e as

relações entre as diferentes entidades, tornando mais fácil entender e documentar o modelo de dados.

O Git também foi utilizado desde o início do projeto. De acordo com o autor Eric Sink, em seu livro "Version Control by Example", o controle de versão é uma parte fundamental da documentação do software, pois registra as mudanças feitas no código-fonte e ajuda a entender a evolução do projeto ao longo do tempo.

O git foi utilizado tanto para versionar o código fonte quanto para centralizar todos os artefatos de documentação desenvolvidos via código, como os diagramas do *mermaid* e o documento *openapi*.

Figura 13 - Swagger UI do web service REST



Fonte: Elaborada pelo autor, 2023.

5 RESULTADOS

Com o webservice REST de automação de testes de usabilidade desenvolvido é possível gerenciar as avaliações de usabilidade baseadas em padrões de forma mais automatizada. A arquitetura baseada em API *first* permite que o software seja facilmente integrado a outras aplicações e a modelagem de banco de dados relacional garante a persistência dos dados de forma eficiente. Além disso, com o uso de *endpoints* REST e o *OpenAPI*, o software é facilmente acessível e mantível.

A utilização arquitetura *Clean Architecture* torna o software escalável e de fácil manutenção, enquanto a arquitetura REST permite a fácil integração com outras aplicações. Em suma, o software desenvolvido é uma ferramenta valiosa para garantir a qualidade da usabilidade de um sistema de forma automatizada, eficiente e escalável.

Com a utilização do webservice, é possível aumentar a eficiência do processo de aplicar os testes de usabilidade, permitindo assim a identificação de problemas de forma mais rápida e precisa.

As próximas seções apresentam imagens com as respostas obtidas pelo webservice REST por meio do software *Postman* e os respectivos comandos *Curls* utilizados em cada consulta.

5.1 Endpoint Customer

As próximas figuras apresentam o comportamento da API quando tentamos cadastrar um cliente

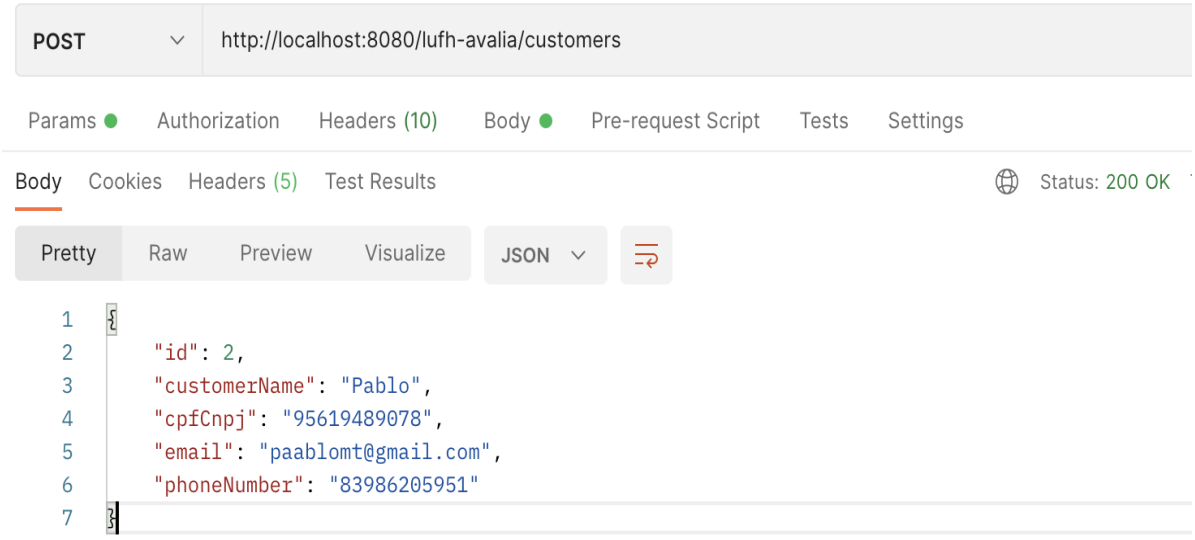
5.1.1 Cadastro de cliente válido

Figura 14 - Comando curl para Cadastro de cliente

```
curl --location --request POST 'http://localhost:8080/lufh-avalia/
    customers' \
--header 'Content-Type: application/json' \
--data-raw '{
  "customerName": "Pablo",
  "cpfCnpj": "956.194.890-78",
  "email": "paablomt@gmail.com",
  "phoneNumber": "83986205951"
}'
```

Fonte: Elaborada pelo autor, 2023.

Figura 15 - Tela do Postman com o Resultado da consulta de criação de um cliente válido



Fonte: Elaborada pelo autor, 2023.

A Figura 7 demonstra o comportamento da API ao tentar cadastrar um cliente com todos os dados inseridos válidos, é possível notar que a API retorna o *status code* 200, que por convenção significa que tudo correu bem com a solicitação HTTP.

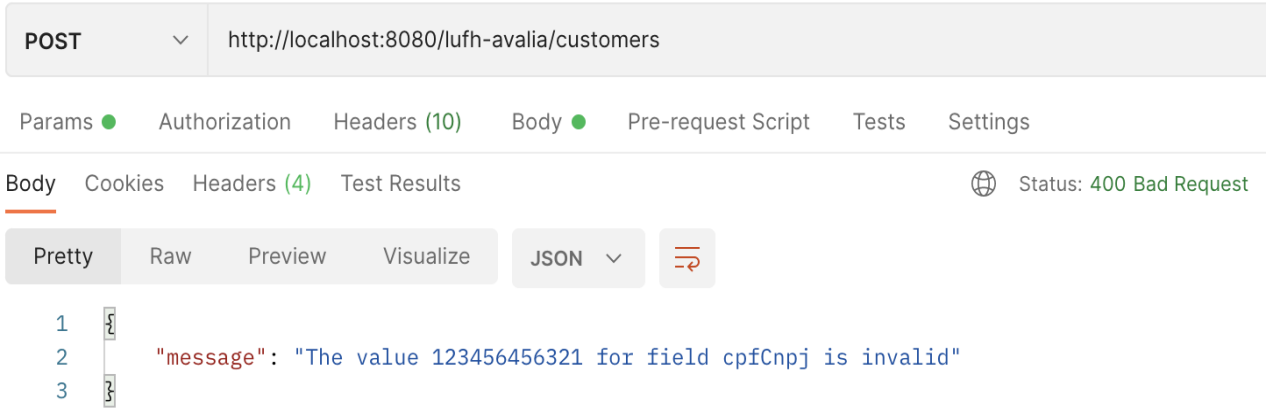
5.1.2 Cadastro de cliente inválido

Figura 16 - Comando curl para Cadastro de cliente inválido

```
curl --location --request POST 'http://localhost:8080/lufh-avalia/customers' \
--header 'Content-Type: application/json' \
--data-raw '{
  "customerName": "Pablo",
  "cpfCnpj": "123.456.456-321",
  "email": "paablomt@gmail.com",
  "phoneNumber": "83986205951"
}'
```

Fonte: Elaborada pelo autor, 2023.

Figura 17 - Tela do Postman com o Resultado da consulta de cadastro de cliente inválido



Fonte: Elaborada pelo autor, 2023.

A Figura 9 exemplifica o comportamento da API ao tentar cadastrar um cliente com CPF ou CNPJ inválidos. Nesse caso, a API retorna o *status code* 400, que indica a presença de erro na entrada de dados fornecidos pelo usuário.
O *payload* de resposta contém uma mensagem que explica qual campo e seu respectivo valor ocasionaram o erro 400. Além disso, a API também realiza a validação dos dados de e-mail e número de telefone fornecidos pelos usuários.

5.2 Endpoint Product

As próximas figuras apresentam o comportamento da API quando tentamos cadastrar um produto.

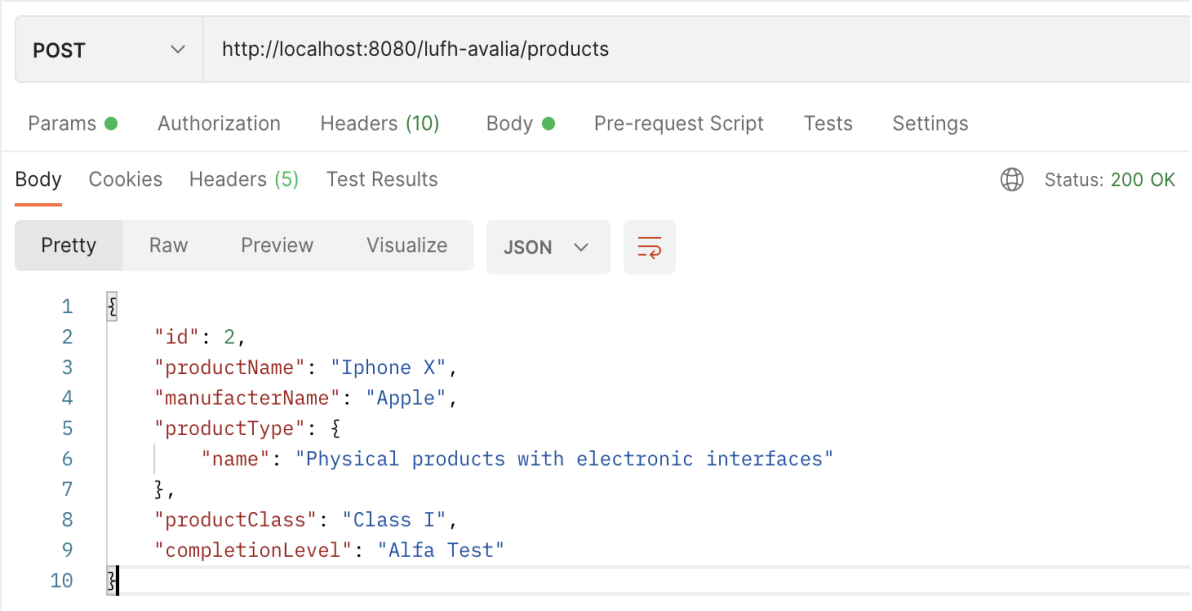
5.2.1 Cadastro de produto válido

Figura 18 - Comando curl para Cadastro de produto válido

```
curl --location --request POST 'http://localhost:8080/lufh-avalia/
    products' \
--header 'Content-Type: application/json' \
--data-raw '{
  "productName": "Iphone X",
  "manufacterName": "Apple",
  "productType": {
    "name": "Physical products with electronic interfaces"
  },
  "productClass": "Class I",
  "completionLevel": "Alfa Test"
}'
```

Fonte: Elaborada pelo autor, 2023.

Figura 19 - Tela do Postman com o Resultado da consulta de cadastro de produto válido



Fonte: Elaborada pelo autor, 2023.

A Figura 11 apresenta o comportamento da API ao tentar cadastrar um produto com todos os dados inseridos de forma válida. Observa-se que a API retorna o *status code* 200, que indica uma resposta bem-sucedida por parte do servidor.

5.2.2 Cadastro de produto inválido

Figura 20 - Comando curl para Cadastro de produto inválido

```
curl --location 'http://localhost:8080/lufh-avalia/products' \
--header 'Content-Type: application/json' \
--data '{
  "productName": "Iphone X",
  "manufacterName": "Apple",
  "productType": {
    "name": "Physical products with electronic interfaces"
  },
  "productClass": "Class one",
  "completionLevel": "Alfa Test"
}'
```

Fonte: Elaborada pelo autor, 2023.

Figura 21 - Tela do Postman com o Resultado da consulta de cadastro de produto inválido

Fonte: Elaborada pelo autor, 2023.

A Figura 21 exemplifica o comportamento da API ao tentar cadastrar um produto com o campo productClass com valor inválido. Os valores aceitos pela API para esse campo são: Class I, Class II, Class III e Class IV. O payload de resposta contém uma mensagem que explica qual campo e seu respectivo valor ocasionaram o erro 400. Além disso, a API também realiza a validação dos valores passados para com os campos productType e completionLevel. convém checar o anexo B deste material com os valores válidos para cada campo.

5.3 Endpoint Request Evaluations

As próximas figuras apresentam o comportamento da API quando tentamos cadastrar uma requisição de avaliação.

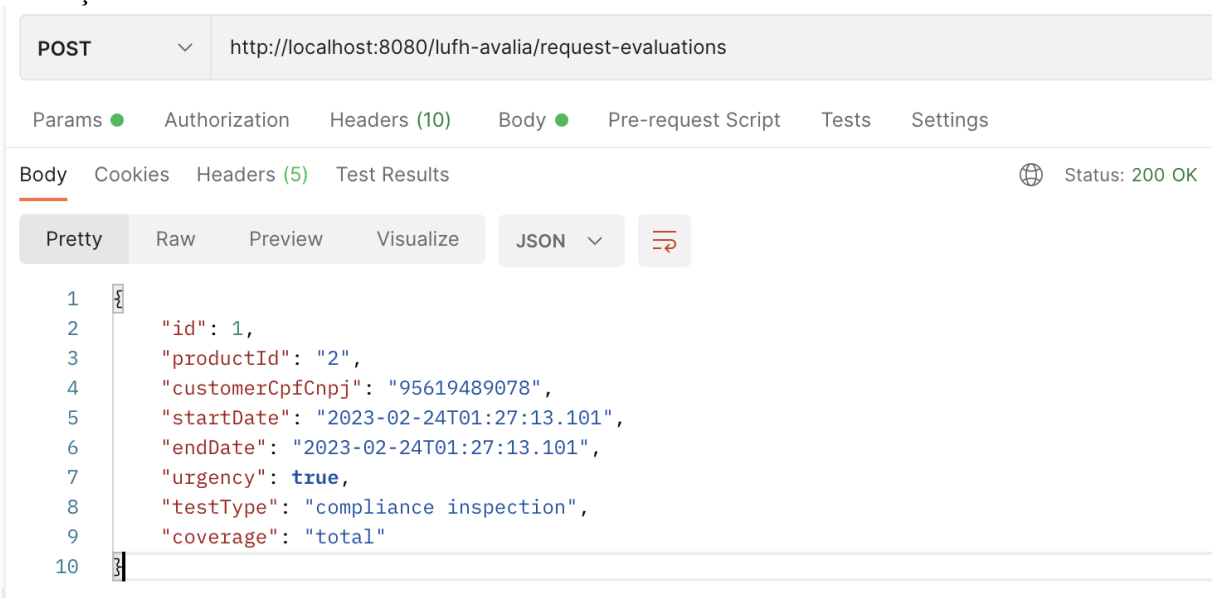
5.3.1 Cadastro de Requisição de avaliação para um produto válido

Figura 22 - Comando curl para Cadastro de requisição de avaliação de um produto previamente cadastrado

```
curl --location --request POST 'http://localhost:8080/lufh-avalia/
    request-evaluations' \
--header 'product_id: 2' \
--header 'customer_cpf_cnpj: 956.194.890-78' \
--header 'Content-Type: application/json' \
--data-raw '{
  "startDate": "2023-02-24T01:27:13.101Z",
  "endDate": "2023-02-24T01:27:13.101Z",
  "urgency": true,
  "testType": "compliance inspection",
  "coverage": "total"
}'
```

Fonte: Elaborada pelo autor, 2023.

Figura 23 - Tela do Postman com o Resultado da consulta de cadastro de requisição de avaliação



Fonte: Elaborada pelo autor, 2023.

A Figura 23 ilustra o comportamento da API ao tentar cadastrar uma requisição de avaliação com todos os dados inseridos válidos. Neste cenário, a API retorna o status code 404, que indica que tudo correu bem com a solicitação HTTP

5.3.2 Cadastro de Requisição de avaliação para um produto inválido

Figura 24 - Tela do Postman com o Resultado da consulta de cadastro de requisição de avaliação

```
curl --location --request POST 'http://localhost:8080/lufh-avalia/
    request-evaluations' \
--header 'product_id: 999' \
--header 'customer_cpf_cnpj: 956.194.890-78' \
--header 'Content-Type: application/json' \
--data-raw '{
  "startDate": "2023-02-24T01:27:13.101Z",
  "endDate": "2023-02-24T01:27:13.101Z",
  "urgency": true,
  "testType": "compliance inspection",
  "coverage": "total"
}'
```

Fonte: Elaborada pelo autor, 2023.

Figura 25 - Tela do Postman com o Resultado da consulta de cadastro de requisição de avaliação para um produto inválido

Fonte: Elaborada pelo autor, 2023.

A Figura 25 ilustra o comportamento da API ao tentar cadastrar uma requisição de avaliação para um produto que ainda não foi cadastrado na base de dados. Neste cenário, a API retorna o status *code* 404, que indica que o recurso não foi encontrado no servidor.

A resposta da API é composta por um *payload* que traz uma mensagem explicando qual campo e valor específicos não foram encontrados no servidor. Essa validação é realizada não apenas para o produto, mas também para o CPF ou CNPJ do cliente.

5.4 Endpoint Question

As próximas figuras apresentam o comportamento da API quando tentamos cadastrar uma questão na base de dados

Figura 26 - Comando curl para Cadastro de uma questão na base de dados da api

```
cURL
curl --location 'http://localhost:8080/lufh-avalia/questions' \
--header 'Content-Type: application/json' \
--data '{
  "class": {
    "name": "Visibility of system status"
  },
  "productType": "Hardware",
  "baseQuestion": "Is every user action followed by a perceivable system feedback?",
  "detailedQuestion": "Is there visual feedback when objects are selected or moved?"
}'
```

Fonte: Elaborada pelo autor, 2023.

Figura 27 - Tela do Postman com o Resultado da consulta de cadastro de questão

Fonte: Elaborada pelo autor, 2023.

5.5 Endpoint Questionnaires

As próximas figuras apresentam o comportamento da API quando tentamos cadastrar um questionário de perguntas.

Figura 28 - Comando curl para Cadastro de um questionário na base de dados da api

```
cURL
curl --location 'http://localhost:8080/lufh-avalia/questionnaires' \
--header 'Content-Type: application/json' \
--data '{
  "questionIds": [
    "2", "3", "4"
  ],
  "creator": "Pablo Monteiro Santos",
  "title": "Questionário teste"
}'
```

Fonte: Elaborada pelo autor, 2023.

Figura 29 - Tela do Postman com o Resultado da consulta de cadastro de questionário


```
{
    "id": 2,
    "questions": [
        {
            "id": 2,
            "class": {
                "name": "Aesthetic and minimalist design"
            },
            "productType": "Software",
            "baseQuestion": "Is it designed minimal?",
            "detailedQuestion": "\"Is only (and all) information, essential to decision making, displayed on the screen?\""
        },
        {
            "id": 3,
            "class": {
                "name": "Aesthetic and minimalist design"
            },
            "productType": "Software",
            "baseQuestion": "Is it designed minimal?",
            "detailedQuestion": "\"If visual elements (such as arrows or lines) are used, do they provide extra information?\""
        },
        {
            "id": 4,
            "class": {
                "name": "Aesthetic and minimalist design"
            },
            "productType": "Software",
            "baseQuestion": "Is it designed minimal?",
            "detailedQuestion": "\"Are there enough function keys to support functionality, but not so many that scanning and finding are difficult?\""
        }
    ],
    "creator": "Pablo Monteiro Santos",
    "title": "Questionário teste"
}
```


Fonte: Elaborada pelo autor, 2022.

5.6 Endpoint Reports

As próximas figuras apresentam o comportamento da API quando tentamos cadastrar um relatório de avaliação

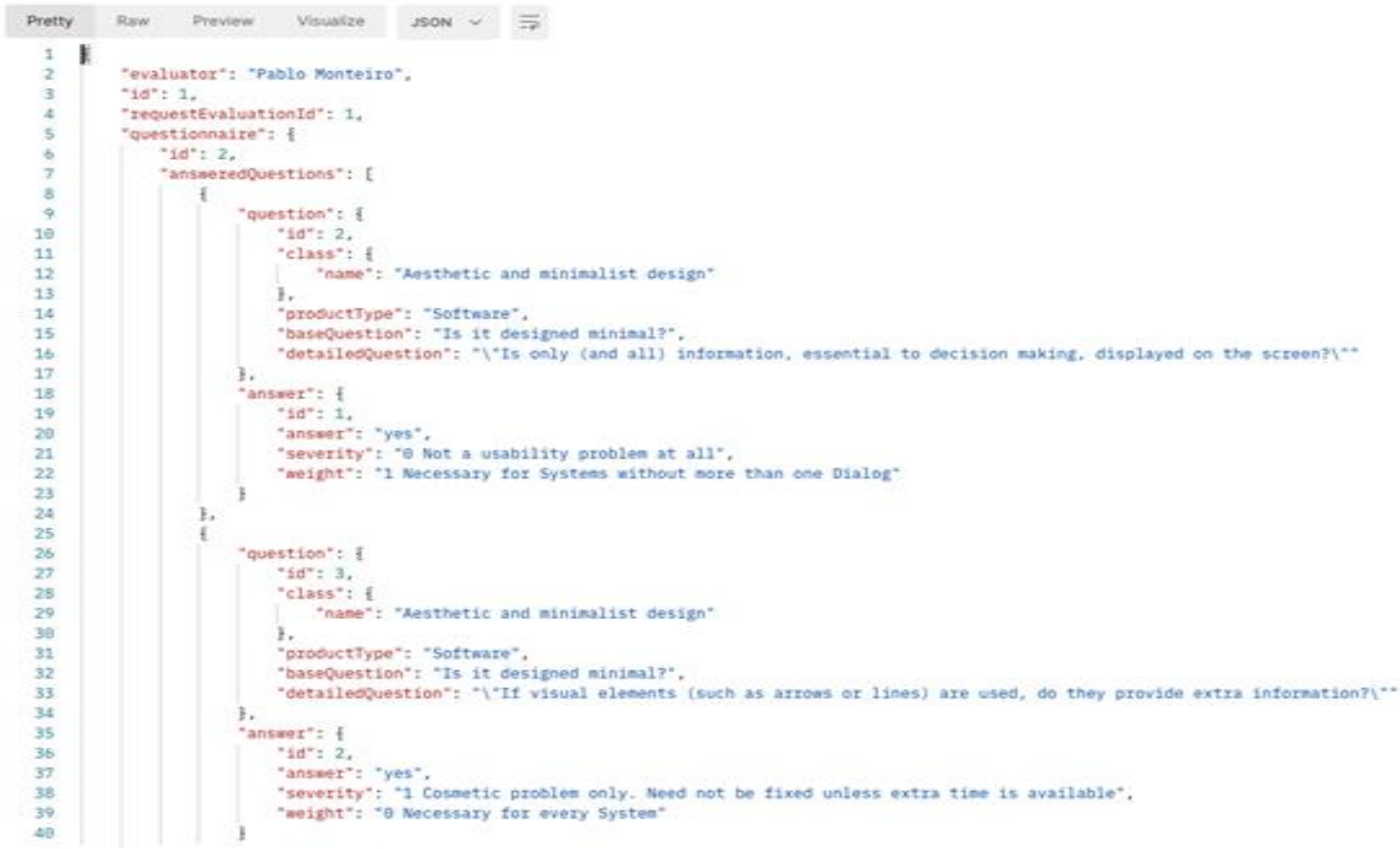
Figura 30 - Comando curl para Cadastro de um relatório na base de dados da api

cURL

```
curl --location 'http://localhost:8080/lufh-avalia/reports' \
--header 'Content-Type: application/json' \
--data '{
  "evaluator": "Pablo Monteiro",
  "requestEvaluationId": 1,
  "questionnaireId": 2,
  "answers": [
    {
      "answer": "yes",
      "severity": "0 Not a usability problem at all",
      "weight": "1 Necessary for Systems without more than one Dialog"
    },
    {
      "answer": "yes",
      "severity": "1 Cosmetic problem only. Need not be fixed unless extra time is available",
      "weight": "0 Necessary for every System"
    },
    {
      "answer": "no",
      "severity": "1 Cosmetic problem only. Need not be fixed unless extra time is available",
      "weight": "3 Necessary for Complex Systems with data have to be entered on several dialogs"
    }
  ]
}'
```

Fonte: Elaborada pelo autor, 2023.

Figura 31 - Tela do Postman com o Resultado da consulta de cadastro de relatório



Fonte: Elaborada pelo autor, 2023.

6 CONCLUSÃO

O presente trabalho teve como objetivo criar um *web service* REST utilizando o método API *First* para automatizar os testes de usabilidade através do método de inspeção a padrões. Ao longo deste estudo, foi possível explorar a importância dos testes de usabilidade na melhoria da experiência do usuário em produtos e serviços, bem como os desafios e custos associados à realização desses testes manualmente.

O método API *First* revelou-se uma abordagem valiosa para o desenvolvimento do *web service*, pois enfatizou a concepção da API antes da implementação do serviço propriamente dito. Essa abordagem permitiu a criação de um serviço altamente integrável, com suporte a diferentes plataformas e tecnologias, tornando-o uma solução eficiente e escalável para automatizar o processo de testes de usabilidade em qualquer tipo de produto.

Ao longo da implementação do *web service*, foram utilizadas as mais recentes tecnologias e ferramentas disponíveis, proporcionando um ambiente de desenvolvimento ágil e produtivo.

Os cenários de uso apresentados demonstraram a versatilidade *do web service*, possibilitando sua aplicação em uma variedade de situações, desde aplicações web e mobile até sistemas complexos e em constante evolução. A automação dos testes de usabilidade proporcionará ganhos significativos em eficiência e precisão na identificação e correção de problemas, resultando em produtos finais mais agradáveis e satisfatórios para os usuários.

Em síntese, este trabalho contribui para a área de testes de usabilidade, fornecendo uma solução inovadora e eficiente para automatizar o processo de avaliação da experiência do usuário em produtos e serviços. O *web service* criado através do método API *First* destaca-se como uma ferramenta poderosa que pode ser facilmente integrada em ambientes de desenvolvimento ágeis e promover melhorias significativas na qualidade dos produtos oferecidos aos usuários.

6.1 Trabalhos futuros

Uma das principais perspectivas para trabalhos futuros consiste na criação de uma interface visual e intuitiva que permita aos usuários interagir com o *web service* de forma mais direta. A implementação de uma interface gráfica possibilitaria a configuração de parâmetros de teste, o acompanhamento dos resultados e a visualização de métricas e insights sobre a usabilidade dos produtos avaliados. Essa interface visual poderia ser desenvolvida utilizando

tecnologias modernas de *frontend* e seria um importante passo para facilitar a adoção e utilização do *web service* em diferentes cenários. Outra abordagem promissora para a evolução do *backend* criado é a implementação de uma fila de espera de prioridades para gerenciar as requisições de avaliação de usabilidade. Essa fila permitiria priorizar os testes de acordo com critérios específicos, como o perfil dos usuários, a importância do produto ou a criticidade dos problemas identificados. Dessa forma, o serviço poderia otimizar os recursos disponíveis, gerando uma abordagem mais eficiente e inteligente para a realização dos testes.

O uso de técnicas de escalonamento e priorização poderia ser incorporado ao *backend* para garantir um melhor gerenciamento dos recursos e um equilíbrio entre a demanda e a capacidade do sistema. Além disso, a expansão do *web service* para oferecer suporte a diferentes métodos de inspeção de usabilidade também é uma oportunidade de desenvolvimento. Atualmente, o serviço está focado no método de inspeção a padrões, porém, a inclusão de outros métodos, como testes de usabilidade com usuários reais, análise de métricas de interação ou análise de calor, poderia enriquecer ainda mais a abordagem e tornar o serviço mais completo e abrangente. Por fim, a realização de estudos de caso e validações empíricas em um maior número de produtos e cenários de uso também seria um caminho relevante para a consolidação e a comprovação da eficácia do *web service* desenvolvido. A coleta de feedback dos usuários, a comparação com outras ferramentas de testes de usabilidade e a análise de resultados a longo prazo seriam contribuições valiosas para fortalecer a confiabilidade e a aplicabilidade do serviço.

REFERÊNCIAS

DAVID A. CHAPPELL;JEWELL T. **Java Web Services**. mar 2002.
KALIN, MARTIN. **Java Web Services: Up and Running. 1st Edition. 2009. O'Reilly Media, Inc. 336p**. (tradução nossa).

LECHETA R. R. **Web Services Restful: Aprenda a Criar web Services RESTful em Java na Nuvem do Google**. Set de 2015.

NIELSEN, J. **Usability 101: Introduction to Usability**. NN/G Nielsen Norman Group, 4 jan 2012.

POP P. D, Altar A. **Designing an MVC Model for Rapid Web Application Development**. Mar 2014.

SOMMERVILLE, IAN. Engenharia de software. 9.ed. São Paulo. Addison Wesley, 2011.

RapidAPI. The State of API Integration Report. 2020.
Ochs, Michael. The Benefits of Code First Approaches for API Development. 2021.
Eisenberg, John. "Why you should consider an API first approach". 2019. Disponível em: https://nordicapis.com/why-you-should-consider-an-api-first-approach/. Acesso em: 09 fev. 2023.

Lim, Y., Kurniawan, S., & Khomh, F. (2019). A systematic literature review on the automation of usability testing. Information and Software Technology, 105, 66-83.
Nielsen, J. (1993). Usability engineering. San Francisco, CA: Morgan Kaufmann Publishers

Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine.

APÊNDICE A – DOCUMENTO OPEN API

```
---
openapi: 3.0.0
info:
  title: Lufh Inspeção API
  description: API responsável por gerenciar a avaliação de usabilidade a
partir do método por inspeção a padrões.
  contact:
    url: https://www.linkedin.com/in/pablo-monteiro-santos/
    email: paablomt@gmail.com
  license:
    name: MIT License
    url: http://choosealicense.com/licenses/mit/
  version: 1.0.0-oas3
servers:
  - url: https://virtserver.swaggerhub.com/lufh-avalia/
tags:
  - name: questions
    description: Disponibiliza operações sobre as questões.
  - name: questionnaires
    description: Disponibiliza operações sobre os questionários utilizado
para avaliar o produto.
  - name: request-evaluations
    description: Disponibiliza operações sobre as requisições de
avaliação.
  - name: reports
    description: Disponibiliza operações sobre os relatórios de uma
avaliação.
  - name: products
    description: Disponibiliza operações sobre os produtos a serem
avaliados.
  - name: customers
    description: Disponibiliza operações sobre os clientes que solicitaram
uma avaliação de um determinado produto.
paths:
  /questions:
    get:
```

```
tags:
  - questions
summary: Finds all questions
operationId: findQuestions
parameters:
  - name: class
    in: query
    description: Class values that can be considered for filter
    required: false
    style: form
    explode: true
    schema:
      type: string
      enum:
        - Aesthetic and minimalist design
        - Consistency and standards
        - Error prevention
        - Flexibility and efficiency of use
        - Help and documentation
        - Help users recognize and recover from errors
        - Match between system and the real world
        - Recognition rather than recall
        - User control and freedom
        - Visibility of system status
        - Privacy & Security
  - name: reference
    in: query
    description: Reference values that can be considered for filter
    required: false
    style: form
    explode: true
    schema:
      type: string
      enum:
        - Article
        - Book
        - eDocument
        - Standard
  - name: product_type
    in: query
    description: Product type values that can be considered for
```

```
filter
          required: false
          style: form
          explode: true
          schema:
            type: string
            enum:
              - Software
              - Hardware
              - Hardware/Software
      responses:
        "200":
          description: successful operation
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/QuestionOutput'
        "404":
          description: Not found
    post:
      tags:
        - questions
      summary: Add a new question
      operationId: createQuestion
      requestBody:
        description: Question object that needs to be added to the form
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/QuestionInput'
        required: true
      responses:
        "500":
          description: Internal server error
  /questions/{question_id}:
    get:
      tags:
        - questions
      summary: Find question by ID
```

```
    operationId: findQuestion
    parameters:
      - name: question_id
        in: path
        description: ID of question to return
        required: true
        style: simple
        explode: false
        schema:
          type: integer
          format: int64
    responses:
      "200":
        description: successful operation
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/QuestionOutput'
      "400":
        description: Invalid ID supplied
      "404":
        description: Question not found
  put:
    tags:
      - questions
    summary: Update an existing question
    operationId: updateQuestion
    parameters:
      - name: question_id
        in: path
        description: Question id to update
        required: true
        style: simple
        explode: false
        schema:
          type: integer
          format: int64
    requestBody:
      description: Question object that needs to be added to the form
      content:
        application/json:
```

```
          schema:
            $ref: '#/components/schemas/QuestionInput'
      required: true
    responses:
      "400":
        description: Invalid ID supplied
      "404":
        description: Question not found
      "405":
        description: Validation exception
  delete:
    tags:
      - questions
    summary: Deletes a question
    operationId: deleteQuestion
    parameters:
      - name: question_id
        in: path
        description: Question id to delete
        required: true
        style: simple
        explode: false
        schema:
          type: integer
          format: int64
    responses:
      "400":
        description: Invalid ID supplied
      "404":
        description: Question not found
/questionnaires:
  get:
    tags:
      - questionnaires
    summary: Finds all questionnaires
    operationId: findQuestionnaires
    parameters:
      - name: evaluator
        in: query
        description: Evaluator values that can be considered for filter
        required: false
```

```
      style: form
      explode: true
      schema:
        type: string
    - name: creator
      in: query
      description: Creator values that can be considered for filter
      required: false
      style: form
      explode: true
      schema:
        type: string
  responses:
    "200":
      description: successful operation
      content:
        application/json:
          schema:
            type: array
            items:
              $ref: '#/components/schemas/QuestionnaireInput'
    "404":
      description: Not found
post:
  tags:
    - questionnaires
  summary: Add a new questionnaire
  operationId: createQuestionaire
  requestBody:
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/QuestionnaireInput'
  responses:
    "200":
      description: A QuestionnaireOutput object
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/QuestionnaireOutput'
    "500":
```

```
        description: Internal server error
/questionnaires/{questionnaire_id}:
  get:
    tags:
      - questionnaires
    summary: Find questionnaire by ID
    operationId: findQuestionnaire
    parameters:
      - name: questionnaire_id
        in: path
        description: ID of questionnaire to return
        required: true
        style: simple
        explode: false
        schema:
          type: integer
          format: int64
    responses:
      "200":
        description: successful operation
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/QuestionnaireInput'
          application/xml:
            schema:
              $ref: '#/components/schemas/QuestionnaireInput'
      "400":
        description: Invalid ID supplied
      "404":
        description: Questionnaire not found
  put:
    tags:
      - questionnaires
    summary: Update an existing questionnaire
    operationId: updateQuestionnaire
    parameters:
      - name: questionnaire_id
        in: path
        description: Questionnaire id to update
        required: true
```

```
        style: simple
        explode: false
        schema:
          type: integer
          format: int64
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/QuestionnaireOutput'
      required: true
    responses:
      "400":
        description: Invalid ID supplied
      "404":
        description: Questionnaire not found
      "405":
        description: Validation exception
  delete:
    tags:
      - questionnaires
    summary: Deletes a questionnaire
    operationId: deleteQuestionnaire
    parameters:
      - name: questionnaire_id
        in: path
        description: Questionnaire id to delete
        required: true
        style: simple
        explode: false
        schema:
          type: integer
          format: int64
    responses:
      "400":
        description: Invalid ID supplied
      "404":
        description: Questionnaire not found
/request-evaluations:
  get:
    tags:
```

```
        - request-evaluations
      summary: Finds all request-evaluations
      operationId: findRequestEvaluations
      parameters:
        - name: customer_cpf_cnpj
          in: query
          description: CPF and CNPJ values that can be considered for
filter
          required: false
          style: form
          explode: true
          schema:
            type: string
      responses:
        "200":
          description: successful operation
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/RequestEvaluationOutput'
        "400":
          description: Bad Request
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
              examples:
                CpfCnpjInvalid:
                  summary: customer_cpf_cnpj invalid
                  value:
                    message: customer_cpf_cnpj invalid
        "404":
          description: Not found
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
              examples:
                RequestEvaluationNotFound:
```

```
              summary: Request Evaluation not found
              value:
                message: Request Evaluation not found
post:
  tags:
    - request-evaluations
  summary: Add a new request-evaluation
  operationId: createRequestEvaluation
  parameters:
    - name: product_id
      in: header
      required: true
      style: simple
      explode: false
      schema:
        type: string
    - name: customer_cpf_cnpj
      in: header
      required: true
      style: simple
      explode: false
      schema:
        type: string
  requestBody:
    description: Request evaluation object
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/RequestEvaluationInput'
    required: true
  responses:
    "200":
      description: A RequestEvaluationOutput object
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/RequestEvaluationOutput'
    "500":
      description: Internal server error
    "400":
      description: Bad Request
```

```
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Error'
            examples:
              CpfCnpjInvalid:
                summary: customer_cpf_cnpj invalid
                value:
                  message: customer_cpf_cnpj invalid
      "404":
        description: Not found
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Error'
            examples:
              ProductNotFound:
                summary: Product not found
                value:
                  message: Product not found
/request-evaluations/{request_evaluation_id}:
  get:
    tags:
      - request-evaluations
    summary: Find request evaluation by ID
    operationId: findRequestEvaluation
    parameters:
      - name: request_evaluation_id
        in: path
        description: ID of request evaluation to return
        required: true
        style: simple
        explode: false
        schema:
          type: integer
          format: int64
    responses:
      "200":
        description: successful operation
        content:
          application/json:
```

```
            schema:
              $ref: '#/components/schemas/RequestEvaluationOutput'
        links:
          FindProductByProductId:
            operationId: findProduct
            parameters:
              product_id: $response.body#/productId
            description: |
              The 'productId' value returned in the response can be used
as the  'product_id' in 'GET /products/{product_id}
      "404":
        description: Not found
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Error'
            examples:
              RequestEvaluationNotFound:
                summary: Request Evaluation not found
                value:
                  message: Request Evaluation not found
  put:
    tags:
      - request-evaluations
    summary: Update an existing request evaluation
    operationId: updateRequestEvaluation
    parameters:
      - name: request_evaluation_id
        in: path
        description: Request Evaluation id to update
        required: true
        style: simple
        explode: false
        schema:
          type: integer
          format: int64
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/RequestEvaluationInput'
```

```
    required: true
  responses:
    "404":
      description: Not found
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
          examples:
            RequestEvaluationNotFound:
              summary: Request Evaluation not found
              value:
                message: Request Evaluation not found
delete:
  tags:
    - request-evaluations
  summary: Deletes a request evaluation
  operationId: deleteRequestEvaluation
  parameters:
    - name: request_evaluation_id
      in: path
      description: Request Evaluation id to delete
      required: true
      style: simple
      explode: false
      schema:
        type: integer
        format: int64
  responses:
    "204":
      description: No content
    "404":
      description: Not found
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
          examples:
            RequestEvaluationNotFound:
              summary: Request Evaluation Not Found
              value:
```

```
                    message: Request Evaluation Not Found
  /reports:
    get:
      tags:
        - reports
      summary: Finds all reports
      operationId: findReports
      parameters:
        - name: request_evaluation_id
          in: query
          description: Request evaluation id values that can be considered
for filter
          required: false
          style: form
          explode: true
          schema:
            type: string
        - name: product_type
          in: query
          description: Product type values that can be considered for
filter
          required: false
          style: form
          explode: true
          schema:
            type: string
            enum:
              - Software
              - Hardware
              - Hardware/Software
        - name: report_date
          in: query
          description: Report date values that can be considered for
filter
          required: false
          style: form
          explode: true
          schema:
            type: string
      responses:
        "200":
```

```
        description: successful operation
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/ReportOutput'
      "404":
        description: Not found
  post:
    tags:
      - reports
    summary: Add a new report
    operationId: createReport
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/ReportInput'
      required: true
    responses:
      "200":
        description: successful operation
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ReportOutput'
/reports/{report_id}:
  get:
    tags:
      - reports
    summary: Find a report by ID
    operationId: findReport
    parameters:
      - name: report_id
        in: path
        description: ID of report to return
        required: true
        style: simple
        explode: false
        schema:
```

```
        type: integer
        format: int64
    responses:
      "200":
        description: successful operation
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ReportOutput'
      "400":
        description: Invalid ID supplied
      "404":
        description: Report not found
  put:
    tags:
      - reports
    summary: Update an existing report
    operationId: updateReport
    parameters:
      - name: report_id
        in: path
        description: Report id to update
        required: true
        style: simple
        explode: false
        schema:
          type: integer
          format: int64
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/ReportInput'
      required: true
    responses:
      "400":
        description: Invalid ID supplied
      "404":
        description: Question not found
      "405":
        description: Validation exception
```

```
    delete:
      tags:
        - reports
      summary: Deletes a report
      operationId: deleteReport
      parameters:
        - name: report_id
          in: path
          description: Report id to delete
          required: true
          style: simple
          explode: false
          schema:
            type: integer
            format: int64
      responses:
        "400":
          description: Invalid ID supplied
        "404":
          description: Report not found
  /products:
    get:
      tags:
        - products
      summary: Finds all products
      operationId: findProducts
      parameters:
        - name: product_name
          in: query
          description: Product name values that can be considered for
filter
          required: false
          style: form
          explode: true
          schema:
            type: string
        - name: product_type
          in: query
          description: Product type values that can be considered for
filter
          required: false
```

```
      style: form
      explode: true
      schema:
        type: string
        enum:
          - Websites
          - Mobile applications
          - Software
          - Electronic games
          - Physical products with electronic interfaces
          - Home appliances
          - GPS navigation systems
          - In-car entertainment systems
          - Vending machines
          - ATMs (Automated Teller Machines)
          - Access control systems
          - Lighting and temperature control systems in buildings
          - Medical devices
          - Consumer products
  responses:
    "200":
      description: successful operation
      content:
        application/json:
          schema:
            type: array
            items:
              $ref: '#/components/schemas/ProductOutput'
post:
  tags:
    - products
  summary: Add a new product
  operationId: createProduct
  requestBody:
    description: Product object that needs to be evaluated
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/ProductInput'
    required: true
  responses:
```

```
      "200":
        description: A ProductOutput object
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ProductOutput'
      "400":
        description: Bad Request
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Error'
            examples:
              ProductTypeInvalid:
                summary: productType invalid
                value:
                  message: productType invalid
/products/{product_id}:
  get:
    tags:
      - products
    summary: Find product by ID
    operationId: findProduct
    parameters:
      - name: product_id
        in: path
        description: ID of product to return
        required: true
        style: simple
        explode: false
        schema:
          type: integer
          format: int64
    responses:
      "200":
        description: successful operation
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ProductOutput'
          application/xml:
```

```
          schema:
            $ref: '#/components/schemas/ProductOutput'
    "404":
      description: Not found
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
          examples:
            CustomerNotFound:
              summary: Product not found
              value:
                message: Product not found
put:
  tags:
    - products
  summary: Update an existing product
  operationId: updateProduct
  parameters:
    - name: product_id
      in: path
      description: Product id to update
      required: true
      style: simple
      explode: false
      schema:
        type: integer
        format: int64
  requestBody:
    description: Product object that needs to be evaluated
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/ProductInput'
    required: true
  responses:
    "400":
      description: Bad Request
      content:
        application/json:
          schema:
```

```
            $ref: '#/components/schemas/Error'
          examples:
            ProductTypeInvalid:
              summary: productType invalid
              value:
                message: productType invalid
    "404":
      description: Not found
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
          examples:
            ProductNotFound:
              summary: Product not found
              value:
                message: Product not found
delete:
  tags:
    - products
  summary: Deletes a product
  operationId: deleteProduct
  parameters:
    - name: product_id
      in: path
      description: Product id to delete
      required: true
      style: simple
      explode: false
      schema:
        type: integer
        format: int64
  responses:
    "404":
      description: Not found
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
          examples:
            ProductNotFound:
```

```
                summary: Product not found
                value:
                  message: Product not found
  /customers:
    get:
      tags:
        - customers
      summary: Finds all customers
      operationId: findCustomers
      parameters:
        - name: customer_cpf_cnpj
          in: query
          description: CPF and CNPJ values that can be considered for
filter
          required: false
          style: form
          explode: true
          schema:
            type: string
      responses:
        "200":
          description: successful operation
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/CustomerOutput'
        "400":
          description: Bad Request
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
              examples:
                CpfCnpjInvalid:
                  summary: customer_cpf_cnpj invalid
                  value:
                    message: customer_cpf_cnpj invalid
        "404":
          description: Not found
```

```
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
              examples:
                CustomerNotFound:
                  summary: Customer not found
                  value:
                    message: Customer not found
    post:
      tags:
        - customers
      summary: Add a new customer
      operationId: createCustomer
      requestBody:
        description: Customer object that needs to have a product
evaluation
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/CustomerInput'
        required: true
      responses:
        "200":
          description: A CustomerOutput object
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/CustomerOutput'
        "400":
          description: Bad Request
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
              examples:
                CpfCnpjInvalid:
                  summary: cpfCnpj invalid
                  value:
                    message: cpfCnpj invalid
  /customers/{customer_cpf_cnpj}:
```

```
get:
  tags:
    - customers
  summary: Find a customer by ID
  operationId: findCustomer
  parameters:
    - name: customer_cpf_cnpj
      in: path
      description: CPF or CNPJ of customer to return
      required: true
      style: simple
      explode: false
      schema:
        type: integer
        format: int64
  responses:
    "200":
      description: successful operation
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/CustomerOutput'
        application/xml:
          schema:
            $ref: '#/components/schemas/CustomerOutput'
    "400":
      description: Bad Request
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
          examples:
            CpfCnpjInvalid:
              summary: customer_cpf_cnpj invalid
              value:
                message: customer_cpf_cnpj invalid
    "404":
      description: Not found
      content:
        application/json:
          schema:
```

```
                $ref: '#/components/schemas/Error'
              examples:
                CustomerNotFound:
                  summary: Customer not found
                  value:
                    message: Customer not found
    put:
      tags:
        - customers
      summary: Update an existing customer
      operationId: updateCustomer
      parameters:
        - name: customer_cpf_cnpj
          in: path
          description: CPF or CNPJ of customer to update
          required: true
          style: simple
          explode: false
          schema:
            type: string
      requestBody:
        description: Customer object that needs to have a product
evaluation
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/CustomerOutput'
          application/xml:
            schema:
              $ref: '#/components/schemas/CustomerOutput'
        required: true
      responses:
        "200":
          description: A CustomerOutput object
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/CustomerOutput'
        "400":
          description: Bad Request
          content:
```

```
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
          examples:
            CpfCnpjInvalid:
              summary: customer_cpf_cnpj invalid
              value:
                message: customer_cpf_cnpj invalid
    "404":
      description: Not found
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
          examples:
            CustomerNotFound:
              summary: Customer not found
              value:
                message: Customer not found
delete:
  tags:
    - customers
  summary: Deletes a customer
  operationId: deleteCustomer
  parameters:
    - name: customer_cpf_cnpj
      in: path
      description: CPF or CNPJ of customer to delete
      required: true
      style: simple
      explode: false
      schema:
        type: integer
        format: int64
  responses:
    "204":
      description: No content
    "404":
      description: Not found
      content:
        application/json:
```

```
              schema:
                $ref: '#/components/schemas/Error'
              examples:
                CustomerNotFound:
                  summary: Customer not found
                  value:
                    message: Customer not found
        "400":
          description: Bad Request
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
              examples:
                CpfCnpjInvalid:
                  summary: customer_cpf_cnpj invalid
                  value:
                    message: customer_cpf_cnpj invalid
components:
  schemas:
    ProductInput:
      type: object
      properties:
        productName:
          type: string
          example: Iphone X
        manufacterName:
          type: string
          example: Apple
        productType:
          $ref: '#/components/schemas/ProductType'
        productClass:
          type: string
          example: Class I
        completionLevel:
          type: string
          enum:
            - Alfa Test
            - Beta Test
            - Finished Product
            - Low Fidelity Prototype
```

```
        - Medium Fidelity Prototype
        - Specification
  description: Represents a product to be evaluated
ProductOutput:
  type: object
  properties:
    id:
      type: integer
      format: int64
    productName:
      type: string
      example: Iphone X
    manufacterName:
      type: string
      example: Apple
    productType:
      $ref: '#/components/schemas/ProductType'
    productClass:
      type: string
      example: Class I
    completionLevel:
      type: string
      example: Alfa Test
  description: Represents a product to be evaluated
ProductType:
  type: object
  properties:
    name:
      type: string
      description: Each product can be tested by usability testing
      enum:
        - Websites
        - Mobile applications
        - Software
        - Electronic games
        - Physical products with electronic interfaces
        - Home appliances
        - GPS navigation systems
        - In-car entertainment systems
        - Vending machines
        - ATMs (Automated Teller Machines)
```

```
            - Access control systems
            - Lighting and temperature control systems in buildings
            - Medical devices
            - Consumer products
      description: Representa o tipo do produto
    QuestionOutput:
      required:
        - description
      type: object
      properties:
        id:
          type: integer
          format: int64
        class:
          $ref: '#/components/schemas/Class'
        productType:
          type: string
          enum:
            - Software
            - Hardware
            - Hardware/Software
        baseQuestion:
          type: string
          example: Is it designed minimal?
        detailedQuestion:
          type: string
          example: Is only (and all) information, essential to decision
making, displayed on the screen?
      xml:
        name: Question
    Class:
      type: object
      properties:
        name:
          type: string
          description: Each class of a question is a Nilsen heuristic
          enum:
            - Aesthetic and minimalist design
            - Consistency and standards
            - Error prevention
            - Flexibility and efficiency of use
```

```
            - Help and documentation
            - Help users recognize, diagnose, and recover from errors
            - Match between system and the real world
            - Recognition rather than recall
            - User control and freedom
            - Visibility of system status
            - Privacy & Security
    QuestionInput:
      required:
        - description
      type: object
      properties:
        id:
          type: integer
          format: int64
        class:
          $ref: '#/components/schemas/Class'
        productType:
          type: string
          enum:
            - Software
            - Hardware
            - Hardware/Software
        baseQuestion:
          type: string
          example: Is it designed minimal?
        detailedQuestion:
          type: string
          example: Is only (and all) information, essential to decision
making, displayed on the screen?
      xml:
        name: Question
    Reference:
      type: object
      properties:
        type:
          type: string
          description: type of reference
          enum:
            - Article
            - Book
```

```
            - eDocument
            - Standard
        bibliography:
          type: string
    Answer:
      type: object
      properties:
        id:
          type: integer
        answer:
          type: string
        severity:
          type: string
          enum:
            - 0 Not a usability problem at all
            - 1 Cosmetic problem only. Need not be fixed unless extra time
is available
            - 2 Minor usability problem. Fixing this should be given low
priority
            - 3 Major usability problem. Important to fix. Should be given
high priority
            - 4 Usability catastrophe. Imperative to fix this before
product can be released
        weight:
          type: string
          enum:
            - 0 Necessary for every System
            - 1 Necessary for Systems without more than one Dialog
            - 2 Necessary for Simple Systems without data have to be
entered or with strict guidance throughout the usage process
            - 3 Necessary for Complex Systems with data have to be entered
on several dialogs
      description: Represents the answer data for the questions
    QuestionnaireOutput:
      properties:
        id:
          type: integer
          format: int64
        questions:
          type: array
          items:
```

```
            $ref: '#/components/schemas/QuestionOutput'
        evaluator:
          type: string
        creator:
          type: string
        title:
          type: string
      description: It represents a grouping of questions that must be
answered when analyzing a product.
    QuestionnaireInput:
      properties:
        questionIds:
          type: array
          items:
            type: string
        evaluator:
          type: string
        creator:
          type: string
        title:
          type: string
      description: It represents a grouping of questions that must be
answered when analyzing a product.
    RequestEvaluationOutput:
      properties:
        id:
          type: integer
          format: int64
        productId:
          type: string
        customerCpfCnpj:
          type: string
        startDate:
          type: string
          description: Entry date into lufh
          format: date-time
        endDate:
          type: string
          description: Planned departure date from lufh
          format: date-time
        urgency:
```

```
          type: boolean
        testType:
          type: string
          description: Type of contracted test
          enum:
            - compliance inspection
            - performance measurement
            - heuristic evaluation
            - user opinion
        coverage:
          type: string
          description: Contracted scope
      description: Represents the output of a request to evaluate a
product.
    RequestEvaluationInput:
      properties:
        startDate:
          type: string
          description: Entry date into lufh
          format: date-time
        endDate:
          type: string
          description: Planned departure date from lufh
          format: date-time
        urgency:
          type: boolean
        testType:
          type: string
          description: Type of contracted test
          enum:
            - compliance inspection
            - performance measurement
            - heuristic evaluation
            - user opinion
        coverage:
          type: string
          description: Contracted scope
      description: Represents the input of a request to evaluate a
product.
    CustomerInput:
      properties:
```

```
        customerName:
          type: string
        cpfCnpj:
          type: string
        email:
          type: string
        phoneNumber:
          type: string
      description: Represents a customer who has requested a product
review
    CustomerOutput:
      properties:
        id:
          type: integer
        customerName:
          type: string
        cpfCnpj:
          type: string
        email:
          type: string
        phoneNumber:
          type: string
      description: Represents a customer who has requested a product
review
    ReportInput:
      properties:
        requestEvaluationId:
          type: integer
        questionnaireId:
          type: integer
        answers:
          type: array
          items:
            $ref: '#/components/schemas/Answer'
      description: Represents the result of the evaluation
    ReportOutput:
      properties:
        requestEvaluationId:
          type: integer
        questionnaire:
          type: object
```

```
      $ref: '#/components/schemas/Questionnaire'
  description: Represents the result of the evaluation
Questionnaire:
  properties:
    id:
      type: integer
    answeredQuestions:
      type: array
      items:
        $ref: '#/components/schemas/AnsweredQuestion'
AnsweredQuestion:
  properties:
    question:
      type: object
      $ref: '#/components/schemas/QuestionOutput'
    answer:
      type: object
      $ref: '#/components/schemas/Answer'


Error:
  type: object
  properties:
    message:
      type: string
```