INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE ENGENHARIA ELETRÔNICA

MARIÉLI FERNANDES MATOS

DESENVOLVIMENTO DE UM *SOFTWARE* PARA AQUISIÇÃO DE DADOS DE UMA ESTAÇÃO DE RECARGA DE VEÍCULOS ELÉTRICOS UTILIZANDO A METODOLOGIA DE DESENVOLVIMENTO GUIADOS A TESTES

Florianópolis, 2021

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE ENGENHARIA ELETRÔNICA

MARIÉLI FERNANDES MATOS

DESENVOLVIMENTO DE UM *SOFTWARE* PARA AQUISIÇÃO DE DADOS DE UMA ESTAÇÃO DE RECARGA DE VEÍCULOS ELÉTRICOS UTILIZANDO A METODOLOGIA DE DESENVOLVIMENTO GUIADOS A TESTES

Trabalho de conclusão de curso submetido ao Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina como parte dos requisitos para obtenção do título de Engenheira Eletrônica

Orientador:
Prof. Me. Samir Bonho

Florianópolis, 2021

Ficha de identificação da obra elaborada pelo autor.

Matos, Mariéli
Desenvolvimento de um software para aquisição de dados
de uma estação de recarga de veículos elétricos utilizando
a metodologia de desenvolvimento guiados a testes
/ Mariéli Matos; orientação de Samir Bonho. -
Florianópolis, SC, 2021.
58 p.
Trabalho de Conclusão de Curso (TCC) - Instituto Federal
de Santa Catarina, Câmpus Florianópolis. Bacharelado
em Engenharia Eletrônica. Departamento Acadêmico
de Eletrônica.
Inclui Referências.

1. Desenvolvimento guiado a testes. 2. Testes automatizados.
3. Estações de recarga de veículos elétricos.
I. Bonho, Samir. II. Instituto Federal de Santa Catarina.
III. Desenvolvimento de um software para aquisição
de dados de uma estação de recarga de veículos
elétricos utilizando a metodologia de desenvolvimento guiados

DESENVOLVIMENTO DE UM *SOFTWARE* PARA AQUISIÇÃO DE DADOS DE UMA ESTAÇÃO DE RECARGA DE VEÍCULOS ELÉTRICOS UTILIZANDO A METODOLOGIA DE DESENVOLVIMENTO GUIADOS A TESTES

MARIÉLI FERNANDES MATOS

Florianópolis, 11 de outubro, 2021.

Banca Examinadora:

Samir Bonho, Me. Eng.

Hugo Marcondes, Me.

CCSantos

Cristiano Constantino Santos

RESUMO

Nos últimos anos, a busca por soluções para a diminuição dos gases que causam o efeito estufa tem sido um dos principais objetivos ao redor do mundo. Uma das alternativas encontradas é a adoção de veículos eletrificados. E, para viabilizar essa mudança, é preciso ofertar estações de carregamento para esses veículos com boa qualidade de serviço, por isso a obtenção dos dados provindos desse sistema de recarga é essencial. Dessa forma, este trabalho tem como objetivo oferecer um meio de acesso aos dados de uma estação de carregamento de veículos eletrificados, utilizando como princípio de desenvolvimento o TDD (desenvolvimento guiado a testes). Para isso, foram definidos os casos de uso do sistema, a fim de obter os requisitos, e, para desenvolver o *software* e os testes automatizados foi utilizada a linguagem de programação *Python*, por meio da metodologia descritiva com abordagem exploratória. Após a integração dos códigos gerados, são mostrados os resultados das requisições dos dados provenientes do eletroposto. Por fim, a criação de *software* utilizando as técnicas de TDD mostrou-se eficiente, além de adicionar confiança no processo de desenvolvimento.

Palavras-chave: Desenvolvimento guiado a testes. Testes automatizados. Estações de recarga de veículos elétricos.

ABSTRACT

In recent years, the search for solutions to reduce the emission of gases that cause the greenhouse effect has been one of the main goals around the world. One of the alternatives found is the adoption of electric vehicles. And, to make this change feasible, it is necessary to offer charging stations for these vehicles with good quality of service. Therefore obtaining data from this system is essential. Thus, this work aims to provide software for accessing data from an electrified vehicle charging station, using TDD (test-driven development) as a development principle. For that, the use cases of the system were defined in order to obtain the requirements. Python programming language was used to develop the software and the automated tests. As a result, integration tests were made and software requests were applied for getting the charging station data. Finally, the software development using TDD techniques proved to be efficient and it added confidence in the build-up process.

Keywords: Test-driven Development. Automated tests. Electric vehicle charging stations.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A última década foi marcada pelo alto crescimento da circulação de carros elétricos. No fim do ano de 2020 eram mais de dez milhões de veículos ao redor do mundo (IEA, 2021). De acordo com o BloombergNEF (2021), este cenário tende a continuar; até 2025, as vendas de carros elétricos devem representar, globalmente, 16% das vendas de veículos, e a previsão para 2040 é de aproximadamente 70%.

No Brasil, também existe um panorama de crescimento. Segundo a Associação Brasileira de Veículos Elétricos (ABVE, 2021) as vendas de carros eletrificados no ano de 2020 aumentaram 66% em relação ao ano anterior. Além disso, a projeção para o ano de 2030 é que os carros eletrificados cheguem a ocupar 1,9% da frota de veículos leves no país (BRASIL. EPE, 2020).

Uma das principais barreiras para a adoção de veículos elétricos (VE) por novos consumidores é a falta de infraestrutura para o carregamento das baterias (COSTA *et al.*, 2020). Uma das formas de melhorar o serviço é a aquisição dos dados dos eletropostos já existentes, pois segundo Almaghrebi *et al.* (2019), a partir dos dados obtidos, possibilita-se o estudo sobre disposição de novos postos de carregamento, como também a análise dos impactos da inserção de carregadores na rede de distribuição de energia (YUNUS; PARRA; REZA, 2011) e formas de melhorar a experiência do usuário, como por exemplo, diminuir o tempo de espera para o uso dos eletropostos (CAO *et al.*, 2017).

Dessa forma, a comunicação entre estações de carregamento e clientes pode ser estabelecida por meio de uma interface de programação de aplicações (API), pois, de acordo com a (REDHAT, 2021b) as APIs permitem a comunicação entre diferentes serviços sem a necessidade de saber como foram implementados. Porém, um dos principais problemas no desenvolvimento de *software* é garantir a qualidade do programa desenvolvido (PRIKLADNICKI, 2004), e uma das formas de aumentar a confiabilidade do código, segundo Bissia, Neto e Emer (2016) é a utilização das técnicas de desenvolvimento guiado a testes (TDD).

A fim de aumentar a confiabilidade e a qualidade do programa desenvolvido, este trabalho objetiva a produção de um *software* para o acesso a dados de uma plataforma de carregamento de carros elétricos, aplicando as técnicas de TDD.

Este trabalho estrutura-se desta forma: no capítulo 2, serão descritos os conceitos utilizados para a elaboração deste trabalho; em seguida, será apresentada a metodologia adotada; posteriormente, será mostrado o desenvolvimento deste *software*; e, por fim, serão apresentadas as considerações finais.

1.1 Justificativa

A queima de combustíveis fósseis é uma das principais causas do aumento da concentração de gás carbônico (CO_2) e óxido nitroso (NO_2) na atmosfera (ALMEIDA *et al.*, 2018), gases esses considerados os principais condutores da mudança climática mundial (RITCHIE; ROSER, 2020). Dentre as soluções adotadas para a redução da emissão de CO_2 à melhoria da qualidade do ar em centros urbanos está a adoção de veículos eletrificados para o transporte (HU *et al.*, 2021).

O Brasil ocupa o décimo quarto lugar na lista de maiores emissores de CO2 no mundo, segundo o Sistema de Estimativas de Emissões e Remoções de Gases de Efeito Estufa (2020). Em 2019, o país lançou na atmosfera aproximadamente 2,2 bilhões de toneladas de gás carbônico e, devido à queima de combustíveis fósseis para gerar a energia para os veículos, o setor de transporte foi responsável por emitir 196,5 milhões de toneladas. Apesar disso, o uso do VE está em estágios iniciais; segundo Sierzchula, Bakker e Wee (2014), uma das formas de estimular a adesão do VE é promover melhorias à infraestrutura de carregamento público, pois, com a aquisição dos dados da estação de carregamento de VE, é possível realizar o monitoramento com a análise de falhas (ZHANG *et al.*, 2021) e gerenciamento com diferentes métodos de pagamento (LIU, 2017). Outra vantagem em obter os dados, é realizar pesquisas para melhorar a infraestrutura já existente com base na utilização desses eletropostos (HOED *et al.*, 2013).

Assim, este trabalho se justifica por contribuir para a oferta de meios de acesso aos dados, seja para o gerenciamento das operações ou para a melhorar a infraestrutura das estações de carregamento.

1.2 Definição do problema

De acordo com Lee e Park (2016), carregadores das baterias de VE produzidos atualmente já possuem controle e gerenciamento digital; então, com os dados obtidos, é possível prover diversos serviços como, por exemplo, a coordenação dos usuários a pontos de recarga desocupados; as diferentes políticas de preço; o gerenciamento de agendamento inteligente; e a criação de uma gestão de sócios.

Dessa forma, este trabalho procurará responder ao seguinte questionamento: como desenvolver um *software* para fornecer os dados de uma estação de carregamento, utilizando os métodos de desenvolvimento guiados por testes, de forma a disponibilizar os dados a usuários externos?

1.3 Objetivo geral

Este trabalho tem como objetivo desenvolver um programa para o acesso aos dados de uma plataforma de recarga de carros elétricos, utilizando como metodologia o desenvolvimento guiado por testes.

1.4 Objetivos específicos

Para o desenvolvimento deste trabalho estabeleceu-se os seguintes objetivos específicos:

a) apresentar os conceitos do desenvolvimento guiado a testes;

b) definir os requisitos do projeto utilizando casos de uso;

c) descrever o desenvolvimento guiado a testes;

d) testar as requisições ao software para acesso aos dados da estação de recarga.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo começa com as definições de TDD e os tipos de testes que podem ser realizados com essa prática; em seguida, serão apresentados os conceitos de *web services* e os métodos de comunicação SOAP e REST; serão definidos, também, os conceitos de API e as comparações, ao utilizar os padrões de REST e SOAP; por fim, será apresentado o conceito de casos de uso.

2.1 Desenvolvimento guiado por testes

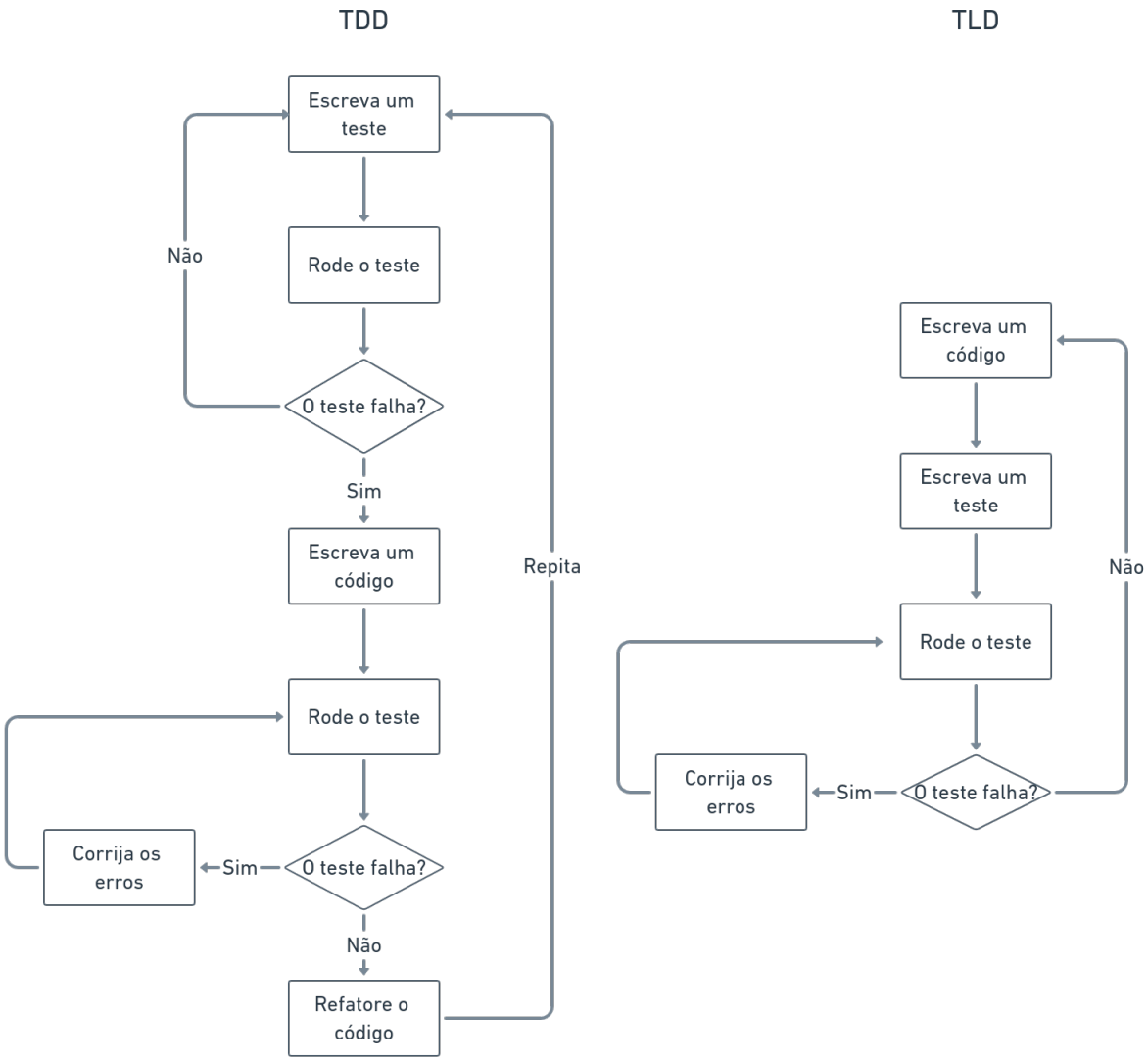
O desenvolvimento guiado por testes é uma técnica de criação de *software* que utiliza em conjunto dois métodos de desenvolvimento: *test-first* e refatoração (AMBLER, 2013).

Para Beck (2001), test-first é uma técnica de análise e design lógico do software que permite, além da definição do que está no escopo do problema, a criação de um código menos acoplado e mais coeso. Nesse contexto, a refatoração é utilizada para eliminar as duplicações criadas no código e deixá-lo limpo (BECK, 2003).

O TDD é uma técnica antiga de desenvolvimento de software que remonta aos anos 60; nesse período, o procedimento era totalmente manual. No entanto, essa ferramenta de desenvolvimento foi redescoberta nos anos 90, impulsionado pelo manifesto ágil e o *extreme programming*, tendo a automação dos testes como grande diferencial da metodologia utilizada no passado (BARBER, 2012).

O desenvolvimento guiado por testes diferencia-se das técnicas de criação de *software* tradicionais pelo fluxo de desenvolvimento (Figura 1). Independente da metodologia de desenvolvimento, as técnicas tradicionais sempre têm a mesma ordem de construção: a primeira etapa é o design do programa; em seguida, o código é criado; e, por último, a aplicação é testada - essa técnica também é conhecida como *test last*. Em contrapartida, na abordagem do TDD, a primeira etapa é o teste; em seguida, é desenvolvido o código; e, por fim, a refatoração (KUMAR; BANSAL, 2013).

Figura 1 – Fluxo de desenvolvimento tradicional (TLD) vs TDD



Fonte: Adaptado de Munir, Petersen e Moayyed (2014).

Ao desenvolver aplicações utilizando a abordagem clássica, criam-se alguns problemas. Quando os testes são realizados apenas no final do desenvolvimento, torna-se difícil para o programador relembrar os detalhes do programa. Caso o responsável pela implementação dos testes for diferente do desenvolvedor do código, existe uma possibilidade de uma parte importante do programa não ser testada, pois o desenvolvedor não conhece profundamente a aplicação. Além disso, quando se faz necessário corrigir um defeito em um código já existente, pode-se criar uma falha em outro lugar, e a infraestrutura de testes existentes pode ou não encontrar esse novo erro (ASTELS, 2003).

O TDD resolve esses problemas, pois como na primeira etapa são criados os testes, não existe o esquecimento dos detalhes da implementação e isso garante o

domínio do desenvolvedor em relação ao código. Outro problema resolvido ao utilizar TDD em relação à abordagem tradicional, refere-se à garantia de funcionamento do sistema ao corrigir um defeito no código, já que ao adicionar uma nova falha à aplicação, um teste falha imediatamente (ASTELS, 2003).

Para Beck (2003), o TDD possui duas leis: a) escrever um novo código apenas se existir um teste automatizado que esteja falhando e; b) eliminar as duplicações. Para isso, essa técnica possui um ciclo de desenvolvimento que se baseia em escrever um pequeno teste falho; em seguida, escrever um código de maneira a fazer o teste passar da forma mais rápida possível e, por último, refatorar o código para eliminar as duplicações criadas. Esse ciclo também é conhecido como *red*, *green* e *refactor* Figura 2.

Figura 2 – Ciclo de desenvolvimento do TDD

Fonte: Adaptado de Soares (2020).

De acordo com Beck (2003), existem três formas de transformar a etapa *red* para a etapa *green*: seguir a estratégia de *faking*; a de implementação óbvia e; a de triangulação.

A tática de *faking* é a maneira mais fácil de fazer um teste ser aprovado. Resume-se em retornar o valor que o teste está esperando e, gradualmente, transformar os valores constantes em expressões reais (TARLINDER, 2016).

Segundo Beck (2003), a próxima técnica é a de implementação óbvia, que utiliza a maneira mais intuitiva de resolver o problema; porém, os passos do ciclo de TDD nessa abordagem costumam ser maiores, podendo acarretar erros inesperados.

A triangulação permite extrair algoritmos por meio de exemplos, e então generalizá-los. É possível um teste retornar um falso positivo, mas quando inseridos outros exemplos com parâmetros diferentes e o teste falhar, pode-se refatorar o código para chegar a um caso de generalização (TARLINDER, 2016).

2.1.1 *Tipos de testes*

Uma maneira de classificar os tipos de testes é em relação ao tamanho de seus escopos. Testes que possuem o escopo pequeno, também chamados de testes de unidade, são responsáveis por validar a lógica em uma parte pequena da base de código. Já os testes com escopo de tamanho médio, conhecidos como testes de integração, são criados para verificar as interações entre um número pequeno de componentes. Por fim, os testes com escopo grande, conhecidos como testes funcionais ou testes de ponta a ponta, são modelados para validar a interação entre partes distintas do sistema, ou um comportamento que não é expressado por uma única classe (WINTERS; MANSHRECK; WRIGHT, 2020).

2.1.1.1 Testes de unidade

Segundo Osherove (2014), os testes de unidade são um código automatizado que invoca um único caso de uso lógico e funcional e verifica se um único comportamento age como o esperado. Essa categoria de teste costuma ter tamanho e escopo pequeno, além de ser rápida e determinística, o que permite um *feedback* imediato. Uma razão de seu uso é a facilidade e a rapidez de escrevê-lo, o que aumenta a cobertura de testes do sistema e, consequentemente, facilita mudanças futuras no software (WINTERS; MANSHRECK; WRIGHT, 2020).

Outro motivo da utilização desses tipos de testes é a ajuda em manter a qualidade interna do programa, pois os pedaços de código precisam ser pouco acoplados e altamente coesos para serem facilmente testados. Mas, apesar de oferecer esse *feedback* sobre a qualidade do código, essa classe de teste não oferece a segurança necessária para garantir que o sistema funcione corretamente (FREEMAN; PRYCE, 2009).

Para Reese (2018), as características que bons testes de unidade devem possuir são: a rapidez, pois em projetos grandes é comum haver muitos testes unitários, então, esses testes devem ser executados de forma rápida; o isolamento, pois devem ser executados sem fatores externos, repetíveis, ou seja, os testes devem sempre retornar aos mesmos resultados com a mesma entrada; e, por fim, automatizados, para isso, os testes devem detectar automaticamente se o seu resultado foi aprovado ou não.

2.1.1.2 Testes de integração

Aplicações geralmente conectam-se com sistemas externos; porém, os testes unitários, devido ao seu intencional isolamento, não verificam o comportamento da aplicação com serviços externos; então para preencher essa lacuna, utilizam-se os testes de integração (VOCKE, 2018).

Como os testes de integração são os responsáveis por testar a infraestrutura da aplicação, eles podem incluir alguns componentes externos ao código, e isso, geralmente, inclui o banco de dados, o sistema de arquivos e os dispositivos de rede. É importante salientar que, ao contrário dos testes unitários, essa classe de testes utiliza os componentes reais que o software usa em produção e, por isso, exige mais processamento de dados e leva mais tempo para ser executado (NELSON; SMITHE; TIL, 2021).

2.1.1.3 Testes de ponta a ponta

Os testes de ponta a ponta são um método de testes que tem como objetivo verificar o funcionamento do fluxo da aplicação do começo ao fim. Para isso, replica os cenários reais do usuário (BOSE, 2021).

Com esse método, verifica-se o comportamento de todas as partes da aplicação que estão conectadas; por isso, pode-se achar falhas na interação entre componentes, o que os outros testes não conseguem. No entanto, testes de ponta a ponta podem ser mais difíceis de se manter e mais lentos ao rodar, se comparados aos outros tipos de testes (FOWLER, 2013).

2.1.2 *Vantagens e desvantagens na utilização do TDD*

Utilizar o TDD traz benefícios ao desenvolvimento de *software*. Segundo Bhat e Nagappan (2006), essa prática aumenta a qualidade interna e externa do *software*. Outra vantagem ao utilizar essa técnica é apontada por Hilton (2009) que, em seu estudo, mostra que a complexidade do código diminuiu ao utilizar essa conduta de desenvolvimento. De acordo com Khanam e Ahsan (2017), tal técnica aumenta a confiança do desenvolvedor em relação ao funcionamento de seu código, além disso, existe o aumento da compreensibilidade de leitura do *software*, o que facilita a sua manutenção.

No entanto, o desenvolvimento guiado por testes pode trazer alguns malefícios. Segundo Bhat e Nagappan (2006), a produtividade do desenvolvedor diminui em relação às técnicas tradicionais. Caso as especificações do projeto não estejam bem definidas e concretas, pode-se ter retrabalho. Além disso, alguns sistemas podem levar muito tempo para ser generalizados devido a sua complexidade (KHANAM; AHSAN, 2017).

2.1.3 Testes automatizados

Os testes automatizados são aplicações que possuem um conjunto de ferramentas para automatizar o processo de validação e revisão de um *software*. Esses processos surgiram para substituir o processamento manual, devido a lentidão, ao

preço e a propensão a erros ao comparado com o processo automatizado (REHKOPF, 2021).

De acordo com Molina (2021), a automação dos testes possibilitam a escalabilidade e rapidez, o que garante a robustez da aplicação desenvolvida. Outro benefício é a agilidade na entrega de novas funcionalidades, pois com a metodologia anterior grande parte do tempo de desenvolvimento era disponibilizado para os testes manuais.

2.1.3.1 Pytest

O Pytest é um *framework* baseado na linguagem de programação Python que oferece um ferramental completo para a criação de testes automatizados e pode ser usado em aplicações de todos os tamanhos. Devido a facilidade de escrita dos testes e as linhas de comando que aumentam a produtividade do desenvolvedor, o Pytest uma das ferramentas mais utilizadas no desenvolvimento de testes em plaicações que utilizam Python como linguagem de programação (OLIVEIRA, 2018).

Esse *framework* é capaz de detectar automaticamente os testes, pois convenciona os arquivos que possuem os nomes de *test_*.py* ou **_test.py* como arquivos de testes. As verificações são feitas por meio de afirmações (*assert*), e verifica se o resultado esperado é o recebido pelo teste (HUNT, 2019).

Segundo Oliveira (2018), com essa ferramenta, é possível fazer customizações por meio de *plugins*, e dessa forma mudar diferentes aspectos nos testes e em como eles rodam. Testes de aplicações maiores podem ter suas configurações realizadas por meio de um mecanismo chamado *fixtures*, que permite adicionar um comportamento antes ou depois de um ou mais teste.

2.2 *Web services*

De acordo com Paik *et al.* (2017), *web services* são um serviço disponível na internet, que não possui vínculo a nenhuma linguagem de programação, arquitetura ou sistema operacional. Esse serviço pode ser acessado remotamente por diversos dispositivos e utiliza a padronização de mensagem XML. As formas mais comuns de implementá-los são utilizando SOAP e REST.

Esses serviços geralmente são compostos por três partes: o provedor do serviço, o registro do serviço e o solicitante do serviço. A interação entre eles ocorre por meio das funções de publicar, procurar e conectar (Figura 3). Tais serviços são descobertos e conectados pelo solicitante; o *web services* é implementado e publicado pelo provedor; e as informações são mantidas no registro (Z.SHENG *et al.*, 2014).

Figura 3 – Arquitetura de *web services*

Fonte: Adaptado de Walker (2021).

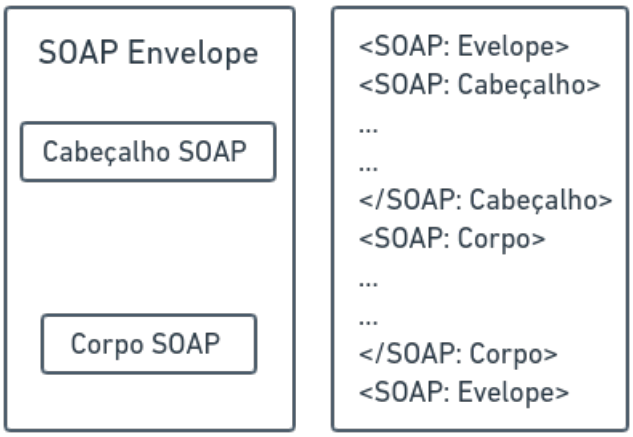
2.2.1 Protocolo Simples de Acesso a Objetos (SOAP)

SOAP é um protocolo de comunicação utilizado por *web services* para a transmissão de mensagens entre aplicações. Esse protocolo permite aos elementos da arquitetura comunicar-se com uma mensagem clara e padronizada. Essas mensagens baseiam-se no padrão XML que descreve como as mensagens de metadata e *payload* devem ser empacotadas dentro do documento ((PAIK *et al.*, 2017).

Os *web services* que utilizam o SOAP possuem *status* e exigem mais recursos de computação, especialmente ao lidar com as mensagens recebidas, e esse serviço geralmente é usado para integrar aplicativos corporativos complexos (Z.SHENG *et al.*, 2014).

As mensagens que utilizam esse protocolo consistem em um envelope que contém um cabeçalho opcional e um corpo obrigatório. O cabeçalho contém um bloco que possui informações relevantes de como a mensagem deve ser processada, incluindo as configurações de rota e entrega, de autorização ou autenticação e contexto de transações (Figura 4). O corpo possui a mensagem para ser entregue e processada; nesse campo, é permitido utilizar qualquer informação desde que seja processada na sintaxe XML (TIDWELL; SNELL; KULCHENKO, 2001)).

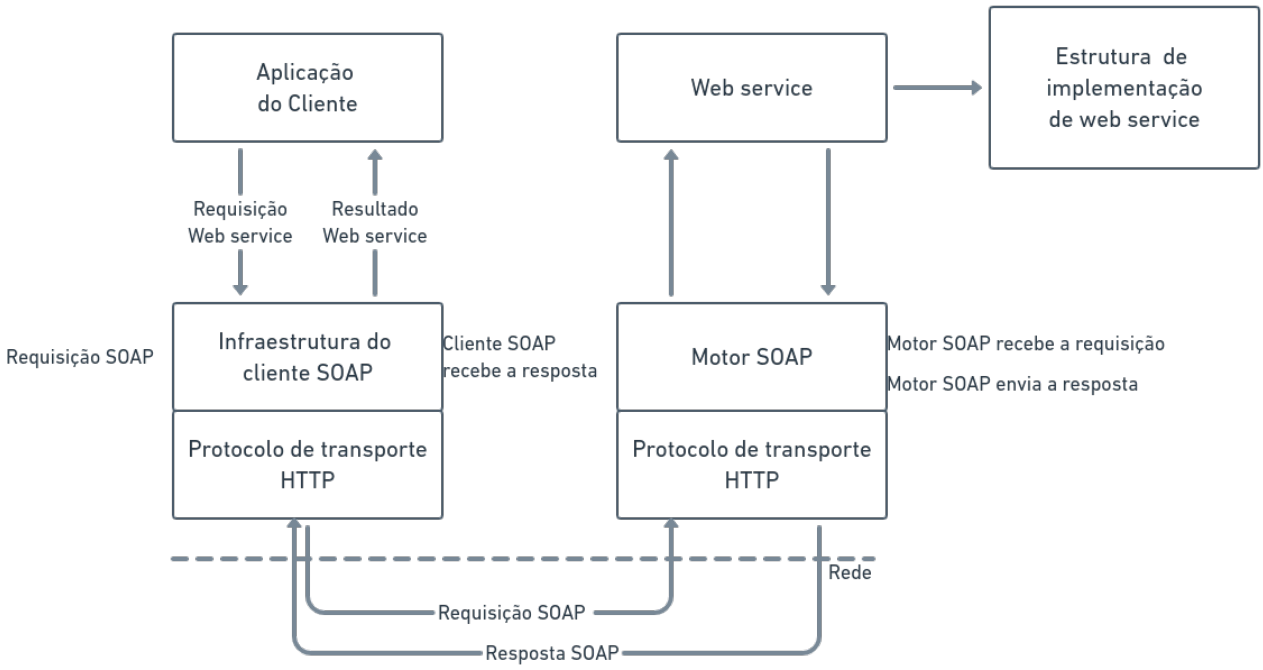
Figura 4 – Mensagem do protocolo SOAP



Fonte: Adaptado de Aroraa e Dash (2018).

Ao comunicar-se com um *web services* por meio do protocolo SOAP, utilizam-se os serviços de requisição e resposta. No primeiro momento, o cliente constrói uma mensagem SOAP (requisição) e transmite pela rede via HTTP. No lado do servidor, o software responsável por monitorar os recebimentos de mensagens SOAP, aceita a mensagem e envia ao destinatário. O serviço é executado pela requisição e, então, a resposta é gerada e construída como uma mensagem SOAP e é transmitida pelo protocolo HTTP de volta para o cliente. A Figura 5 mostra esse fluxo de interação (PAIK *et al.*, 2017).

Figura 5 – Fluxo de comunicação do protocolo SOAP



Fonte: Adaptado de Paik *et al.* (2017).

2.2.2 Transferência de Estado Representacional (REST)

Segundo Paik *et al.* (2017), a arquitetura REST foi apresentada por Roy Fielding em sua dissertação de Pós-Doutorado, com a proposta de um novo método para a criação de aplicações distribuídas. Enquanto o protocolo SOAP tem o enfoque em padronizar as mensagens, o REST priorizou em estudar a lógica por trás da arquitetura da *web*.

De acordo com Webber, Parastatidis e Robinson (2010), a arquitetura da web é o conjunto de milhares de interações simples e de uma pequena escala entre entre agentes e recursos que usam as tecnologias HTTP (Protocolo de Transferência de Hipertexto) e URI (identificador uniforme de recurso). Os recursos são qualquer coisa que está exposta na web como, por exemplo, um documento. Cada recurso possui um identificador único; para isso, utiliza-se uma URI que, além de identificar um recurso, faz com que ele se torne acessível e manipulável com a utilização do protocolo HTTP. Para um recurso enviar o seu status para o cliente, utiliza-se um recurso chamado representação; com isso, é possível enviar um dado com o estado atual (PAIK *et al.*, 2017).

Segundo Massé (2012), a arquitetura REST utiliza seis restrições consideradas chave para a criação de aplicativos distribuídos, que se chamam estilo arquitetural

da *web*. Os princípios são: cliente-servidor, interface uniforme, sistema por camada, *cache*, *stateless* e código sob demanda.

O cliente-servidor diz respeito à independência de implementação e implantação entre as partes; dessa forma, simplifica os componentes do servidor e melhora a escalabilidade. O sistema por camadas permite que um serviço intermediário, como um *proxy*, possa ser implementado de forma transparente entre um cliente e um servidor. O *cache* refere-se ao envio de uma autorização da parte do cliente sobre o armazenamento de seus dados pelo servidor. O *stateless* relaciona-se com o fato de que o servidor não precisa memorizar o estado da aplicação do cliente, que tem a responsabilidade de enviar todas as informações necessárias a cada interação com o servidor. O código sob demanda, único princípio opcional, permite que o *web services* transmita ao cliente por tempo limitado códigos executáveis (MASSÉ, 2012).

A última restrição refere-se à interface uniforme. Essa restrição fornece a arquitetura, o desacoplamento entre o cliente e o serviço REST implementado. Para isso, foram definidos padrões que todo serviço REST suporta. Para implementar esses padrões, estabeleceram-se quatro regras que as interfaces devem seguir:

a) identificações de recursos - indicam que todo o recurso deve possuir uma URI para identificá-lo;

b) manipulações de recursos por meio de representações - quando um cliente fornece um recurso, deve conter todas as informações necessárias para acessar o recurso;

c) mensagens auto-descritivas - as mensagens passadas devem conter todas as informações sobre o dado para ser entendido e processado;

d) HETEOAS (hipermídia como o mecanismo do estado do aplicativo) - a representação retornada pelo serviço deve conter todas as ações futuras por meio de um link (ARORAA; DASH, 2018).

Para integrar os itens da interface uniforme, são utilizados os métodos HTTP, que são um conjunto de ações com semântica bem definida, suficiente para atender grande parte das aplicações. Atualmente, utilizam-se os métodos *get*, *post*, *delete*, *put*, *options*, *head*, *trace*, *connect* e *patch*, (WEBBER; PARASTATIDIS; ROBINSON, 2010). A Tabela 1 mostra um resumo das funções para cada método.

De acordo com Aroraa e Dash (2018), a resposta fornecida pelo servidor para as solicitações desses métodos é feita por meio de *status codes*. A especificação do protocolo HTTP possui um guia com diversos *status code*; porém, uma aplicação mínima deve conter os códigos 200, que representa uma resposta sem erro; o 400, em que ocorre um erro na requisição do cliente, e o 500, em que o ocorre um erro interno durante o processo da requisição (PAIK *et al.*, 2017). A Figura 6 mostra um exemplo

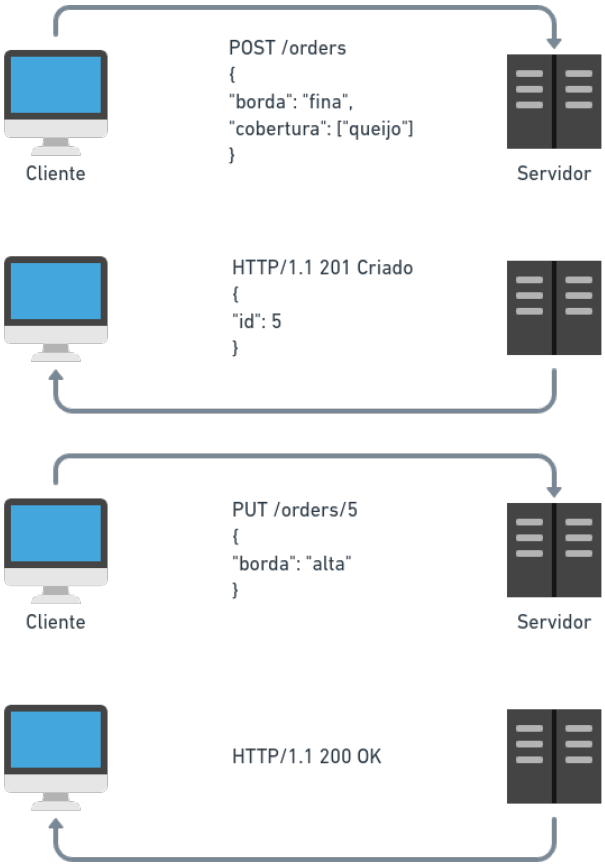
Tabela 1 – Funções dos métodos HTTP

Método	Função
get	Solicita a representação de um dado.
post	Submete uma entidade a um recurso específico
delete	Remove um recurso.
put	Descreve as opções de comunicações com o recurso de destino do recurso de destino pelos dados da requisição.
options	Descreve as opções de comunicações com o recurso de destino.
head	Solicita uma representação dos dados, porém sem conter o *body*.
trace	Envia uma mensagem por todo o caminho até o destino.
connect	Estabelece uma comunicação de duas vias com o recurso solicitado.
patch	Aplica modificações parciais em um recurso.

Fonte: Mozilla (2021).

de interação entre o cliente e o servidor, em que o cliente cria um novo item na rota *orders*, e recebe uma resposta de confirmação com um identificador e, em seguida, o cliente envia uma retificação da informação para a mesma rota com o identificador e em seguida recebe a confirmação do recebimento.

Figura 6 – Fluxo de comunicação do protocolo HTTP



Fonte: Adaptado de Cooksey (2014).

2.3 Interface de programação de aplicações

APIs são um conjunto de regras que o cliente e o provedor do serviço seguem (COOKSEY, 2014) e são usadas para construir *software* de sistemas distribuídos, aos quais os componentes são pouco acoplados. De acordo com Biehl (2015), as *web* APIs entregam dados de recursos por meio da *web*, já as aplicações que utilizam APIs são aplicativos móveis, aplicações web e aplicativos em nuvem.

O benefício de usar esse tipo de abordagem é a simplicidade, a clareza e a acessibilidade. As APIs provêm uma interface reusável, o que difere de outras abordagens. No entanto, APIs não oferecem uma interface para o usuário final, a ideia é operar por trás dos panos e ser apenas chamadas por outros aplicativos, ou seja, APIs são usadas para comunicação entre máquinas e para a integração entre dois ou mais softwares (BIEHL, 2015).

De acordo com Jacobson, Brail e Woods (2012), as APIs são classificadas pela formalidade de negócios; elas podem ser privadas ou públicas. Uma API é chamada de pública quando ela está disponível para acesso de qualquer usuário com

nenhum ou um pequeno contrato com o provedor do serviço. Uma API é considerada privada quando existe um contrato entre o consumidor e o provedor da API.

O estilo de arquitetura é uma solução de estrutura em larga escala pré-definida, que agiliza o processo de criação do serviço. Existem diversos estilos de arquitetura para implementar APIs, dentre eles os estilos REST e SOAP (BIEHL, 2015).

De acordo com Biehl (2015), o estilo REST é uma arquitetura para serviços que define um conjunto de restrições e acordos com cliente e provedor do serviço; além disso, essa arquitetura faz o uso otimizado da infraestrutura do protocolo HTTP.

As APIs baseadas em REST possuem quatro níveis de maturidade, considerando as aplicações das tecnologias de URI, HTTP e *Hypermedia* (RICHARDSON; RUBY, 2007).

O nível zero refere-se aos serviços que utilizam apenas uma URI e um método HTTP (geralmente *post*). O nível um diz respeito à implementação de diversas URIs e apenas um método HTTP. No nível dois, as APIs possuem diversas URIs e métodos HTTP (geralmente o *create*, *read*, *update* e *delete*). O último nível refere-se à implementação do HATEOAS, ou seja, as representações contêm um link para outros recursos que podem interessar o cliente (WEBBER; PARASTATIDIS; ROBINSON, 2010).

O estilo SOAP é um protocolo para *web services* padronizado pela W3C, que expõe procedimentos como conceito central. Esse protocolo permite o uso de diversos protocolos de programação e utiliza XML como padronização de suas mensagens (BIEHL, 2015). A Tabela 2 mostra um comparativo entre as APIs REST e SOAP.

Tabela 2 – Comparação entre APIs REST e SOAP

SOAP	REST
É um protocolo baseado em mensagem XML	É um estilo arquitetônico
A comunicação usada entre cliente e servidor é o WSDL.	XML ou JSON podem ser usados para a comunicação entre cliente o servidor.
Serviços são invocados pela chamada utilizando o método RCP.	São usados URI para expor os serviços.
As respostas são facilmente lidas por um humano.	As respostas são facilmente lidas na forma de XML ou JSON.
A transferência de dados ocorre por meio de diversos protocolos de comunicação.	Utiliza apenas o protocolo HTTP.

Fonte: Aroraa e Dash (2018).

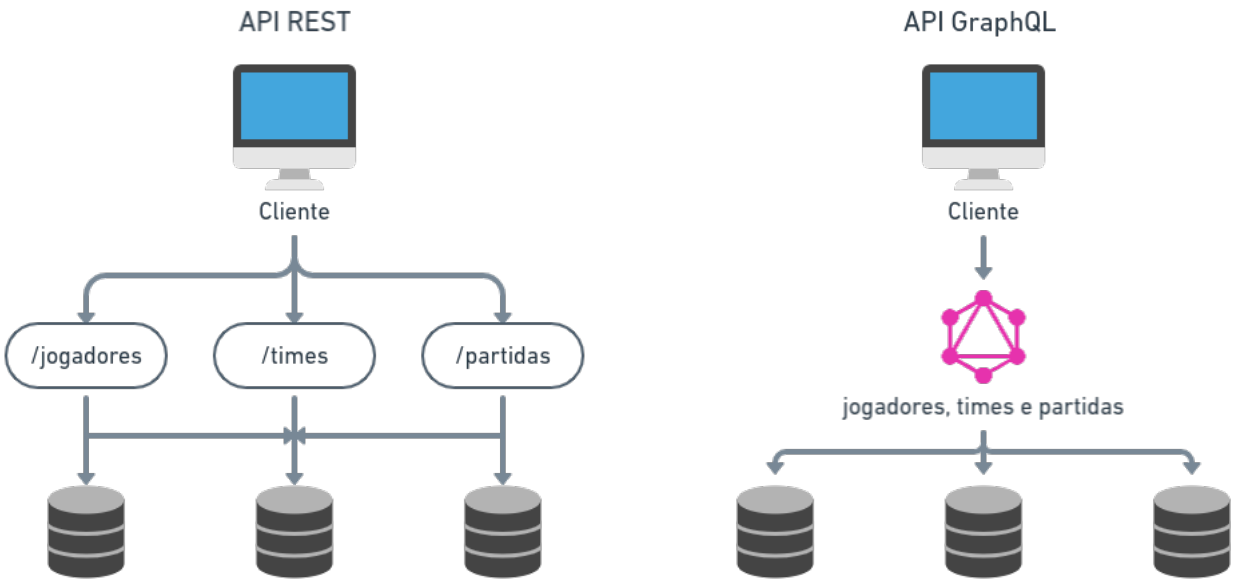
2.3.1 GraphQL

Graphql é uma linguagem de consulta utilizada em APIs, que permite ao cliente o requisitar apenas os dados que deseja, mesmo que estejam em fontes diferentes (GRAPHQL, 2021).

Em geral, as API que utilizam essa tecnologia baseiam-se no protocolo HTTP para a comunicação entre cliente e servidor, porém apenas os métodos GET e POST são suportados. Os dois métodos são utilizados para obter informações armazenadas no servidor, quando utilizado o método GET a requisição do dado está na URL enquanto no método POST a requisição é enviada em seu corpo (PORCELLO; BANKS, 2018).

De acordo com Wieruch (2019), o problema que essa linguagem resolve em comparação a arquitetura REST é em relação ao excesso de dados não utilizados que são recebidos pelo cliente ao requisitar uma informação. GraphQL atua no lado do servidor e do cliente que permite ao usuário decidir qual informação é necessária ao realizar uma requisição ao servidor. Como é mostrado na Figura 7, onde na linguagem GraphQL é possível o cliente acessar informações em diferentes banco de dados com apenas uma requisição, enquanto na arquitetura REST é preciso realizar requisições diferentes para acessar dados em diferentes base de dados.

Figura 7 – Comparação entre API GraphQL e REST



Fonte: Adaptado de Everis (2019).

O cliente possui três opções de operações quando utilizado GraphQL, as mutações (*mutations*), consultas (*querys*) e subscrições (*subscriptions*). As consultas são usadas para obter dados armazenados no servidor, as mutações são utilizadas para alterar os dados do lado do servidor e as subscrições que permitem ao cliente obter as alterações das informações em tempo real.

Essa linguagem utiliza esquemas, consultas e resolvedores para manipular os dados. Os esquemas descrevem todos os dados disponíveis para consulta pelos clientes, que quando recebidas validam e executam as consultas. A API anexa cada campo do esquema em um resolvedor (REDHAT, 2021a), que é uma função que preenche os dados em um único campo do esquema (APOLLO, 2019).

2.4 Linguagem de modelagem unificada (UML)

De acordo com Rumbaugh, Jacobson e Booch (1999), a UML (linguagem de modelagem unificada) é uma linguagem utilizada para a modelagem visual, a fim de especificar, documentar, visualizar e construir as unidades de um *software*.

Essa linguagem, criada em 1997, é controlada pela *Object Management Group* (OMG) e trata-se de um conjunto de diversas linguagens de modelagem gráfica orientada a objetos que ganharam notoriedade no final dos anos 80 e começo dos anos 90 (FOWLER, 2003).

O principal objetivo dessa linguagem é oferecer uma ferramenta que permita a análise, o projeto e a implementação de sistemas baseados em *software*, ou a modelagem de negócios e processos semelhantes (OMG, 2015).

Um modelo no UML é representado por diagramas, que apresentam uma visualização da descrição do modelo. Os diagramas de UML podem ser subdivididos em dois grupos, o diagrama estrutural e o diagrama comportamental. O diagrama de estrutura mostra os objetos no sistema, e o diagrama de comportamento mostra a interação no sistema. Existem ao todo 15 diagramas diferentes, mas apenas cinco são os mais utilizados: o diagrama de classe, o diagrama de casos de uso, o diagrama de máquina de estado, o diagrama de sequência e o diagrama de atividades (SEIDL *et al.*, 2014).

2.4.1 Caso de uso

De acordo com Cockburn (2000), um caso de uso é um acordo entre as partes interessadas sobre o funcionamento de um sistema. Os casos de uso incluem todos os comportamentos que existem em um sistema, o comportamento principal, as variações e as exceções que podem ocorrer em comparação com a linha principal, sem explicitar a estrutura interna do sistema (RUMBAUGH; JACOBSON; BOOCH, 1999).

Casos de uso possuem um ator primário, responsável por acionar o sistema para entregar um serviço - é esse ator que possui um objetivo do qual os casos de usos buscam satisfazer; mas também podem ter outros atores com quem o sistema pode se comunicar para o caso de uso ser completado (FOWLER, 2003).

De acordo com Rumbaugh, Jacobson e Booch (1999), cada caso de uso possui relações com outros casos de uso e atores, os quais podem expressar uma extensão, que amplia o caso de uso principal; uma inclusão, que insere um comportamento adicional ao caso de uso; e uma generalização, que relaciona um caso de uso geral para uma especificação desse caso de uso.

Segundo Fowler (2003), utiliza-se o formato em texto para prover uma descrição dos sistemas de forma clara e concisa, como mostra a Figura 8. Nesse formato, explicita-se o cenário principal de sucesso e cenários diferentes como extensões. Os cenários alternativos podem ser casos de sucesso ou de falhas; porém, um problema nesse tipo de caso de uso é a falta de especificações das relações de inclusões, pois não existe uma padronização. Uma alternativa é utilizar um sublinhado nos casos que possuem esse tipo de relacionamento.

Figura 8 – Descrição de um caso de uso

Compara um produto

Nível do objetivo: nível do mar

Cenário principal de sucesso:

1. O cliente navega pelo catálogo e seleciona o item para compra
2. O cliente vai até o checkout.
3. O cliente preenche as informações de envio (endereço, entrega para o próximo dia ou em 3 dias).
4. O sistema apresenta a informação de preço completo, incluindo o frete.
5. O sistema preenche as informações do cartão de crédito.
6. O sistema confirma o pagamento.
7. O sistema autoriza a compra imediatamente.
8. O sistema manda um email de confirmação para o cliente.

Extensões:

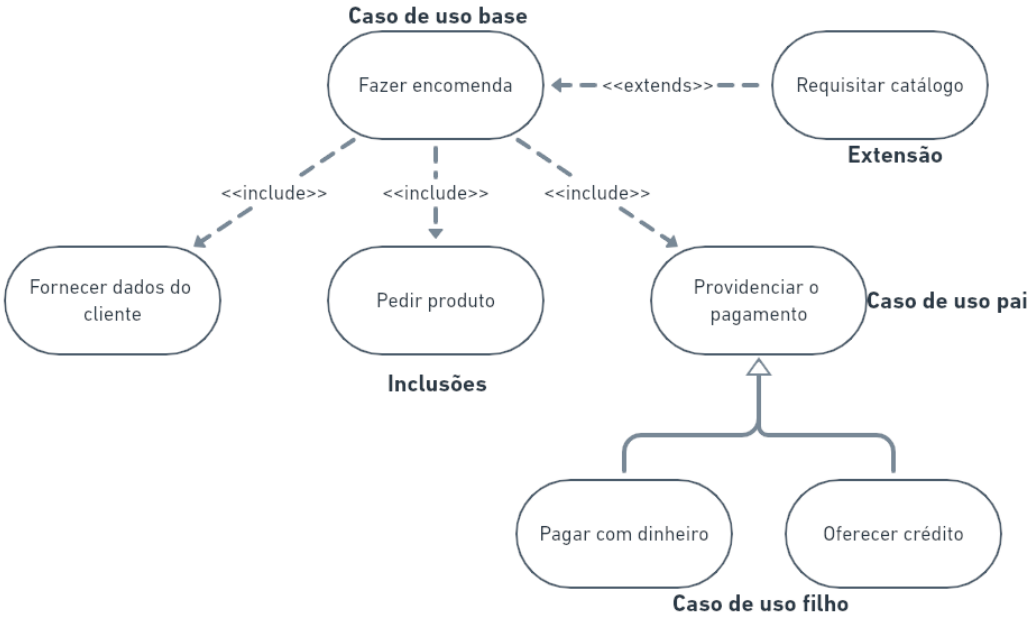
3a: O cliente é regular:
.1: O sistema mostra informações atuais de envio, preço, faturamento.
.2: O cliente pode aceitar ou substituir esses padrões, retorna a etapa 6.

6a: O sistema falha em autorizar o pagamento.
.1: O cliente pode inserir novamente as informações do cartão ou pode cancelar.

Fonte: Adaptado de Fowler (2003).

Os diagramas de caso de uso são utilizados para oferecer uma representação visual de alto nível dos requisitos de usuário (WIEGERS; BEATTY, 2013). Nesse tipo de representação, os casos de uso são ilustrados por elipses, atores, bonecos e retângulos. Os casos de extensões, inclusões e generalizações são representeados por elipses, e são diferenciados pelo tipo de ligação ao caso de uso principal. As extensões são ligados ao caso de uso principal por flechas pontilhadas, com orientação da extensão para o caso de uso principal. As inclusões também são ligados ao caso de uso principal por flechas pontilhadas, mas com orientação do caso de uso principal para a inclusão. E as generalizações são ligados ao caso de uso pai por flechas continuas, essas interações são ilustradas na Figura 9 (RUMBAUGH; JACOBSON; BOOCH, 1999).

Figura 9 – Diagrama de caso de uso



Fonte: Adaptado de Rumbaugh, Jacobson e Booch (1999).

3 METODOLOGIA

Este trabalho apresenta o desenvolvimento de um *software* para a aquisição dos dados de uma estação de recarga de veículos elétricos e, para isso, será utilizada a metodologia de desenvolvimento guiado a testes. Dessa forma, esta pesquisa pode ser classificada como descritiva com abordagem qualitativa.

De acordo com Gil (2002), "as pesquisas descritivas têm como objetivo primordial a descrição das características de determinada população ou fenômeno ou, então, o estabelecimento de relações entre variáveis". A abordagem qualitativa leva em consideração a relação entre o ambiente e o indivíduo, em que a interpretação dos resultados é o foco principal deste procedimento (SILVA; MENEZES, 2005).

A fundamentação teórica baseou-se na pesquisa do tipo bibliográfica, que, segundo Gil (2002), é um tipo de pesquisa que se origina de outros materiais já criados, que geralmente são compostas de livros e artigos científicos.

3.1 Métodos Aplicados

Para definir as funcionalidades que este *software* deveria fornecer ao cliente, foram definidos os casos de uso do sistema, com os cenários principais de sucesso, assim como os fluxos alternativos. A partir dos casos de uso, foram definidos os requisitos aos quais o programa a ser criado precisaria atender para o completo funcionamento.

Para obter os dados da estação de carregamento, foi utilizada a API baseada em GraphQL, fornecida pela empresa Voltbras, desenvolvedora do equipamento de recarga. Para a conexão com a API, foi fornecida uma chave de acesso, necessária para o envio das requisições.

Na construção deste *software*, foi utilizada a linguagem de programação *Python*, e para a produção do código foi utilizado o editor *VS Code*. Para os testes, foi utilizado o *framework Pytest*, que possui recursos para executar e validar os testes, além da possibilidade de estender funcionalidades com *plugins*. As requisições dos dados a API foram feitas utilizando a biblioteca em Python para requisições HTTP que chama-se *Requests*.

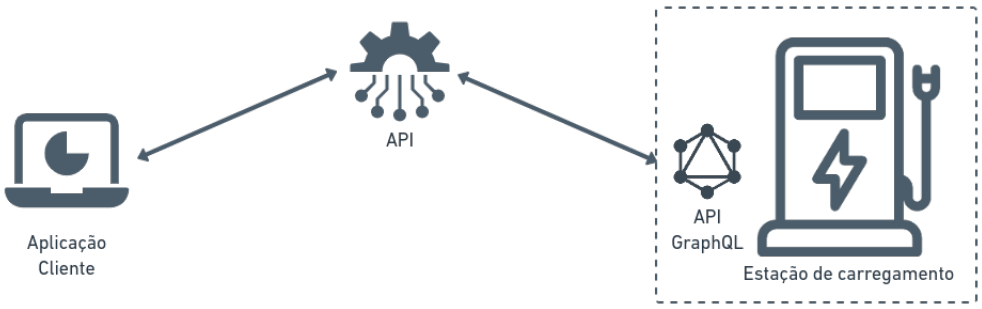
4 CRIAÇÃO DE UM *SOFTWARE* UTILIZANDO DESENVOLVIMENTO GUIADO A TESTES

Nesta seção serão abordadas as etapas para a criação de um *software* para adquirir os dados de uma estação de carregamento de carros elétricos, para isso, determinou-se os casos de uso e os requisitos, utilizando como metodologia de criação o desenvolvimento guiado a teste.

4.1 Descrição do sistema

O sistema que este trabalho desenvolve é a criação de uma API para o acesso aos dados de uma estação de carregamento de veículos elétricos (Figura 9). Esta aplicação tem como objetivo a padronização do formato das requisição aos dados, dessa forma, a haverá uma abstração das tecnologias usadas em diferentes plataformas de carregamento para a aplicação cliente

Figura 10 – Diagrama do sistema



Fonte: Elaboração própria (2021).

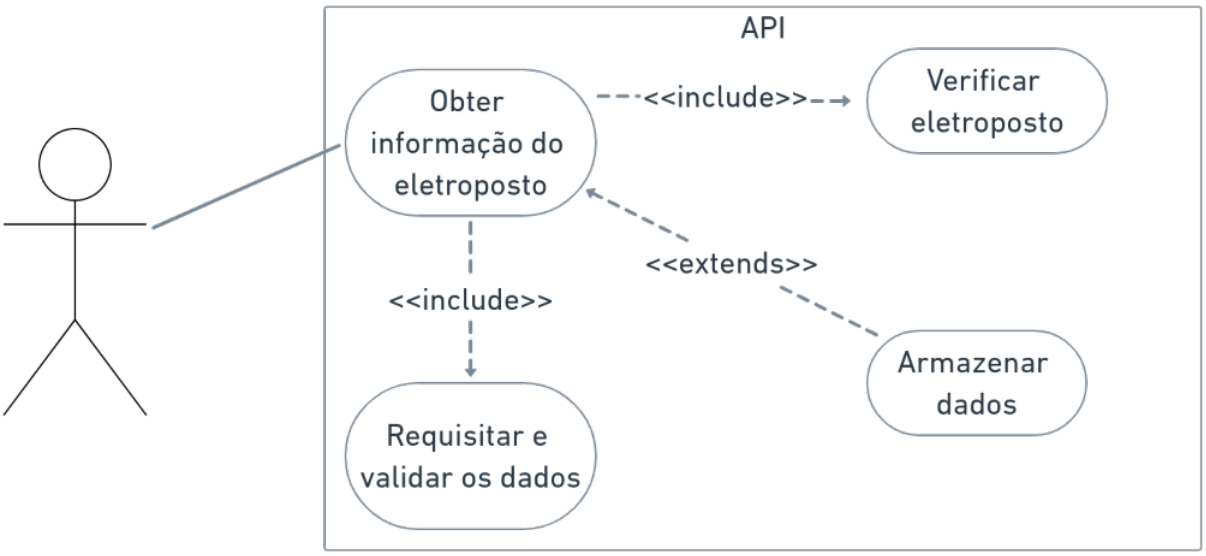
O cliente do *software* desenvolvido, identificado como API, faz a requisição utilizando as rotas definidas pela arquitetura REST. Para a obtenção dos dados, é preciso se conectar a uma API que utiliza a tecnologia GraphQL, fornecida pela empresa Voltbras, fabricante da estação de carregamento.

4.2 Casos de uso

Para produzir o *software*, definiram-se os casos de uso do sistema para entender a dinâmica de interações entre o cliente e o sistema a ser desenvolvido. Com isso, encontraram-se quatro casos de uso: a) obter informação do eletroposto; b) requisitar e validar os dados; c) verificar disponibilidade do eletroposto e c) realizar a persistência dos dados lidos.

A Figura 11 mostra o diagrama de casos de uso em seu fluxo de sucesso, cujo caso de uso principal sempre verifica o estado do EVSE; se estiver disponível, o sistema requisita e valida os dados recebidos, em seguida, retorna ao cliente e o salva.

Figura 11 – Diagrama de caso de uso



Fonte: Elaboração própria (2021).

O primeiro caso de uso (Figura 12), refere-se à comunicação entre o cliente e a API, que, a partir de uma requisição, invoca os outros casos de uso para obter as informações solicitadas e retornar o dado.

Figura 12 – Caso de uso 1

Caso de uso: 1 Obter informação do eletroposto

INFORMAÇÕES CARACTERÍSTICAS

Objetivo em Contexto: O *software* criado irá receber uma requisição de informação onde será responsável por obter essa informação do eletroposto.
Escopo: *software*.
Nível: resumo.
Pré-condições: o eletroposto precisa estar *online*.
Condição final de sucesso: o eletroposto retorna o dado.
Condição final de falha: o eletroposto não retorna o dado.
Ator principal: qualquer pessoa/cliente.
Trigger: o cliente requisita acesso aos dados.

PRINCIPAL CENÁRIO DE SUCESSO

1. O cliente requisita acesso aos dados.
2. Verificar disponibilidade do eletroposto no **caso de uso 3**.
3. Verificar o dado no **caso de uso 2**.
4. O *software* retorna o valor ao cliente.

EXTENSÃO/FLUXO ALTERNATIVO

2a: O eletroposto está *offline*.
2a1: O *software* retorna uma mensagem de erro para informar a indisponibilidade da estação de recarga provinda do caso de uso 3.
3a: O dado é inválido.
3a1: O *software* retorna uma mensagem de erro informando qual dado está inválido provindo do caso de uso 2.

Fonte: Elaboração própria (2021).

O segundo caso de uso é responsável por verificar se a informação solicitada é fornecida pelo sistema, requisitar o dado do eletroposto e verificar se o dado retornado não é nulo e consistente com o tipo esperado, como mostrado na Figura 13.

Figura 13 – Caso de uso 2

Caso de uso: 2 Requisitar e validar os dados

INFORMAÇÕES CARACTERÍSTICAS

Objetivo em Contexto: O *software* criado irá requisitar as informações do eletroposto.

O *software* será capaz de fornecer as seguintes informações:

- ID da estação.
- Tarifa.
- Energia consumida.
- Preço da recarga.
- Potência.

Escopo: *software*.
Nível: resumo
Pré-condições: o eletroposto precisa estar *online*.
Condição final de sucesso: os dados estão acessíveis.
Condição final de falha: os dados não estão acessíveis.
Ator principal: proprietário do software.
Trigger: requisição dos dados ao *software*.

PRINCIPAL CENÁRIO DE SUCESSO

1. O *software* deverá requisitar os seguintes dados do barramento:
    a. ID da estação.
    b. Tarifa.
    c. Energia consumida.
    d. Preço da recarga.
    e. Potência.

EXTENSÃO/FLUXO ALTERNATIVO

1a: O *software* não provê o dado requerido.
1a1: O *software* retorna um erro para informar que o dado não está disponível

Fonte: Elaboração própria (2021).

O terceiro caso de uso (Figura 14) diz respeito à verificação da disponibilidade do eletroposto para a requisição dos dados.

Figura 14 – Caso de uso 3

Caso de uso: 3 Verificar disponibilidade do eletroposto

INFORMAÇÕES CARACTERÍSTICAS

Objetivo em Contexto: O *software* criado irá verificar se o eletroposto está disponível e retornando dados.
Escopo: *software*.
Nível: resumo
Pré-condições: requisição de dados do cliente.
Condição final de sucesso: eletroposto retorna dados.
Condição final de falha: o eletroposto está offline
Ator principal: eletroposto.
Trigger: requisição dos dados ao *software*.

PRINCIPAL CENÁRIO DE SUCESSO

1. O *software* deverá acessar a API Voltbras com uma chave de acesso.
2. O software deverá verificar se a API está disponível ao acesso dos dados**.**

EXTENSÃO/FLUXO ALTERNATIVO

1a: A API Voltbras está *offline*.
1a1: O *software* retorna uma mensagem de erro para informar a indisponibilidade da API.
2a: Os dados da API Voltbras estão indisponíveis.
2a1: O *software* retorna para informar a indisponibilidade dos dados.

Fonte: Elaboração própria (2021).

O caso de uso quatro relaciona-se ao armazenamento dos dados requisitados ao eletroposto, como pode ser visto na Figura 15.

Figura 15 – Caso de uso 4

Caso de uso: 4 Realizar a persistência dos dados lidos

INFORMAÇÕES CARACTERÍSTICAS

Objetivo em Contexto: o *software* criado será responsável por realizar a persistência dos dados lidos.
Escopo: *software.*
Nível: resumo.
Pré-condições: os dados estão disponíveis.
Condição final de sucesso: o dado é salvo.
Condição final de falha: o dado não é salvo.
Ator principal: *software.*
Trigger: recebimento dos dados pelo eletroposto.

PRINCIPAL CENÁRIO DE SUCESSO

1. O *software* deverá receber o dado lido.
2. O *software* deverá salvar o dado em uma base de dados.
3. O *software* deverá disponibilizar os dados para acesso futuro.

EXTENSÃO/FLUXO ALTERNATIVO

1a: Os dados não estão sendo salvos.
1a1: O *software* retorna uma mensagem de erro informando que os dados não estão sendo salvos.
2a: O *software* não consegue acessar o banco de dados.
2a1: O *software* retorna uma mensagem de erro informando que não consegue acessar o banco de dados.

Fonte: Elaboração própria (2021).

Baseando-se nas descrições de caso de uso, foi estabelecido os requisitos do sistema para a criação do software. A Tabela 2 mostra os requisitos estipulados.

Tabela 3 – Requisitos do *software*

Caso de uso	Requisito	Descrição
Caso de uso 1	1.1	O *software* deve receber uma requisição HTTP via arquitetura REST do cliente com as informações de interesse.
	1.2	O *software* deve enviar uma mensagem de erro “O eletroposto está OFFLINE” se a estação de recarga estiver inativa.
	1.3	O *software* deve enviar uma mensagem de erro “O parâmetro ‘X’ não foi encontrado” se o dado não estiver indisponível.
	1.4	O *software* deve enviar o dado para o cliente via requisição HTTP, juntamente com o status da requisição seguido de uma mensagem sobre o ocorrido.
Caso de uso 2	2.1	O *software* deve verificar se o dado requisitado é provido pelo *software*: ID da estação - tipo: *string*; tarifa - tipo: *float*; energia consumida - inteiro; preço da recarga - tipo: *float*; Potência - tipo: *float*.
	2.2	O *software* deve retornar um erro se o dado não for provido.
	2.3	O *software* deve requisitar o dado solicitado ao eletroposto utilizando uma chave de acesso por meio da arquitetura GraphQL.
	2.4	O *software* deve retornar um erro se o dado não for retornado.
	2.5	O *software* deve verificar se os dados retornados pelo eletroposto não são nulos ou de tipo diferente do esperado.
Caso de uso 3	3.1	O *software* deve se conectar à API Voltbras utilizando uma chave de acesso por meio da arquitetura GraphQL.
	3.2	O *software* deve retornar falso caso a API Voltbras estiver *offline* e retornar verdadeiro caso esteja *online*.
Caso de uso 4	4.1	O *software* deve salvar os dados requisitados no requisito R2.1 em um arquivo em formato JSON.
	4.2	O *software* deve enviar uma mensagem de erro "O dado ‘X’ não foi armazenado"se os dados não forem armazenados.
	4.3	O *software* deve armazenar junto ao dado o horário em formato *timestamp* e a ID da estação requisitada.

Fonte: Elaboração Própria (2021).

Uma tabela de erros foi criada para especificar seus códigos com o seu

significado, que pode ser observado na Tabela 4.

Tabela 4 – Erros do *software*

Erro	Requisito	Código
Eletroposto não retorna o dado	2.2	-1
Dado solicitado não é provido pelo sistema	2.1	-2
Dado retornado pelo EVSE não é como o esperado	2.4	-3

Fonte: Elaboração Própria (2021).

4.3 Criação do *software*

Uma vez que os requisitos de *software* foram elencados, pôde-se dividir o desenvolvimento dos códigos. O software foi desenvolvido em paralelo, entre a autora e a acadêmica Letícia de Oliveira Nunes, separando os requisitos dos casos de uso em duas partes: a primeira delas relaciona-se aos casos de uso um e quatro, e a parte abordada por este trabalho, refere-se aos casos de uso dois e três.

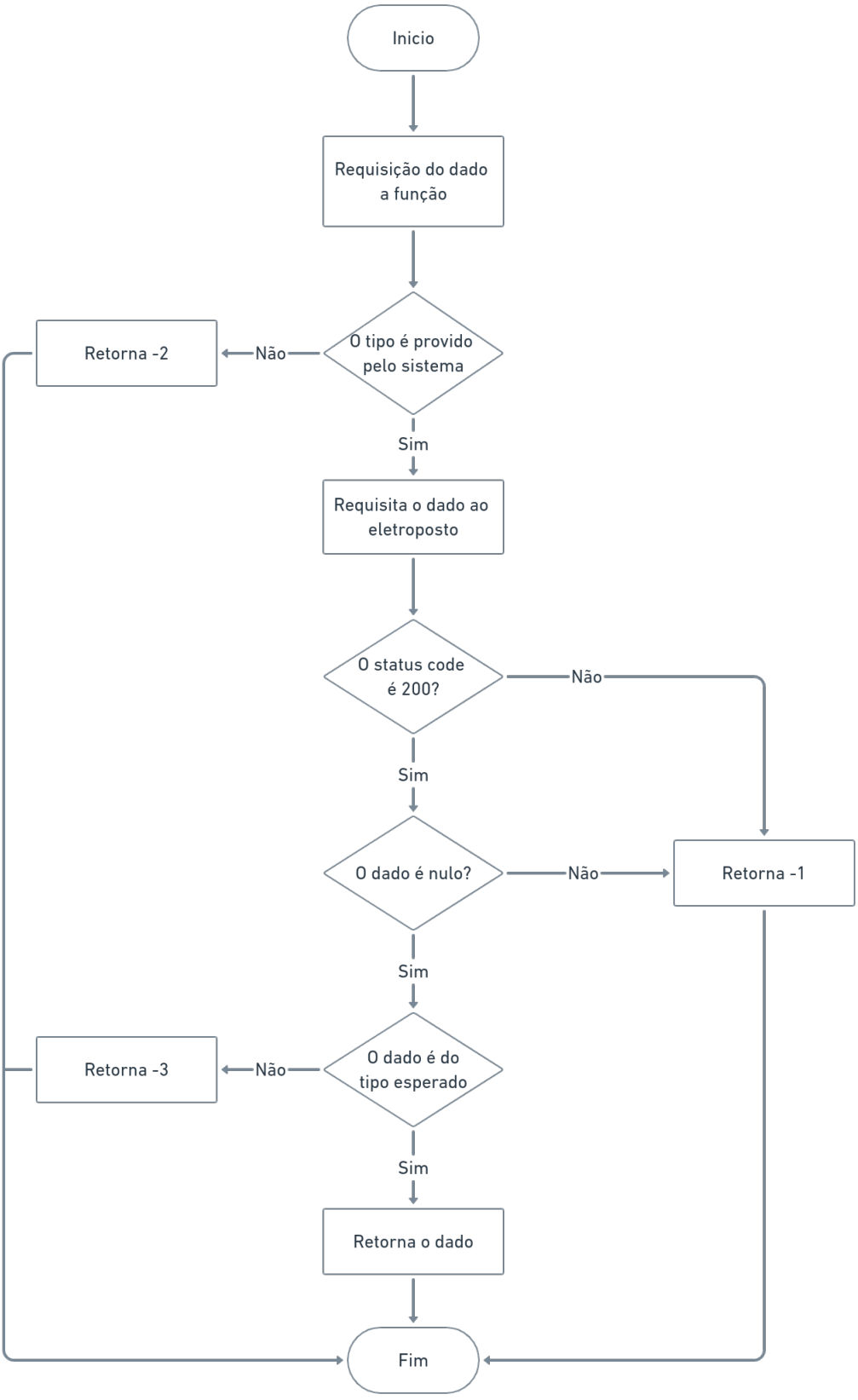
O programa foi desenvolvido seguindo os métodos de TDD criado por Beck (2003); portanto, seguiu o fluxo de criação de testes falhos, criação de um código mínimo para a aceitação do teste e a refatoração. Utilizou-se a linguagem de programação *Python* para o desenvolvimento do *software* em conjunto com a biblioteca *Pytest* para a criação dos testes.

4.3.1 Aquisição dos dados ao eletroposto

O caso de uso dois tem como objetivo obter os dados do eletroposto, verificar se os dados são providos pelo sistema e se tais informações retornam da forma esperada. Na Figura 16, pode-se ver o fluxograma desse caso de uso.

Quando o sistema recebe uma requisição, é feito uma confirmação da disponibilidade dessa informação. Caso seja disponível, o *software* faz uma requisição à API do eletroposto. Se o EVSE não retornar um erro, é realizada uma verificação da consistência do dado recebido; para isso, é analisado se a informação não é nula ou de um tipo diferente do esperado. Quando correto, o dado é retornado, caso contrário, é repassado um erro.

Figura 16 – Fluxograma do caso de uso dois



Fonte: Elaboração própria (2021).

Para obter os dados do eletroposto, foi preciso se conectar a uma API externa, que utiliza a linguagem GraphQL, e realizar uma requisição com o parâmetro desejado. O teste para essa aplicação é do tipo funcional. O Código 4.1 mostra um exemplo de teste para o parâmetro ID, em que existe a comparação entre o dado esperado de retorno e o dado real do eletroposto.

Código 4.1 – Teste para a requisição do ID

```
1    def test_query_data_from_evse():
2        expected_id = {"stations": [{"id":"7eeab9db-d9c9-4865-bbda-9090baf9c884"}]}
3        result_id = get_data_from_evse("id")
4        assert result_id == expected_id
5
```

Fonte: Elaboração própria (2021).

No primeiro momento, espera-se que o teste falhe, pois a função que solicita o dado ao eletroposto (*get_data_from_evse*) não existe, esse comportamento pode ser observado na Figura 17.

Figura 17 – Teste falho da aquisição dos dados do eletroposto

```
marieli in Documentos/Projetos/tcc
❯ via tcc pytest tests/functional
========================== test session starts ==========================
platform linux -- Python 3.9.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /home/marieli/Documentos/Projetos/tcc
plugins: mock-3.6.1
collected 0 items / 1 error

================================ ERRORS =================================
____________ ERROR collecting tests/functional/test_evse.py ____________
ImportError while importing test module '/home/marieli/Documentos/Projet
os/tcc/tests/functional/test_evse.py'.
Hint: make sure your test modules/packages have valid Python names.
Traceback:
/usr/lib/python3.9/importlib/__init__.py:127: in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
tests/functional/test_evse.py:2: in <module>
    from api_axs.data_evse import get_data_from_evse
E   ModuleNotFoundError: No module named 'api_axs'
====================== short test summary info ======================
ERROR tests/functional/test_evse.py
!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!
============================ 1 error in 0.07s ============================
```

Fonte: Elaboração própria (2021).

Para desenvolver o código mínimo para o teste passar, foi criada essa função com o retorno dos dados fixos, que simulam a resposta do eletroposto, como pode ser observado no Código 4.2

Código 4.2 – Código mínimo para o teste passar

```
6     def get_data_from_evse(req):
7         if (req == "id"):
8             result = {"stations": [{"id": "7eeab9db-d9c9-4865-bbda-9090baf9c884"}]}
9             status_code = 200
10            if status_code == 200:
11                return result
12            else:
13                return -1
14
```

Fonte: Elaboração própria (2021).

Com a adição dessa função, o teste é aceito, como pode ser observado na Figura 18.

Figura 18 – Resultado do teste com dados simulados

```
marieli in Documentos/Projetos/tcc
❯ via tcc make functional-tests
pytest -rP tests/functional/ --verbose
========================== test session starts ==========================
platform linux -- Python 3.9.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1 -
- /home/marieli/Documentos/Projetos/tcc/bin/python
cachedir: .pytest_cache
rootdir: /home/marieli/Documentos/Projetos/tcc
plugins: clarity-1.0.1
collected 1 item

tests/functional/test_evse.py::test_query_data_from_evse PASSED  [100%]

================================ PASSES =================================
========================== 1 passed in 0.07s ============================
```

Fonte: Elaboração própria (2021).

Para solicitar os dados do eletroposto, o código foi refatorado e utilizou-se a biblioteca Request para requisitar o dado, como mostra o Código 4.3.

Código 4.3 – Função para solicitar os dados do eletroposto

```
15    def get_data_from_evse(req):
16        if req == "id":
17            data = { 'query' : '{ stations { id }}' }
18            result = requests.post(BASE_URL, json=data, headers={"api-key": TOKEN})
19            if result.status_code == 200:
20                return result.json()["data"]
21            else:
22                return -1
23
```

Fonte: Elaboração própria (2021).

Para requisitar os dados do eletroposto é preciso montar uma *query*, que utiliza o formato JSON em conjunto com uma palavra chave para especificar o tipo de operação a ser executada. Além disso, é preciso indicar a base de dados e o dado desejado.

O resultado do teste com a refatoração pode ser visto na Figura 19. Com a requisição ao eletroposto, é possível perceber um acréscimo no tempo do teste, que passa de 0,07s com os dados simulados para 0,56s com os dados reais.

Figura 19 – Resultado do teste com os dados vindo do eletroposto

```
marieli in Documentos/Projetos/tcc
❯ via tcc make functional-tests
pytest -rP tests/functional/ --verbose
========================== test session starts ==========================
platform linux -- Python 3.9.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1 -
- /home/marieli/Documentos/Projetos/tcc/bin/python
cachedir: .pytest_cache
rootdir: /home/marieli/Documentos/Projetos/tcc
plugins: clarity-1.0.1
collected 1 item

tests/functional/test_evse.py::test_query_data_from_evse PASSED  [100%]

================================ PASSES =================================
========================== 1 passed in 0.56s ============================
```

Fonte: Elaboração própria (2021).

Outra funcionalidade que esse caso de uso precisa fornecer refere-se à análise do dado retornado do eletroposto. O teste de verificação do ID é mostrado no Código 4.4. É examinado se o tipo do dado retornado pelo eletroposto é correspondente com o tipo fornecido na documentação de sua API .

Código 4.4 – Teste para verificar o tipo do retorno

```
24        def test_data_received_is_correct():
25            data_id = verify_data("id", "7eeab9db-d9c9-4865-bbda-9090baf9c884")
26            assert type(data_id) is str
27
```

Fonte: Elaboração própria (2021).

Espera-se que no primeiro momento o teste não passe, pois a função para verificar os dados não existe, como pode ser observado na Figura 20.

Figura 20 – Verificação da disponibilidade do dado

```
❯ via tcc pytest tests/unit
========================= test session starts ==========================
platform linux -- Python 3.9.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /home/marieli/Documentos/Projetos/tcc
plugins: clarity-1.0.1
collected 0 items / 1 error

================================ ERRORS ================================
_____________ ERROR collecting tests/unit/test_data.py _____________
ImportError while importing test module '/home/marieli/Documentos/Projet
os/tcc/tests/unit/test_data.py'.
Hint: make sure your test modules/packages have valid Python names.
Traceback:
/usr/lib/python3.9/importlib/__init__.py:127: in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
tests/unit/test_data.py:2: in <module>
    from data_evse import verify_data
E   ImportError: cannot import name 'verify_data' from 'data_evse' (/hom
e/marieli/Documentos/Projetos/tcc/data_evse.py)
====================== short test summary info =======================
ERROR tests/unit/test_data.py
!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!
========================= 1 error in 0.13s =========================
```

Fonte: Elaboração própria (2021).

Um exemplo de implementação para o ID é mostrado no Código 4.5, em que é verificado qual o tipo da informação; e se não for o esperado, retorna-se um erro.

Código 4.5 – Verificação do tipo de dado

```
28    def verify_data(data, value):
29        if data == "id":
30            if type(value) == str:
31                return value
32            else:
33                return -3
34
```

Fonte: Elaboração própria (2021).

Com essa nova função, o teste criado no Código 4.4 é aprovado, como se pode observar na Figura 21.

Figura 21 – Teste da verificação da disponibilidade do dado

```
marieli in Documentos/Projetos/tcc
❯ via tcc make unit-test
pytest -rP tests/unit/ --verbose
========================= test session starts =========================
platform linux -- Python 3.9.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1 -
- /home/marieli/Documentos/Projetos/tcc/bin/python
cachedir: .pytest_cache
rootdir: /home/marieli/Documentos/Projetos/tcc
plugins: clarity-1.0.1
collected 1 item

tests/unit/test_data.py::test_data_received_is_correct PASSED    [100%]

============================== PASSES ==============================
========================== 1 passed in 0.06s ==========================
```

Fonte: Elaboração própria (2021).

Foram criados testes para verificar se é repassado um erro, caso o dado retornado pelo eletroposto não seja o esperado, como mostrado no Código 4.6 para o exemplo do id, em que é esperado um tipo *string*, mas é recebido um inteiro.

Código 4.6 – Teste para erro de retorno de dado do eletroposto

```
35    def test_if_received_error():
36        expected_error = -3
37        data_id =  verify_data("id", 12)
38
39        assert data_id == expected_error
40
```

Fonte: Elaboração própria (2021).

O teste é aceito, pois esta função já está implementada (Código 4.5), como mostrado na Figura 22.

Figura 22 – Resultado do teste para verificação do dado

```
marieli in api-axs on  teste_dados_do_eletroposto [⇡✘!?]
❯ via api-axs make unit-test
pytest -rP tests/unit/ --verbose
============================= test session starts ==============================
platform linux -- Python 3.9.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1 -- /home/ma
rieli/Documentos/Projetos/api-axs/bin/python
cachedir: .pytest_cache
rootdir: /home/marieli/Documentos/Projetos/api-axs
plugins: mock-3.6.1, clarity-1.0.1, requests-mock-1.9.3, print-0.3.0, cov-2.12.1
collected 2 items

tests/unit/test_data.py::test_data_received_is_correct PASSED             [ 50%]
tests/unit/test_data.py::test_if_received_error PASSED                    [100%]

=================================== PASSES ===================================
============================== 2 passed in 0.02s ===============================
```

Fonte: Elaboração própria (2021).

A função responsável por receber as chamadas do caso de uso um e retornar ao resultado das verificações requisitadas, tem seu teste mostrado no Código 4.7.

Código 4.7 – Teste para a função chamada pelo caso de uso um

```
41    def test_that_receives_ordem_and_returns_value():
42        expected_data_id = "7eeab9db-d9c9-4865-bbda-9090baf9c884"
43        received_data_id = request_data("id")
44        assert received_data_id == expected_data_id
45
```

Fonte: Elaboração própria (2021).

Como esperado, o teste falhou, pois essa função não existe. Porém, ao ser implementada, o teste passou. Esses comportamentos são mostrados na Figura 23.

Figura 23 – Teste para a função chamada pelo caso de uso um

(a) Teste falho

```
marieli in Documentos/Projetos/tcc
❯ via tcc pytest tests/unit
========================= test session starts =========================
platform linux -- Python 3.9.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /home/marieli/Documentos/Projetos/tcc
plugins: clarity-1.0.1
collected 0 items / 1 error

=============================== ERRORS ================================
_____________ ERROR collecting tests/unit/test_data.py _____________
ImportError while importing test module '/home/marieli/Documentos/Projet
os/tcc/tests/unit/test_data.py'.
Hint: make sure your test modules/packages have valid Python names.
Traceback:
/usr/lib/python3.9/importlib/__init__.py:127: in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
tests/unit/test_data.py:2: in <module>
    from data_evse import verify_data, request_data
E   ImportError: cannot import name 'request_data' from 'data_evse' (/ho
me/marieli/Documentos/Projetos/tcc/data_evse.py)
======================= short test summary info =======================
ERROR tests/unit/test_data.py
!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!
========================== 1 error in 0.13s ===========================
```

(b) Teste aprovado

```
marieli in api-axs on  teste_dados_do_eletroposto [⇡✘!?] took 2s
❯ via api-axs make unit-test
pytest -rP tests/unit/ --verbose
============================= test session starts ==============================
platform linux -- Python 3.9.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1 -- /home/mar
ieli/Documentos/Projetos/api-axs/bin/python
cachedir: .pytest_cache
rootdir: /home/marieli/Documentos/Projetos/api-axs
plugins: mock-3.6.1, clarity-1.0.1, requests-mock-1.9.3, print-0.3.0, cov-2.12.1
collected 3 items

tests/unit/test_data.py::test_that_receives_ordem_and_returns_value PASSED  [ 33%]
tests/unit/test_data.py::test_data_received_is_correct PASSED               [ 66%]
tests/unit/test_data.py::test_if_received_error PASSED                      [100%]

=================================== PASSES ====================================
================================== 3 passed in 0.67s ==========================
```

Fonte: Elaboração própria (2021).

O Código 4.8 mostra a implementação de um exemplo desta função para o parâmetro ID, bem como a chamada das outras funções implementadas anteriormente.

Código 4.8 – Função para a chamada do caso de uso um

```
46      def request_data(req):
47          if req == "id":
48              data = get_data_from_evse("id")
49              data = data["stations"][0]["id"]
50              verified_data = verify_data("id", data)
51              return verified_data
52
```

Fonte: Elaboração própria (2021).

O último requisito para esse caso de uso refere-se ao da disponibilidade do dado solicitado pelo cliente. Caso o dado não seja provido, o sistema deve retornar um erro ao cliente; para isso, foi criado um teste que pode ser observado no Código 4.8.

Código 4.9 – Teste que verifica a disponibilidade da informação

```
53      def test_wrong_request_to_evse():
54          expected = -2
55          result = request_data("ip")
56          assert result == expected
57
```

Fonte: Elaboração própria (2021).

Para isso, foi adicionada uma verificação ao código criado na função mostrada no Código 4.8, em que analisa se a requisição recebida é provida pelo sistema; caso não seja, um erro é retornado. Esse comportamento pode ser visto no Código 4.10.

Código 4.10 – Função chamada pelo caso de uso um

```
58      def request_data(req):
59          if req == "id":
60              data = get_data_from_evse("id")
61              data = data["stations"][0]["id"]
62              verified_data = verify_data("id", data)
63              return verified_data
64          else:
65              return -2
66
```

Fonte: Elaboração própria (2021).

O resultado desse teste pode ser conferido na Figura 24.

Figura 24 – Teste da função chamada pelo caso de uso principal

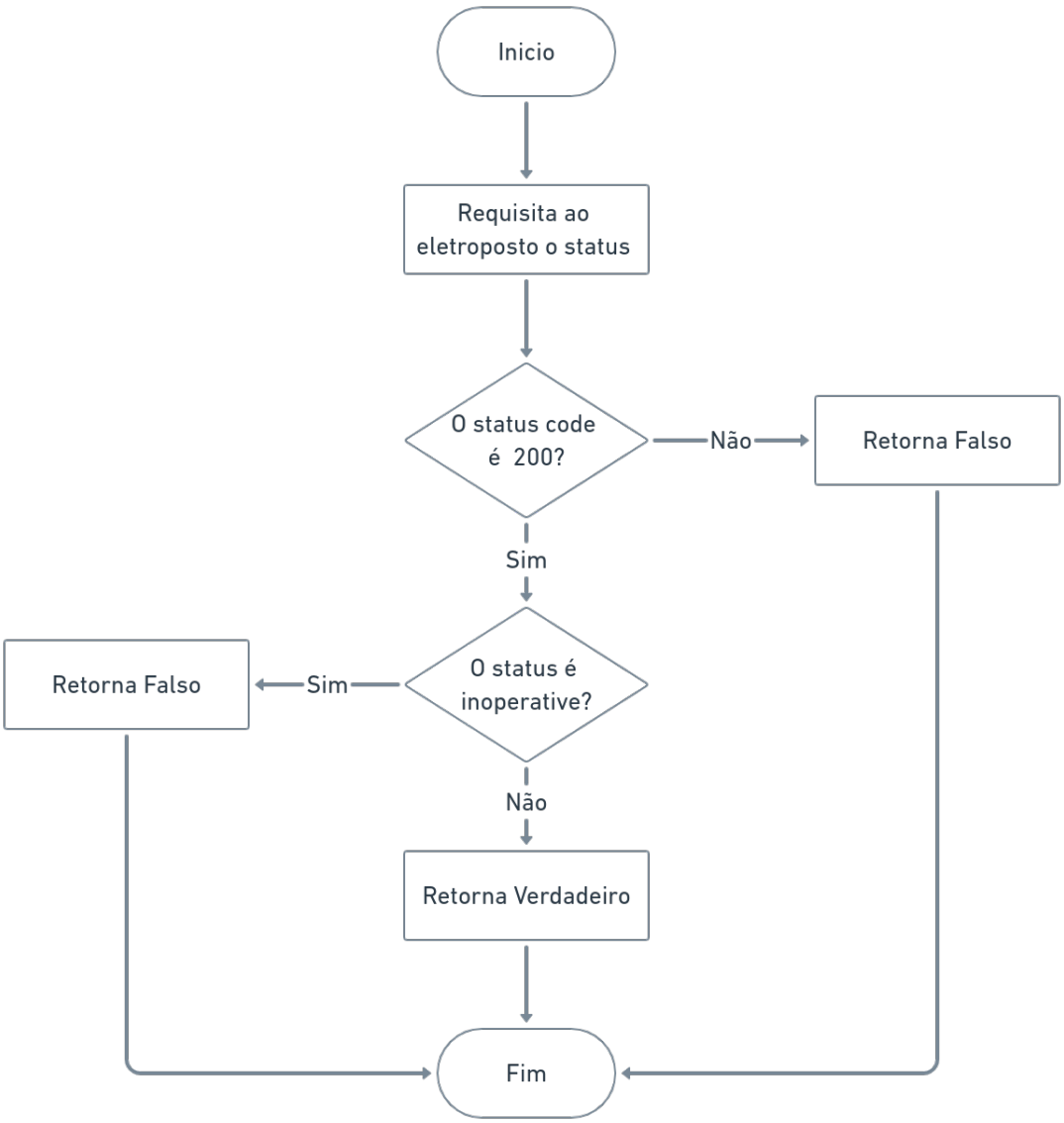
```
marieli in api-axs on  teste_dados_do_eletroposto [!✘!?] took 2s
❯ via api-axs make unit-test
pytest -rP tests/unit/ --verbose
============================= test session starts ==============================
platform linux -- Python 3.9.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1 -- /home/ma
rieli/Documentos/Projetos/api-axs/bin/python
cachedir: .pytest_cache
rootdir: /home/marieli/Documentos/Projetos/api-axs
plugins: mock-3.6.1, clarity-1.0.1, requests-mock-1.9.3, print-0.3.0, cov-2.12.1
collected 4 items

tests/unit/test_data.py::test_that_receives_ordem_and_returns_value PASSED [ 25%]
tests/unit/test_data.py::test_data_received_is_correct PASSED              [ 50%]
tests/unit/test_data.py::test_if_received_error PASSED                     [ 75%]
tests/unit/test_data.py::test_wrong_request_to_evse PASSED                 [100%]

==================================== PASSES ====================================
============================== 4 passed in 1.13s ===============================
```

Fonte: Elaboração própria (2021).

4.3.2 Verificação de disponibilidade do eletroposto

O caso de uso três refere-se à verificação da disponibilidade de acesso aos dados do eletroposto. Para essa verificação, existem na API da Voltbras quatro tipos de *status*. Quando a estação está em operação, é possível ter três resultados: *unknown*, *charging* e *available*. Quando a mesma não está disponível, é retornado o status *inoperative*. A função que verifica a disponibilidade deve retornar "verdadeiro", caso o eletroposto esteja disponível para o acesso aos dados, e responder "falso", caso os dados estejam indisponíveis. A Figura 22 mostra o fluxograma dos requisitos deste caso de uso.

Figura 25 – Fluxograma da implementação do caso de uso dois



Fonte: Elaboração própria (2021).

Ao receber uma requisição, o sistema solicita o status ao eletroposto; caso não ocorra erro, o software verifica o dado retornado. Se o valor for diferente de *unknown*, *charging* e *available* é retornado "falso"para o cliente, caso contrário é retornado o valor verdadeiro. O Código 4.11 mostra o teste criado para essa verificação.

Código 4.11 – Função que verifica a disponibilidade do EVSE

```
67      def test_if_evse_is_working():
68          expected_data = True
69          received_data = verify_evse()
70          assert received_data == expected_data
71
```

Fonte: Elaboração própria (2021).

Esse teste deve falhar quando rodado pela primeira vez, pois a função não existe, como mostrado na Figura 26.

Figura 26 – Teste falho da função para verificar a disponibilidade do eletroposto

```
marieli in Documentos/Projetos/tcc
❯ via tcc pytest tests/functional
============================= test session starts ==============================
platform linux -- Python 3.9.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /home/marieli/Documentos/Projetos/tcc
plugins: clarity-1.0.1
collected 0 items / 1 error

==================================== ERRORS ====================================
_________________ ERROR collecting tests/functional/test_evse.py _________________
ImportError while importing test module '/home/marieli/Documentos/Projetos/tcc/tes
ts/functional/test_evse.py'.
Hint: make sure your test modules/packages have valid Python names.
Traceback:
/usr/lib/python3.9/importlib/__init__.py:127: in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
tests/functional/test_evse.py:3: in <module>
    from evse import verify_evse
E   ImportError: cannot import name 'verify_evse' from 'evse' (/home/marieli/Docum
entos/Projetos/tcc/evse.py)
=========================== short test summary info ============================
ERROR tests/functional/test_evse.py
!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!
=============================== 1 error in 0.13s ===============================
```

Fonte: Elaboração própria (2021).

A implementação dessa funcionalidade pode ser visto no Código 4.12, que requisita o status para o eletroposto. Ao possuir o *status code* 200, verifica-se o retorno, e, de acordo com a resposta, retorna verdadeiro ou falso.

Código 4.12 – Função que verifica a disponibilidade do eletroposto

```
72      def verify_evse():
73          data = {"query": "{ stations { status }}"}
74          result = requests.post(BASE_URL, json=data, headers={"api-key": TOKEN})
75
76          if result.status_code == 200:
77              data = result.json()["data"]["stations"][0]["status"]
78              if (data == "AVAILABLE") or (data == "CHARGING") or (data == "UNKNOWN"):
79                  return True
80              else:
81                  return False
82          else:
83              return False
84
```

Fonte: Elaboração própria (2021).

Os resultados dos testes podem ser vistos na Figura 27, que confirma o funcionamento dos testes do caso de uso três, e também, dos casos criados para o caso de uso dois.

Figura 27 – Teste aprovado da função para verificar a disponibilidade do eletroposto

```
marieli in api-axs on  teste_dados_do_eletroposto [⇡✘!?]
❯ via api-axs make functional-tests
pytest -rP tests/functional/ --verbose
============================= test session starts ==============================
platform linux -- Python 3.9.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1 -- /home/ma
rieli/Documentos/Projetos/api-axs/bin/python
cachedir: .pytest_cache
rootdir: /home/marieli/Documentos/Projetos/api-axs
plugins: mock-3.6.1, clarity-1.0.1, requests-mock-1.9.3, print-0.3.0, cov-2.12.1
collected 2 items

tests/functional/test_evse.py::test_query_data_from_evse PASSED            [ 50%]
tests/functional/test_evse.py::test_if_evse_is_working PASSED              [100%]

==================================== PASSES ====================================
============================== 2 passed in 1.19s ===============================
```

Fonte: Elaboração própria (2021).

4.3.3 Integração com os casos de uso um e quatro

A integração dos códigos dos requisitos dos casos de uso um e quatro, desenvolvidos por outra colaboradora, junto aos criados neste trabalho, foi implementada por meio de chamadas de funções, apresentadas nos Código 4.10 e no Código 4.12. Para um usuário requisitar esses dados, é preciso se conectar às rotas específicas para cada parâmetro, mostradas na Tabela 5.

Tabela 5 – Rotas de acesso aos dados do eletroposto

Rota	Descrição
URL/id	Parâmetro de id da estação
URL/tarifa	Parâmetro do valor da tarifa
URL/energiaconsumida	Parâmetro de energia consumida pelo veículo
URL/preco	Parâmetro de valor total da recarga
URL/potencia	Parâmetro de potência da estação
URL/todos	Parâmetro para adquirir todos os parâmetro disponibilizado por esta API

Fonte: Elaboração Própria (2021).

Caso o dado seja recebido com sucesso, é retornada ao cliente uma mensagem com o *status code* 200, em conjunto com o valor e uma mensagem de sucesso. No entanto, quando um erro surge no processo de aquisição ou verificação da validade

desse dado, uma mensagem de erro é retornada em conjunto com o código do erro em específico.

Como não foi possível implementar os testes com um carro conectado à estação de carregamento, os parâmetros de energia consumida, preço e todos os dados retornam erro quando requisitados, pois dependem dessa conexão para possuir um valor, esse comportamento pode ser visto na Figura 29.

Figura 28 – Parâmetros que retornam erro

(a) Energia consumida

```
// 20210910141148
// http://127.0.0.1:5000/energiaconsumida

{
  "msg": "O parâmetro ENERGIA CONSUMIDA  não foi encontrado",
  "status": "-1"
}
```

(b) Preço

```
// 20210910141226
// http://127.0.0.1:5000/preco

{
  "msg": "O parâmetro PREÇO  não foi encontrado",
  "status": "-1"
}
```

(c) Todos os dados

```
// 20210910142250
// http://127.0.0.1:5000/todos

{
  "msg": "O parâmetro todos não foi encontrado",
  "status": "-1"
}
```

Fonte: Elaboração própria (2021).

Nos casos em que o dado não depende de conexão com o carro (ID, potência e tarifa) a API retorna um valor, como se pode observar na figura 26.

Figura 29 – Parâmetros que retornam um valor

(a) EID

```
// 20210908090114
// http://127.0.0.1:5000/id

{
  "id": "7eeab9db-d9c9-4865-bbda-9090baf9c884",
  "msg": "O parâmetro ID foi encontrado",
  "status": "200"
}
```

(b) Potência

```
// 20210910141358
// http://127.0.0.1:5000/potencia

{
  "msg": "O parâmetro POTÊNCIA foi encontrado",
  "potencia": 22000,
  "status": "200"
}
```

(c) Tarifa

```
// 20210910141052
// http://127.0.0.1:5000/tarifa

{
  "msg": "O parâmetro TARIFA foi encontrado",
  "status": "200",
  "tarifa": 0
}
```

Fonte: Elaboração própria (2021).

O armazenamento dos dados foi realizado em forma de JSON, em que é salvo o id da estação de carregamento, em conjunto com o valor do parâmetro escolhido e do instante de tempo da requisição. Essa base de dados pode ser vista na Figura 30.

Figura 30 – Base de dados


```
{
    "clients": [
        {
            "id": "7eeab9db-d9c9-4865-bbda-9090baf9c884",
            "tarifa": 0,
            "timestamp": 1630697804.528685
        },
        {
            "id": "7eeab9db-d9c9-4865-bbda-9090baf9c884",
            "potencia": 22000,
            "timestamp": 1630697808.40912
        }
    ]
}
```


Fonte: Elaboração própria (2021)..

Com a integração dos casos de uso, conclui-se o funcionamento geral do sistema. Dessa forma, um usuário é capaz de a partir de, uma requisição de a uma API, obter os dados de uma EVSE, desde que tal parâmetro esteja previsto no projeto.

5 CONSIDERAÇÕES FINAIS

Este trabalho visou à elaboração de um *software* para a aquisição de dados de uma estação de carregamento de veículos elétricos, utilizando a metodologia de desenvolvimento guiado a teste, tendo como objetivo o acesso às informações geradas por esses eletropostos de forma facilitada, para que fosse possível obter dados de gerenciamento.

A criação desse programa utilizando as técnicas de TDD mostrou-se satisfatório, sem perda de produtividade; o ciclo de *red*, *green*, *refactor* foi implementado em todas as funcionalidades criadas, apesar de a etapa de refatoração ter se mostrado muito simples ou desnecessária devido à baixa complexidade do código. Porém, ao aplicar essa técnica, a sensação de segurança durante o desenvolvimento aumentou, pois os testes asseguravam o funcionamento correto da aplicação ao adicionar ou modificar trechos de código.

A falta de diálogo com os usuários finais se mostrou um desafio para o levantamento de requisitos para o desenvolvimento do *software*; pouca informação foi oferecida em relação às necessidades dos interessados deste serviço para ter como base para a criação dos casos de uso do sistema.

Apesar dessa dificuldade, o fato de ter-se utilizado uma metodologia de desenvolvimento de *software*, com a criação dos casos de uso e seus requisitos, possibilitou a divisão do desenvolvimento da API entre duas pessoas. Isto traz uma abordagem semelhante ao que se é empregado nas empresas, onde a carga de criação de um *software* pode ser compartilhada a partir de uma documentação UML concisa e objetiva.

Aos trabalhos futuros, recomenda-se uma reunião com os usuários finais deste *software* para obter *feedback* das funcionalidades implementadas. Além disso, sugere-se subir o código em um servidor para testá-lo em produção. E, por fim, aconselha-se testar o programa com um carro conectado na estação de carregamento para verificar o funcionamento do projeto.

REFERÊNCIAS

ALMAGHREBI, A. *et al.* Analysis of user charging behavior at public charging stations. In: *2019 IEEE Transportation Electrification Conference and Expo (ITEC)*. [S.l.: s.n.], 2019. p. 1–6. Citado na página 9.

ALMEIDA, J. R. C. U. *et al.* Veículos elétricos no brasil, desafios para a sua adoção e seu potencial de contribuição na redução dos gases de efeito estufa. In: *Congresso Nacional de Engenharia Mecânica*. Salvador: [s.n.], 2018. p. 1–10. Citado na página 10.

AMBLER, S. *Introduction to test driven development (TDD)*. 2013. Disponível em: http://agiledata.org/essays/tdd.html. Acesso em: 30 jun. 2021. Citado na página 12.

APOLLO. *Resolvers*. 2019. Disponível em: https://www.apollographql.com/docs/apollo-server/data/resolvers/. Acesso em: 30 jun. 2021. Citado na página 26.

ARORAA, G.; DASH, T. *Building RESTful Web Services with .NET Core*. 1. ed. Birmingham: Packt Publishing, 2018. 323 p. Citado 3 vezes nas páginas 19, 21 e 25.

ASTELS, D. *Test-Driven Development: A Practical Guide: A Practical Guide*. 2. ed. [S.l.]: Prentice Hall, 2003. 592 p. Citado 2 vezes nas páginas 13 e 14.

BARBER, D. *Why test-driven development?* 2012. Disponível em: http://derekbarber.ca/blog/2012/03/27/why-test-driven-development/. Acesso em: 30 jun. 2021. Citado na página 12.

BECK, K. Aim, fire. *IEEE Software*, IEEE, p. 87–89, 2001. Citado na página 12.

BECK, K. *Test-Driven Development: by exemple*. 1. ed. [S.l.]: Addison-Wesley Professional, 2003. 240 p. Citado 2 vezes nas páginas 12 e 14.

BHAT, T.; NAGAPPAN, N. Evaluating the efficacy of test-driven development: Industrial case studies. In: *2006 International Symposium on Empirical Software Engineering (ISESE 2006)*. Rio de Janeiro: [s.n.], 2006. p. 1–8. Citado na página 16.

BIEHL, M. *API Architecture: The Big Picture for Building APIs*. 1. ed. [S.l.: s.n.], 2015. 192 p. Citado 2 vezes nas páginas 23 e 24.

BISSIA, W.; NETO, A. G. S. S.; EMER, M. C. F. P. The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology*, Elsevier, v. 74, p. 45–54, jun. 2016. Disponível em: https://doi.org/10.1016/j.infsof.2016.02.004. Citado na página 9.

BLOOMBERGNEF. *Electric Vehicle: outlook 2021*. 2021. Disponível em: https://bnef.turtl.co/story/evo-2021/page/3/1?teaser=yes. Acesso em: 14 jun 2021. Citado na página 9.

BOSE, S. *End To End Testing: A Detailed Guide*. 2021. Disponível em: https://www.browserstack.com/guide/end-to-end-testing. Acesso em: 30 jun. 2021. Citado na página 16.

BRASIL. EPE. *Demanda de energia dos veículos leves: 2021-2030*. [S.l.], 2020. Disponível em: https://www.epe.gov.br/sites-pt/publicacoes-dados-abertos/publicacoes/PublicacoesArquivos/publicacao-331/topico-569/Informe_Demanda_Veículos_Leves_2021_2030.pdf. Citado na página 9.

CAO, Y. *et al.* Toward efficient, scalable, and coordinated on-the-move ev charging management. *IEEE Wireless Communications*, IEEE, v. 24, n. 2, p. 66–73, 2017. Disponível em: https://doi.org/10.1109/MWC.2017.1600254WC. Citado na página 9.

COCKBURN, A. *Writing Effective Use Cases*. 1. ed. Boston: Addison-Wesley, 2000. 249 p. Citado na página 27.

COOKSEY, B. *An Introduction to APIs*. 1. ed. [S.l.]: Zapier, 2014. 72 p. Citado na página 23.

COSTA, E. *et al.* Diffusion of electric vehicles in brazil from the stakeholders' perspective. *International Journal of Sustainable Transportation*, Taylor and Francis Online, v. 14, n. 9, p. 1–14, 2020. Disponível em: https://doi.org/10.1080/15568318.2020.1827317. Citado na página 9.

EVERIS. *GraphQL – is it the new REST?* 2019. Disponível em: https://everis.passle.net/post/102fjja/graphql-is-it-the-new-rest-part-4-of-4. Acesso em: 30 jun. 2021. Citado na página 26.

FOWLER, M. *UML Distilled*. 3. ed. Boston: Addison-Wesley, 2003. 209 p. Citado 3 vezes nas páginas 26, 27 e 28.

FOWLER, M. *Broad stack test*. 2013. Disponível em: https://martinfowler.com/bliki/BroadStackTest.html. Acesso em: 30 jun. 2021. Citado na página 16.

FREEMAN, S.; PRYCE, N. *Growing Object-Oriented Software, Guided by Tests*. 1. ed. [S.l.]: Addison-Wesley Professional, 2009. 358 p. Citado na página 15.

GIL, A. C. *Como elebaorar prjetos de pesquisa*. 4. ed. São Paulo: Editora Atlas, 2002. 175 p. Citado na página 30.

GRAPHQL. *Introduction to GraphQL*. 2021. Disponível em: https://graphql.org/learn/. Acesso em: 30 jun. 2021. Citado na página 25.

HILTON, R. *Quantitatively Evaluating Test-Driven Development by Applying Object-Oriented Quality Metrics to Open Source Projects*. Dissertação (Mestrado) — Department of Computer Information Sciences, Regis University, 2009. Disponível em: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.330.1817&rep=rep1&type=pdf. Acesso em: 30 jun. 2021. Citado na página 16.

HOED, R. van den *et al.* Data analysis on the public charge infrastructure in the city of amsterdam. In: *2013 World Electric Vehicle Symposium and Exhibition (EVS27)*. Barcelona: [s.n.], 2013. p. 1–10. Citado na página 10.

HU, X. *et al.* The potential impacts of electric vehicles on urban air quality in shanghai city. *Sustainability*, MDPI, v. 13, n. 2, p. 1–12, jan. 2021. Citado na página 10.

HUNT, J. *Advanced guide to Python 3 programming*. 1. ed. Switzerland: Springer, 2019. 494 p. Citado na página 17.

IEA. *Global EV Outlook 2021*. 2021. Disponível em: https://iea.blob.core.windows.net/assets/ed5f4484-f556-4110-8c5c-4ede8bcba637/GlobalEVOutlook2021.pdf. Acesso em: 14 jun 2021. Citado na página 9.

JACOBSON, D.; BRAIL, G.; WOODS, D. *APIs: A strategy guide*. 1. ed. Sebastopol: O'Reilly Media, 2012. 148 p. Citado na página 23.

KHANAM, Z.; AHSAN, M. N. Evaluating the effectiveness of test driven development: Advantages and pitfalls. *International Journal of Applied Engineering Research*, Research India Publications, v. 12, n. 18, p. 7705–7716, 2017. Citado na página 16.

KUMAR, S.; BANSAL, S. Comparative study of test driven development with traditional techniques. *International Journal of Soft Computing and Engineerin*, Blue Eyes Intelligence Engineering Sciences Publication, v. 3, n. 1, p. 352–360, mar. 2013. Citado na página 12.

LEE, J.; PARK, G. L. Electric vehicle charger management system for interoperable charging facilities. *Jurnal Teknologi*, Penerbit UTM Press, v. 78, p. 117–121, jan. 2016. Citado na página 10.

LIU, J. Research on electric vehicle fast charging station billing and settlement system. In: *2nd IEEE International Conference on Intelligent Transportation Engineering (ICITE)*. Singapore: [s.n.], 2017. p. 233–226. Citado na página 10.

MASSÉ, M. *REST API Design Rulebook*. 1. ed. Sebastopol: O'Reilly Media, 2012. 114 p. Citado 2 vezes nas páginas 20 e 21.

MOLINA, A. *Crafting Test-Driven Software with Python*. 1. ed. Birmingham: Packt Publishing, 2021. 323 p. Citado na página 17.

MOZILLA. *Métodos de requisição HTTP*. 2021. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods. Acesso em: 30 jun. 2021. Citado na página 22.

MUNIR, H.; PETERSEN, K.; MOAYYED, M. Considering rigor and relevance when evaluating test driven development: A systematic review. *Information and Software Technology*, Elsevier, v. 56, n. 4, p. 375–394, 2014. Citado na página 13.

NELSON, J. C.; SMITHE, S.; TIL, J. van der. *Testes de integração no ASP.NET Core*. 2021. Disponível em: https://docs.microsoft.com/pt-br/aspnet/core/test/integration-tests?view=aspnetcore-5.0. Acesso em: 30 jun. 2021. Citado na página 16.

OLIVEIRA, B. *pytest Quick Start Guide*. 1. ed. Birmingham: Packt Publishing, 2018. 162 p. Citado na página 17.

OMG. *Unified Modeling Language*. [S.l.], 2015. Disponível em: https://www.omg.org/spec/UML/2.5/PDF. Acesso em: 30 jun. 2021. Citado na página 27.

OSHEROVE, R. *The Art of Unit Testing*. 2. ed. [S.l.]: Manning Books, 2014. 294 p. Citado na página 15.

PAIK, H. *et al. Web Service Implementation and Composition Techniques*. 1. ed. Cham: Springer, 2017. 264 p. Citado 5 vezes nas páginas 17, 18, 19, 20 e 21.

PORCELLO, E.; BANKS, A. *Learning GraphQL*. 1. ed. Sebastopol: O'Reilly Media, 2018. 299 p. Citado na página 25.

PRIKLADNICKI, R. *Problemas, desafios e abordagens do processo de desenvolvimento de software*. 2004. Disponível em: https://www.inf.pucrs.br/munddos/docs/TI1.pdf. Acesso em: 16 jun 2021. Citado na página 9.

REDHAT. *GraphQL - O que é e para que serve?* 2021. Disponível em: https://www.redhat.com/pt-br/topics/api/what-is-graphql. Acesso em: 30 jun. 2021. Citado na página 26.

REDHAT. *Interface de Programação de aplicações: o que é api?* 2021. Disponível em: https://www.redhat.com/pt-br/topics/api/what-are-application-programming-interfaces. Acesso em: 16 jun 2021. Citado na página 9.

REESE, J. *Melhores práticas de teste de unidade com .NET Core e .NET Standard*. 2018. Disponível em: https://docs.microsoft.com/pt-br/dotnet/core/testing/unit-testing-best-practices. Acesso em: 30 jun. 2021. Citado na página 15.

REHKOPF, M. *Automated software testing*. 2021. Disponível em: https://www.atlassian.com/continuous-delivery/software-testing/automated-testing. Acesso em: 03 out. 2021. Citado na página 17.

RICHARDSON, L.; RUBY, S. *RESTful Web Services*. 1. ed. Sebastopol: O'Reilly Media, 2007. 440 p. Citado na página 24.

RITCHIE, H.; ROSER, M. Fossil fuels. *Our World in Data*, 2020. Disponível em: https://ourworldindata.org/energy. Acesso em: 16 jun 2021. Citado na página 10.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. *The Unified Modeling Language Reference Manual*. 1. ed. Boston: Addison-Wesley, 1999. 568 p. Citado 4 vezes nas páginas 26, 27, 28 e 29.

SEIDL, M. *et al. UML @ Classroom*. 1. ed. Heidelberg: Springer, 2014. 215 p. Citado na página 27.

SIERZCHULA, W.; BAKKER, S.; WEE, K. M. B. van. The influence of financial incentives and other socio-economic factors on electric vehicle adoption. *Energy Policy*, Elsevier, v. 68, p. 183–194, 2014. Citado na página 10.

SILVA, E. L. da; MENEZES, E. M. *Metodologia da Pesquisa e elaboração de Dissertação*. 4. ed. Florianópolis: UFSC, 2005. 138 p. Citado na página 30.

SISTEMA DE ESTIMATIVAS DE EMISSõES E REMOçõES DE GASES DE EFEITO ESTUFA. *Análise das emissões brasileiras de gases de efeito estufa e suas implicações para as metas de clima do Brasil 1970-2019*. 2020. Disponível em: https://seeg-br.s3.amazonaws.com/Documentos20Analiticos/SEEG_8/SEEG8_DOC_ANALITICO_SINTESE_1990-2019.pdf. Acesso em: 30 jun 2021. Citado na página 10.

SOARES, L. *How to write a test using TDD?* 2020. Disponível em: https://medium.com/codex/how-to-write-a-test-using-tdd-b2828788d7ea. Acesso em: 30 jun. 2021. Citado na página 14.

TARLINDER, A. *Developer Testing: Building Quality into Software*. 1. ed. [S.l.]: Addison-Wesley Professional, 2016. 352 p. Citado na página 14.

TIDWELL, D.; SNELL, J.; KULCHENKO, P. *Programming Web Services with SOAP*. 1. ed. Sebastopol: O'Reilly Media, 2001. 225 p. Citado na página 18.

VOCKE, H. *The Practical Test Pyramid*. 2018. Disponível em: https://martinfowler.com/articles/practical-test-pyramid.html#IntegrationTests. Acesso em: 30 jun. 2021. Citado na página 15.

WALKER, A. *What are Web Services? Architecture, Types, Example*. 2021. Disponível em: https://www.guru99.com/web-service-architecture.html. Acesso em: 30 jun. 2021. Citado na página 18.

WEBBER, J.; PARASTATIDIS, S.; ROBINSON, I. *REST in Practice*. 1. ed. Sebastopol: O'Reilly Media, 2010. 448 p. Citado 3 vezes nas páginas 20, 21 e 24.

WIEGERS, K.; BEATTY, J. *Software Requirement*. 3. ed. Redmond: Microsoft Press, 2013. 673 p. Citado na página 28.

WIERUCH, R. *The Road to GraphQL*. 1. ed. [S.l.]: Leanpub, 2019. 355 p. Citado na página 25.

WINTERS, T.; MANSHRECK, T.; WRIGHT, H. *Software Engineering at Google*. 1. ed. [S.l.]: O'Reilly Media, 2020. 602 p. Citado na página 15.

YUNUS, K.; PARRA, H. Z. D. L.; REZA, M. Distribution grid impact of plug-in electric vehicles charging at fast charging stations using stochastic charging models. In: *Proceedings of the 2011 14th European Conference on Power Electronics and Applications*. [S.l.: s.n.], 2011. p. 1–11. Citado na página 9.

ZHANG, Y. *et al.* Electric vehicle charging fault monitoring and warning method based on battery model. *World Electric Vehicle Journal*, MDPI, v. 12, p. 1–15, jan. 2021. Citado na página 10.

Z.SHENG, Q. *et al.* Web services composition: A decade's overview. *Information Sciences*, Elsevier, v. 280, n. 1, p. 218–238, 2014. Citado 2 vezes nas páginas 17 e 18.