UNIVERSIDADE ESTADUAL DA PARAÍBA

CAMPUS I - CAMPINA GRANDE

CENTRO DE CIÊNCIAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

CURSO DE GRADUAÇÃO EM COMPUTAÇÃO

RENAN GUSTAVO DE ALBUQUERQUE MELO

DESENVOLVIMENTO BACK-END PARA UM MODELO DE APLICAÇÃO DE FRETES

CAMPINA GRANDE

2023

RENAN GUSTAVO DE ALBUQUERQUE MELO

DESENVOLVIMENTO BACK-END PARA UM MODELO DE APLICAÇÃO DE FRETES

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

Área de concentração:
Desenvolvimento de Sistemas de Gestão de Fretes.

Orientadora: Profa. DSc Kátia Elizabete Galdino

CAMPINA GRANDE

2023

M528d Melo, Renan Gustavo de Albuquerque.
Desenvolvimento *back-end* para um modelo de aplicação de fretes [manuscrito] / Renan Gustavo de Albuquerque Melo. - 2023.
46 p.

Digitado.
Trabalho de Conclusão de Curso (Graduação em Computação) - Universidade Estadual da Paraíba, Centro de Ciências e Tecnologia, 2023.
"Orientação : Profa. Dra. Kátia Elizabete Galdino, Departamento de Computação - CCT. "

1. Application Programming Interface - API. 2. Software as a Service - SaaS. 3. Desenvolvedores. I. Título

21. ed. CDD 005.3

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

RENAN GUSTAVO DE ALBUQUERQUE MELO

DESENVOLVIMENTO BACK-END PARA UM MODELO DE APLICAÇÃO DE FRETES

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

Área de concentração: Desenvolvimento de Sistemas de Gestão de Fretes.

Aprovada em 07/07/2023.

BANCA EXAMINADORA

Profa. DSc Kátia Elizabete Galdino (DC - UEPB)

Orientador(a)

Prof. DSc Frederico Moreira Bublitz (DC - UEPB)

Examinador(a)

MSc Túlio Henriques Costa (NUTES / UEPB)

Examinador(a)

RESUMO

Considerando o grande número de desenvolvedores iniciando suas carreiras no desenvolvimento *back-end*, este trabalho tem como objetivo demonstrar o funcionamento de uma API criada para uma aplicação de fretes com base arquitetural SaaS. Essa API é compatível com dispositivos desktop e mobile. A motivação para esse estudo parte da busca do desenvolvimento de uma aplicação que apresentasse uma boa lógica, funcionamento e comunicação. Para alcançar esse objetivo, foi necessário compreender conceitos de desenvolvimento *back-end* utilizando as práticas discutidas pelos autores Mario Casciaro e Luciano Mammino e conteúdos adquiridos na documentação das tecnologias utilizadas no projeto. Por essa razão, este trabalho caracteriza-se por ser de natureza bibliográfica, que se baseia nos modelos desenvolvidos conforme os estudos mencionados.

Palavras-Chave: desenvolvimento *back-end*; SaaS; API.

ABSTRACT

Considering the large number of developers starting their careers in back-end development, this work aims to demonstrate the operation of an API created for a freight application with a SaaS architectural base. This API is compatible with desktop and mobile devices. The motivation for this study comes from the search for the development of an application that presented good logic, operation and communication. To achieve this goal, it was necessary to understand back-end development concepts using the practices discussed by authors Mario Casciaro and Luciano Mammino and content acquired in the documentation of the technologies used in the project. For this reason, this work is characterized as being of a bibliographic nature, based on models developed according to the studies mentioned.

Keywords: back-end development; SaaS; API.

LISTA DE ILUSTRAÇÕES

LISTA DE TRECHOS DE CÓDIGO

LISTA DE ABREVIATURAS E SIGLAS

API - *Application Programming Interface - Interface de Programação de Aplicação*

CoC - *Convention Over Configuration - Convenção sobre Configuração*

HTTP - *Hypertext Transfer Protocol - Protocolo de Transferência de Hipertexto*

IDE - *Integrated Development Environment - Ambiente de Desenvolvimento Integrado*

MVC - *Model, view, controller* - Modelo, visualização, controlador.

Nest - *NestJS*

NUTES - *Núcleo De Tecnologias Estratégicas em Saúde*

Restful - *Representational State Transfer - Transferência de Estado Representacional*

SaaS - *Software as a Service - Software como Serviço*

SGBD - *Sistema Gerenciador de Banco de Dados*

SOAP - *Simple Object Access Protocol - Protocolo Simples de Acesso a Objetos*

SQL - *Structured Query Language - Linguagem de Consulta Estruturada*

SUMÁRIO

1 INTRODUÇÃO

A sociedade contemporânea está passando por mudanças contínuas na ciência, tecnologia, cultura e sociedade que afetam a forma como as pessoas interagem, trabalham, aprendem e se divertem. Nessa situação, as pessoas esperam que as tecnologias sejam otimizadas, ou seja, que forneçam soluções rápidas, eficazes, acessíveis e soluções personalizadas para seus problemas e necessidades.

Um exemplo disso, segundo a pesquisa sobre o uso das Tecnologias de Informação e Comunicação nos domicílios brasileiros (TIC Domicílios 2020), é o uso das tecnologias digitais durante a pandemia de covid-19, que intensificou a conectividade nos domicílios brasileiros, passando de 71% em 2019 para 83% em 2020. Além disso, a internet e os dispositivos móveis passaram a desempenhar um papel central para a continuidade de atividades empresariais, comerciais, educacionais, de saúde e de lazer.

O objetivo principal do presente estudo é desenvolver e obter conhecimento das técnicas e conceitos necessários para criar o *back-end* (serviço responsável pelas operações do sistema do lado do servidor, no qual o usuário não tem acesso) de um sistema de fretes que usa a base arquitetônica de um *software* como serviço, ressaltando a importância de ter uma boa comunicação do cliente com o servidor, e ter um *back-end* escalável, seguro e robusto para que sua aplicação possa lidar com o crescimento do tráfego e das necessidades de negócio para que tudo funcione de maneira eficiente.

A decisão de implementar um sistema de fretes foi motivada por uma ideia concebida durante o período de estágio profissional do autor no laboratório SenseLab, pertencente ao NUTES. A partir desse momento, as ideias relacionadas ao projeto ganharam impulso e se tornaram o foco principal da pesquisa.

A escolha do tema está de acordo com a percepção do autor de que, diante da vivência em um mundo digital, é importante aprofundar-se na área temática em estudo. Essa percepção é influenciada pela participação em projetos acadêmicos e desenvolvimentos de projetos pessoais com o objetivo de profissionalização, que traz questionamentos e desejo de informações sobre o trabalho atual.

Este trabalho está dividido em duas fases principais: a primeira destaca os fundamentos teóricos envolvendo os conceitos de *back-end* e arquitetura SaaS, e a segunda foca nos resultados do estudo por meio do código produzido de acordo com os princípios das metodologias.

2 OBJETIVOS

2.1 Objetivo Geral

Abordar e destacar a importância das boas práticas, lógica, funcionamento e comunicação no desenvolvimento de um *back-end* para o sistema de fretes chamado 6Leva.

2.2 Objetivos Específicos

- Desenvolver parte do *back-end* da aplicação;
- Fazer uma abordagem sobre boas práticas, lógica, funcionamento e comunicação;

3 FUNDAMENTAÇÃO TEÓRICA

Todo o material de base do trabalho teórico será abordado nesta sessão, ou todas as ênfases necessárias para realizar este projeto. Como resultado, esta seção está dividida em três subseções principais: *software* como serviço (subseção 4.1), *back-end* (subseção 4.2) e resultados (seção 5).

3.1 Software como Serviço

Software como serviço é um modelo de negócios que permite aos clientes acessar e usar *software* baseado em nuvem em troca de uma taxa de assinatura ou pagamento por uso. Nesse modelo de negócios, o provedor de SaaS é responsável pelo desenvolvimento, manutenção, segurança e atualização de *software*, enquanto os clientes podem ganhar com maior flexibilidade, economia de custos e escalabilidade. Alguns exemplos de SaaS incluem: Salesforce, Netflix, Spotify e Gmail.

Segundo uma pesquisa da (UserGuiding 2022), devido a pandemia do COVID-19, algumas empresas passaram a adotar o uso do SaaS pois a tecnologia de nuvem seria a solução para as novas necessidades que surgiram com a pandemia, como o trabalho remoto e as interações remotas. Além disso, há dados estatísticos sobre como as empresas de SaaS reagiram aos efeitos do COVID-19 em relação aos seus preços:

- Quase 30% das empresas SaaS ofereceram funcionalidades adicionais para seus clientes.
- 33% da maioria das empresas de SaaS de médio e grande porte ofereceram serviços adicionais.
- Cerca de 30% das empresas ofereceram redução de preços.
- 40% das empresas de SaaS estão pensando em oferecer novos modelos de preços quando os negócios se recuperarem.
- 50% das empresas de SaaS ainda dependem de preços baseados no usuário.
- Mais de 50.000 fornecedores de SaaS oferecem descontos superiores a 30%.
- 40% estão planejando oferecer incentivos de compra.

3.2 Back-end

Desenvolvimento *back-end* é o termo usado para se referir à parte da programação que lida com a lógica, o funcionamento e a comunicação entre o banco de dados e a interface visual de uma aplicação mobile ou o navegador de uma aplicação web. O desenvolvedor *back-end* é responsável por criar, testar e manter o código que garante a segurança, a performance e a atualização dos dados que são exibidos na interface gráfica do usuário (*front-end*). O desenvolvimento *back-end* é essencial para o funcionamento de qualquer aplicação, pois é ele que dá suporte e estrutura para as interações do usuário com a máquina.

3.2.1 API RESTful

Uma API RESTful é uma interface que permite a comunicação entre diferentes sistemas de *software* pela internet, seguindo os princípios da arquitetura REST. As APIs RESTful são mais simples, rápidas e flexíveis do que outros tipos de APIs, como SOAP.

Segundo um estudo realizado por Usman Riaz, Samir Hussain e Hemil Patel (Springer Link 2021), para remover a alta latência, reduzir o tráfego de rede e os atrasos de processamento causados pelo SOAP, o REST foi introduzido para superar todos esses problemas. Além disso, 70% dos sites usam a arquitetura REST, pois muitos consideram o SOAP desatualizado.

A comunicação da interface do usuário (*front-end*) e a API é dada através de requisições HTTP que utilizam verbos como *GET*, *POST*, *PUT*, *DELETE*, entre outros para indicar a ação a ser executada para um dado recurso. Os *statuscode* (códigos de status das respostas) HTTP indicam se uma requisição HTTP foi corretamente concluída. As respostas são agrupadas em cinco classes: Respostas de informação (100-199), Respostas de sucesso (200-299), Redirecionamentos (300-399), Erros do cliente (400-499) e Erros do servidor (500-599).

3.2.2 Frameworks

Frameworks para desenvolvimento *back-end* são ferramentas que facilitam e aceleram a criação da arquitetura de um site, aplicativo ou *software*. Eles fornecem

bibliotecas de modelos, módulos e funcionalidades que podem ser usados e personalizados pelos desenvolvedores, sem a necessidade de codificar tudo do zero. Existem diversos *frameworks* para *back-end*, cada um com suas vantagens, desvantagens e características específicas.

Exemplos de frameworks populares: Django que é *open source*, baseado na linguagem de programação Python e segue o padrão MVC; Spring que é *open source* para a plataforma Java e segue os padrões de inversão de controle e injeção de dependências; e NestJS que utiliza por padrão a biblioteca Express como base, utiliza a linguagem Typescript e segue o padrão MVC.

3.2.3 Banco de dados

Um banco de dados é uma coleção de informações estruturadas ou dados estruturados que normalmente são armazenados eletronicamente em um sistema de computador. Um banco de dados normalmente é gerenciado por um SGBD, que é um *software* com os recursos para manipular informações do banco de dados e se comunicar com os usuários.

Segundo Korth (Devmedia 2006), um banco de dados é uma coleção de dados inter-relacionados, representando informações sobre um domínio específico, ou seja, sempre que for possível agrupar informações que se relacionam e tratam de um mesmo assunto, é possível afirmar que se tem banco de dados.

3.3 Tecnologias que foram utilizadas no projeto

As tecnologias citadas serviram de alicerce para a base do desenvolvimento do projeto, sendo necessária a utilização de linguagens de programação, *framework* e bancos de dados. Estes serão abordados a seguir.

3.3.1 JavaScript

Segundo sua documentação oficial, JavaScript é uma linguagem de programação leve, interpretada ou compilada *just-in-time* (compilada em tempo de execução) com funções de primeira classe (suas funções são tratadas como qualquer outra variável). Embora seja mais conhecida como a linguagem de *script* para páginas da Web, muitos ambientes que não são navegadores também a

utilizam, como Node.js, Apache CouchDB e Adobe Acrobat. JavaScript é uma linguagem dinâmica baseada em protótipo, multiparadigma, de *thread* único, que suporta estilos orientados a objetos, imperativos e declarativos (por exemplo, programação funcional).

3.3.2 TypeScript

De acordo com sua documentação oficial, TypeScript é uma linguagem que é um superconjunto de JavaScript. A sintaxe refere-se à maneira como escrevemos o texto para formar um programa. Por exemplo, este código da figura 1 tem um erro de sintaxe porque está faltando um parênteses:

Figura 1: Exemplo de código TypeScript

Fonte: Typescritplang

O TypeScript não considera nenhum código JavaScript como um erro por causa de sua sintaxe. Isso significa que podemos pegar qualquer código JavaScript funcional e colocá-lo em um arquivo TypeScript sem nos preocuparmos exatamente como ele foi escrito. No entanto, TypeScript é um superconjunto tipado, o que significa que adiciona regras sobre como diferentes tipos de valores podem ser usados, dessa forma, há mais garantia de segurança durante a fase de desenvolvimento de software graças principalmente à checagem de tipos.

3.3.3 Node.js

Na documentação oficial do Node.js é dito que como um *runtime* JavaScript assíncrono orientado a eventos, o Node.js foi projetado para criar aplicativos de rede escalonáveis. No exemplo de "*hello world*" na figura 2, muitas conexões podem ser tratadas simultaneamente. A cada conexão, o retorno de chamada é acionado, mas se não houver trabalho a ser feito, o Node.js será suspenso.

Figura 2: Exemplo de código Node.js

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Fonte: nodejs.org

Isso contrasta com o modelo de simultaneidade mais comum de hoje, no qual os encadeamentos do sistema operacional são empregados. A rede baseada em thread é relativamente ineficiente e muito difícil de usar. Além disso, os usuários do Node.js estão livres da preocupação de travar o processo, já que não há bloqueios. Quase nenhuma função no Node.js realiza I/O diretamente, portanto, o processo nunca é bloqueado, exceto quando o I/O é executado usando métodos síncronos da biblioteca padrão do Node.js. Isso acontece devido ao *event-loop* (loop de eventos) que se trata de um loop executado enquanto sua aplicação Node.js estiver funcionando.

3.3.4 NestJS

De acordo com sua documentação, Nest é uma estrutura para criar aplicativos Node.js eficientes e escaláveis do lado do servidor. Ele usa JavaScript progressivo (de maneira gradual e incremental), é construído e compatível com TypeScript (mas ainda permite que os desenvolvedores codifiquem em JavaScript puro) e combina elementos de Programação Orientada a Objetos, Programação Funcional e Programação Reativa Funcional.

Um dos diferenciais do Nest em relação a outros frameworks é sua base nos fundamentos de design e arquitetura do Angular, o que significa que ele foi projetado sob um conceito de módulos. Alguns dos benefícios do Nest incluem: CoC (responsável por fornecer a estrutura de pastas do projeto para separar as lógicas

do sistema), arquitetura escalável, fácil integração com diversos tipos de bancos de dados e suporte a microsserviços (FullCycle 2022).

3.3.5 PostgreSQL

O PostgreSQL é um poderoso sistema de banco de dados objeto-relacional de código aberto com mais de 35 anos de desenvolvimento ativo que lhe rendeu uma forte reputação de confiabilidade, robustez de recursos e desempenho.

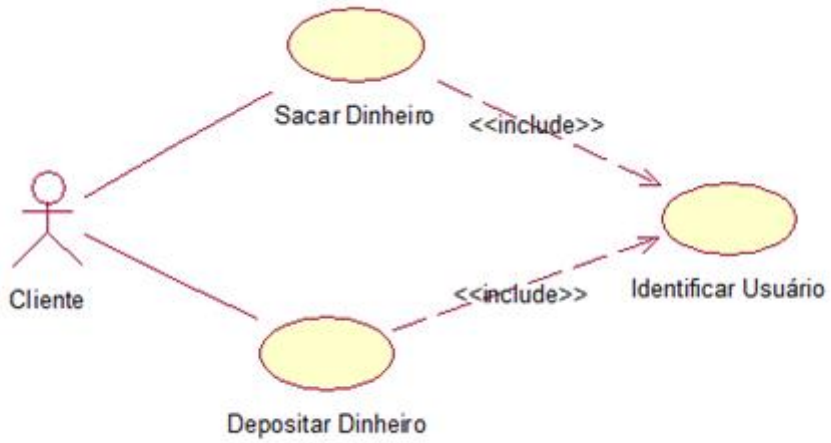
3.4 Arquitetura de software

A arquitetura de software tem como objetivo definir qual tipo de estrutura que faz mais sentido para determinado projeto, potencializando a experiência do usuário, garantindo maior fluidez no uso das soluções (Datum 2023).

3.4.1 Caso de uso

De acordo com um artigo da (Devmedia 2008), caso de uso é um artefato muito utilizado que nos permite dirigir todo o desenvolvimento de software. Com ele, é possível rastrear os impactos nos requisitos de software e nos demais artefatos que foram criados a partir dele. Na figura 3 é possível observar um exemplo de caso de uso sobre movimentação bancária, onde há um ator chamado de cliente que realiza operações de sacar dinheiro e depositar dinheiro, mas é necessário que o usuário seja identificado antes de realizar quaisquer operações, por isso o uso do termo <<include>>.

Figura 3: Caso de uso de movimentação bancária

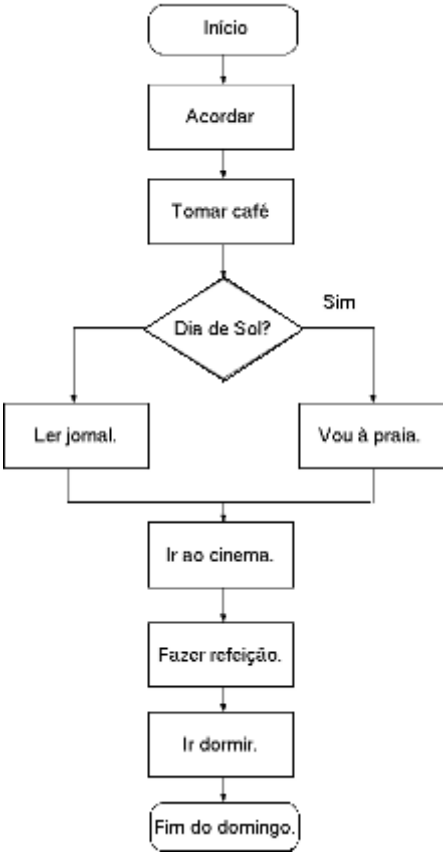


Fonte: Devmedia

3.4.2 Fluxograma

Um fluxograma é uma representação visual de um processo ou sistema, apresentando as etapas, os fluxos e as decisões envolvidas. Ele utiliza símbolos, formas geométricas e setas para ilustrar as diferentes partes do processo e suas interações (Flowup 2023). É possível observar que o fluxograma é adaptável para quaisquer situações na figura 4.

Figura 4: Exemplo de fluxograma de um dia de domingo



Fonte: Devmedia

3.5 Programas que foram utilizados no projeto

Durante o projeto foi necessário a utilização de alguns programas para obter uma melhor performance no desenvolvimento dos códigos e nos testes. Estes serão abordados nesta seção.

3.5.1 WebStorm

De acordo com a JetBrains, WebStorm é um ambiente de desenvolvimento integrado para JavaScript e tecnologias relacionadas. Assim como outros IDEs da JetBrains, ele torna sua experiência de desenvolvimento mais agradável, automatizando o trabalho de rotina e ajudando você a lidar com tarefas complexas com facilidade.

3.5.2 DataGrip

Ainda sobre de um produto da JetBrains, DataGrip se trata de uma nova IDE de banco de dados adaptado para atender às necessidades específicas de desenvolvedores SQL profissionais.

3.5.3 Postman

De acordo com seu site oficial, Postman é uma plataforma de API para criar e usar APIs. O Postman simplifica cada etapa do ciclo de vida da API e agiliza a colaboração para que você possa criar APIs melhores com mais rapidez.

O Postman é responsável pelos testes de requisições HTTP no projeto, através dos verbos para que seja possível observar o *statuscode* e o retorno da requisição.

4 METODOLOGIA APLICADA

Inicialmente, a metodologia utilizada para alcançar o resultado atual baseou-se em um estudo de conceitos e técnicas sobre o desenvolvimento *back-end* na perspectiva de dois autores. Dessa forma, o trabalho está embasado nos autores Mario Casciaro e Luciano Mammino, a partir da obra “Node.js *Design Patterns*, 3ed”. A justificativa para a escolha desses autores parte do fato de que eles são especialistas no assunto e têm experiência prática no desenvolvimento de aplicativos usando Node.js. Os autores têm um histórico comprovado de conhecimento e habilidades em programação, com um amplo domínio de várias linguagens de programação, incluindo JavaScript. A terceira edição do livro indica que ele foi bem recebido e atualizado para refletir as últimas tendências e práticas no desenvolvimento com Node.js.

Como resultado, foi utilizada uma metodologia de pesquisa bibliográfica baseada em documentos para investigar as tecnologias utilizadas nos produtos em questão. O foco da pesquisa foi nas informações de lógica, funcionamento e comunicação, empregando técnicas de desenvolvimento *back-end*. Além disso, foi realizada uma análise interpretativa das informações coletadas a partir das fontes consultadas.

5 RESULTADOS

Entender a necessidade do usuário em relação ao sistema projetado é necessário para colocar em prática o conhecimento sobre os estudos e tecnologias citados. Esse entendimento pode ser alcançado examinando os conceitos e tecnologias discutidos anteriormente, em conjunto com alguns dos principais requisitos funcionais que foram considerados durante uma das etapas de planejamento de software e são mostrados no Quadro 1.

Quadro 1: Principais Requisitos Funcionais do Sistema 6Leva.

Requisito Funcional	Descrição	Grau de Importância
RF01	O sistema deve permitir que o usuário faça um cadastro	Alto
RF02	O sistema deve permitir que o usuário entre em sua conta	Alto
RF03	O sistema deve permitir que o usuário rastreie sua encomenda	Alto
RF04	O sistema deve permitir que o usuário realize uma simulação de valores para os fretes	Alto

Fonte: Rafael Araujo, 2022.

Nesta seção, é apresentado o resultado dos estudos juntamente com casos de uso, fluxogramas e o resultado de requisições HTTP para entender o funcionamento dos requisitos definidos. Vale ressaltar que no final do trabalho encontram-se Apêndices com trechos de código de cada requisito.

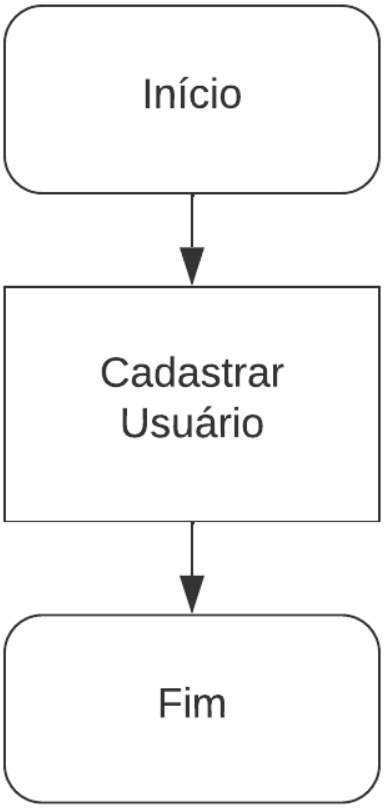
5.1 Cadastro de Usuário

No requisito de cadastro de usuário é possível observar uma relação simples entre o ator e o caso de uso (figura 5) e também um fluxograma simples (figura 6). É válido observar que no retorno de sua requisição HTTP (figura 7) a senha é retornada criptografada, garantindo maior segurança para o usuário.

Figura 5: Diagrama de Caso de Uso

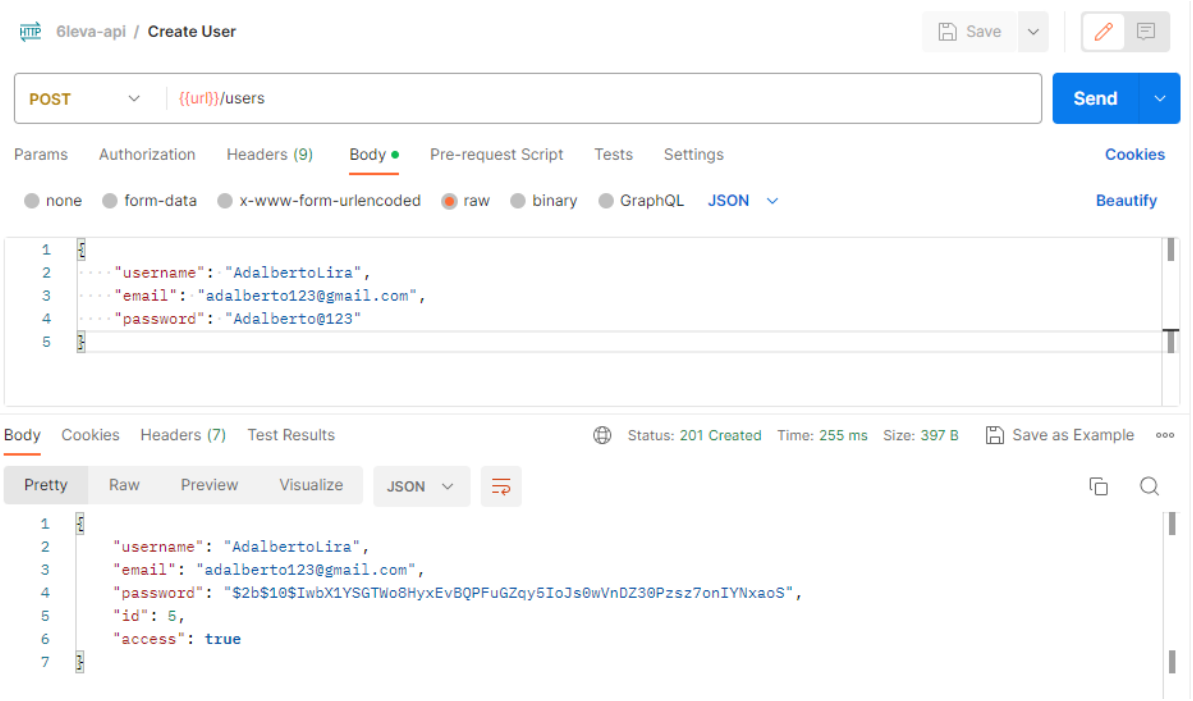
Fonte: Elaborado pelo autor, 2023.

Figura 6: Fluxograma



Fonte: Elaborado pelo autor, 2023.

Figura 7: Exemplo de *Request* HTTP no Postman



Fonte: Elaborado pelo autor, 2023.

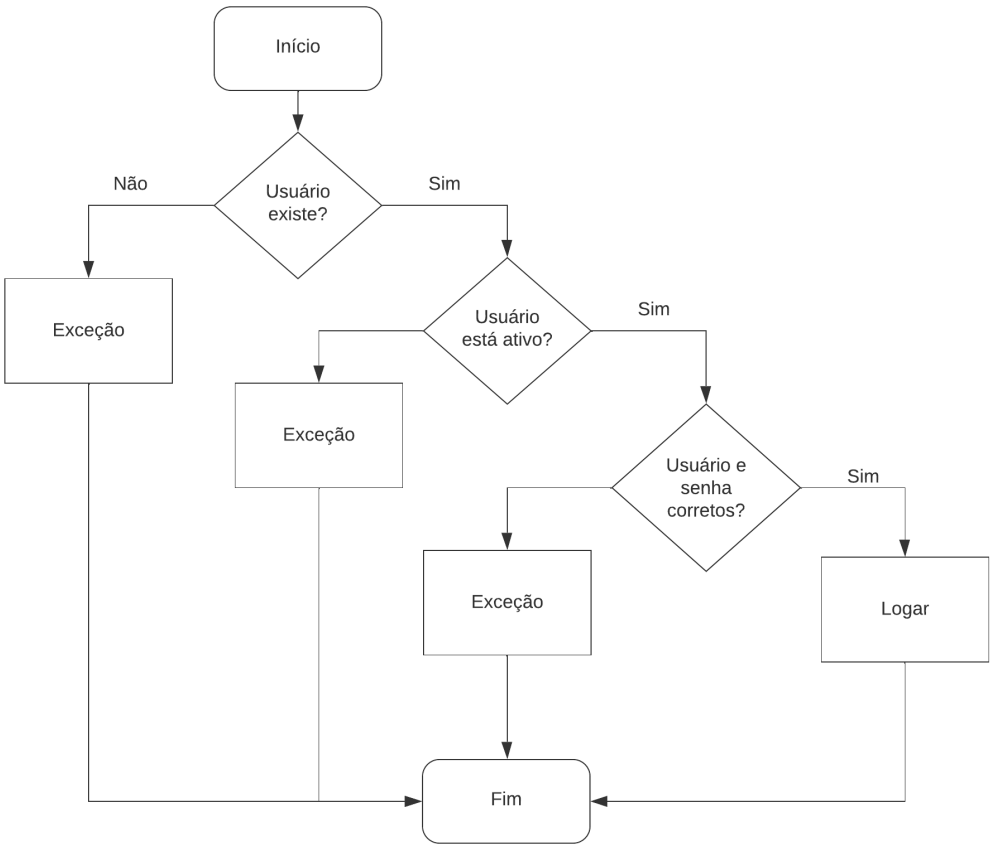
5.2 Login de Usuário

No requisito de login de usuário é possível observar uma relação simples entre o ator e o caso de uso (figura 8), porém um fluxograma mais complexo (figura 9), contendo estruturas de decisão para a continuidade do fluxo. É válido ressaltar que o retorno da sua requisição HTTP (figura 10) é um *accessToken* (token de acesso), responsável por garantir a autorização de rotas privadas do sistema para o usuário.

Figura 8: Diagrama de Caso de Uso

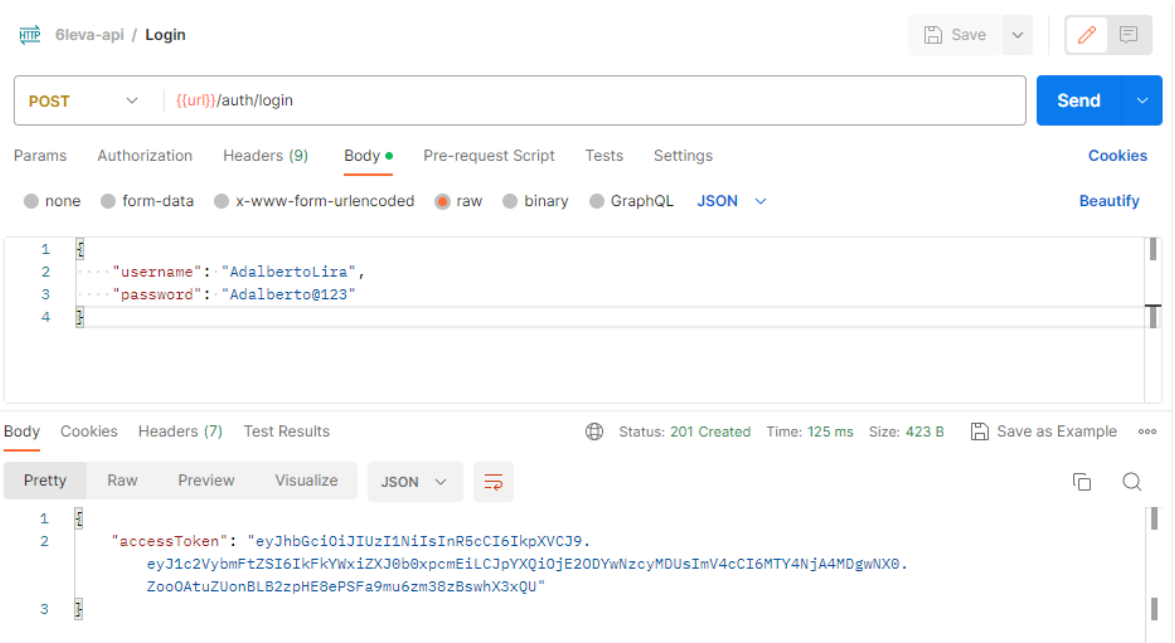
Fonte: Elaborado pelo autor, 2023.

Figura 9: Fluxograma



Fonte: Elaborado pelo autor, 2023.

Figura 10: Exemplo de *Request* HTTP no Postman

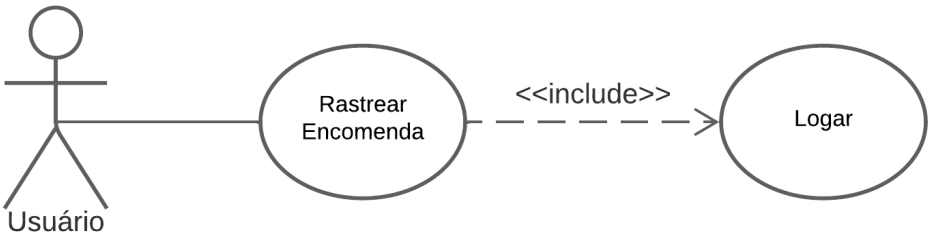


Fonte: Elaborado pelo autor, 2023.

5.3 Rastreio de Encomenda

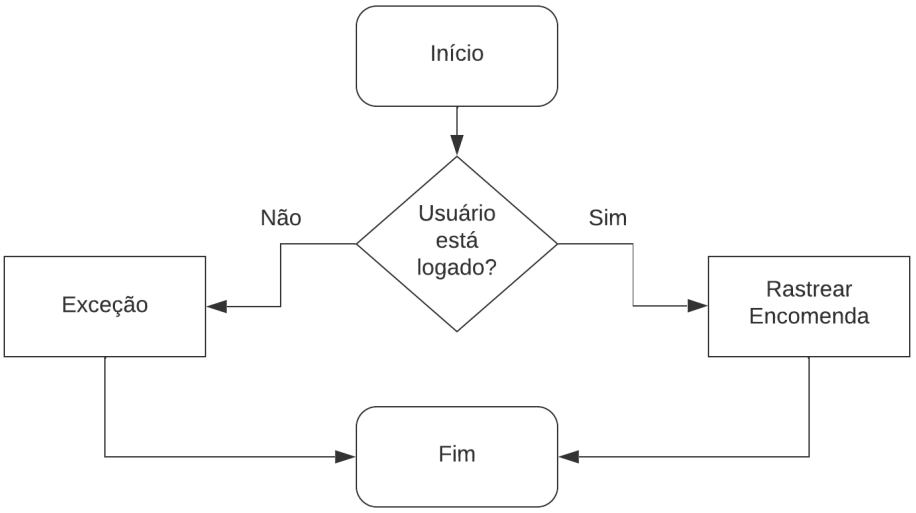
No requisito de rastreio de encomenda é possível observar uma relação simples entre o ator e o caso de uso (figura 11), mas com uma ação necessária obrigatoriamente de login do usuário, definida pela pela palavra chave <<*include*>>, seu fluxograma também contém estrutura de decisão para a continuidade do fluxo (figura 12). No retorno da sua requisição HTTP (figura 13) é possível observar que é retornada a lista de todas as encomendas atreladas ao usuário.

Figura 11: Diagrama de Caso de Uso



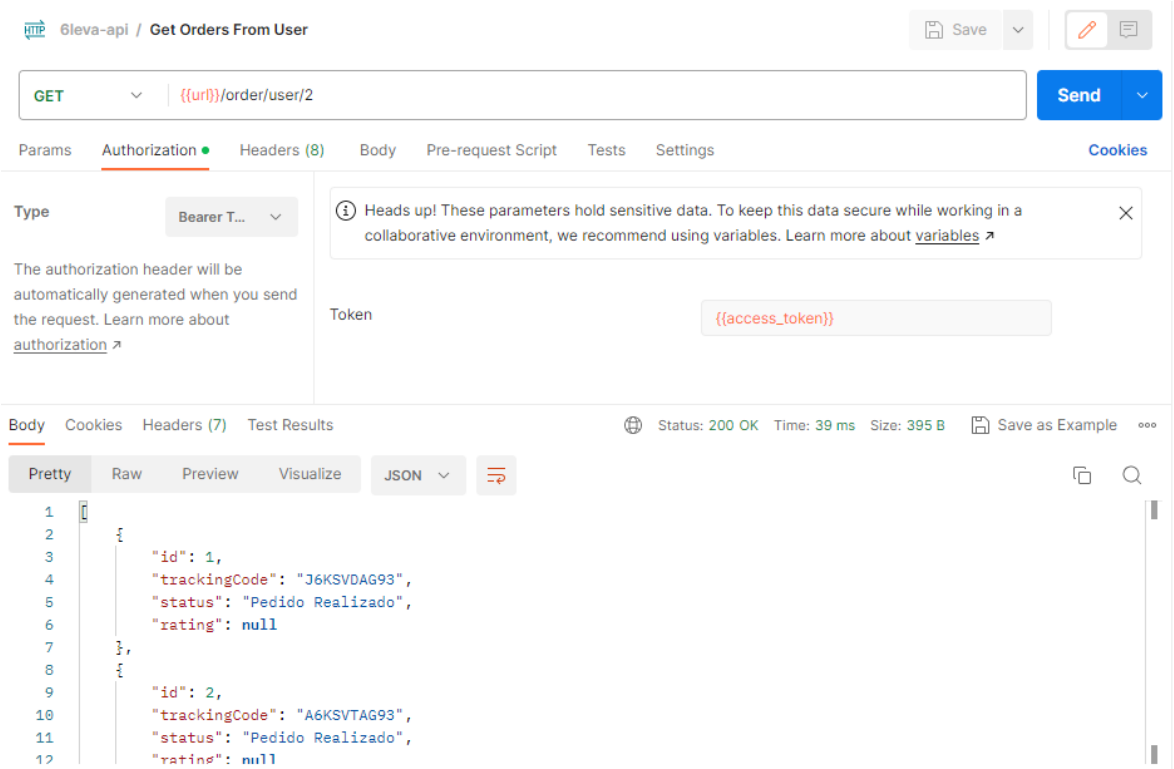
Fonte: Elaborado pelo autor, 2023.

Figura 12: Fluxograma



Fonte: Elaborado pelo autor, 2023.

Figura 13: Exemplo de *Request* HTTP no Postman



Fonte: Elaborado pelo autor, 2023.

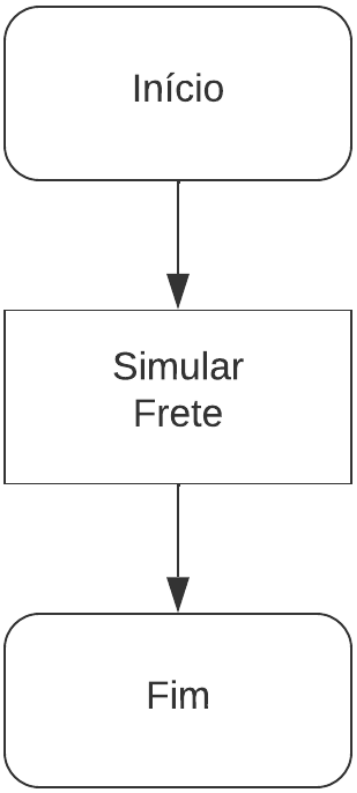
5.4 Simulador de Valores para Fretes

No requisito de simulador de valores para fretes é possível observar uma relação simples entre o ator e o caso de uso (figura 14) e também um fluxograma simples (figura 15). Observa-se também o retorno de sua requisição HTTP (figura 16) com o resultado baseado na fórmula genérica que foi utilizada para o cálculo da simulação.

Figura 14: Diagrama de Caso de Uso

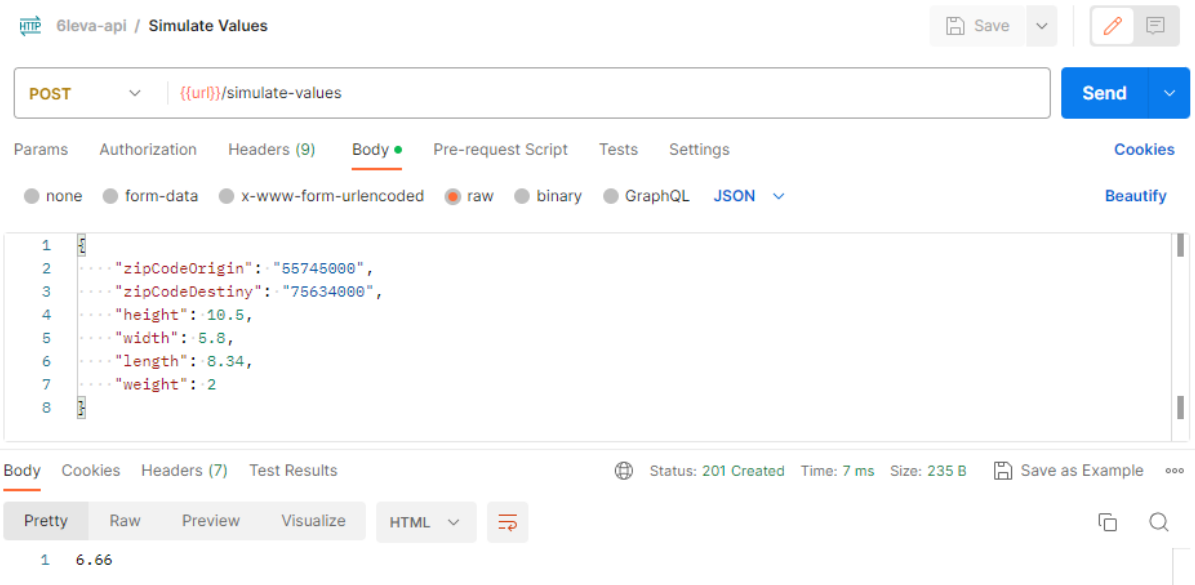
Fonte: Elaborado pelo autor, 2023.

Figura 15: Fluxograma



Fonte: Elaborado pelo autor, 2023.

Figura 16: Exemplo de *Request* HTTP no Postman



Fonte: Elaborado pelo autor, 2023.

6 CONSIDERAÇÕES FINAIS

Este trabalho aborda o *back-end* de um aplicativo de frete, destacando a importância das boas práticas, lógica, funcionalidade e comunicação durante a fase de desenvolvimento. Compreender esses aspectos é fundamental, segundo os autores Mario Casciaro e Luciano Mammino, para acompanhar o crescimento do mercado, competitividade e a entrada de novos desenvolvedores.

A aplicação desses conhecimentos na prática, juntamente com a definição de requisitos web e *mobile*, levou a realizações satisfatórias dentro dos parâmetros deste projeto. No entanto, dado que o projeto ainda está em seus estágios iniciais, há espaço para aumentar as pesquisas e esforços e aprimorar o desenvolvimento do código do sistema, levando em consideração que novos requisitos surgirão ao longo do projeto.

6.1 Trabalhos Futuros

Trabalhos futuros são necessários para garantir a continuidade, mesmo em estudos relacionados às fases que faltam na construção do *back- end* do aplicativo, devido às inúmeras etapas de desenvolvimento necessárias para transformar uma ideia em um *software* funcional de grande porte que atenda a certos padrões. Diante dessa premissa, os seguintes itens devem ser destacados:

- Documentação: A criação da documentação é uma etapa crucial do desenvolvimento que pode ser extensivamente explorada como resultado deste trabalho. A documentação, que é utilizada em todas as fases do desenvolvimento do *software*, pode garantir segurança e melhor entendimento da aplicação para os desenvolvedores atuais e possíveis desenvolvedores futuros;
- Requisitos restantes: Ainda há requisitos para serem desenvolvidos e compor ainda mais a aplicação tornando-a uma aplicação cada vez mais completa, neste trabalho foram apresentados e codificados apenas os principais requisitos do sistema.
- Codificação das telas (*front-end*): A modelagem inicial das principais telas foi desenvolvida no trabalho de Rafael Araújo em 2022, mas ainda é necessário a codificação das mesmas para que o sistema *front-end* funcione corretamente em conjunto com o *back-end*.

REFERÊNCIAS

AGÊNCIA BRASIL. **Estudo mostra que pandemia intensificou uso das tecnologias digitais.** Disponível em: https://agenciabrasil.ebc.com.br/geral/noticia/2021-11/estudo-mostra-que-pandemia-intensificou-uso-das-tecnologias-digitais/. Acesso em: out. 2022.

CASCIARO, M; MAMMINO, L. **Node.js Design Patterns.** 3. ed. Packt Publishing, jul. 2020.

DATUM. **Arquitetura de software: entenda o que é e quais os benefícios.** Disponível em: https://datumit.com/blog/arquitetura-de-software/

DEV MEDIA. **Conceitos Fundamentais de Banco de Dados.** Disponível em: https://www.devmedia.com.br/conceitos-fundamentais-de-banco-de-dados/1649.

DEV MEDIA. **Desenvolvimento de Software Dirigido por Caso de Uso.** Disponível em: https://www.devmedia.com.br/desenvolvimento-de-software-dirigido-por-caso-de-uso/9148

FLOWUP. **Fluxogramas: O Guia Completo sobre Flowchart.** Disponível em: https://www.flowup.me/blog/fluxogramas/

FULLCYCLE. **O grande diferencial do NestJS.** Disponível em: https://fullcycle.com.br/o-grande-diferencial-do-nestjs/

JETBRAINS. **DataGrip - Cross-platform database IDE.** Disponível em: https://www.jetbrains.com/datagrip/

JETBRAINS. **The Best JavaScript IDE - WebStorm by JetBrains.** Disponível em: https://www.jetbrains.com/webstorm/

MDN WEB DOCS. **JavaScript - MDN Web Docs.** Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript

MONTEIRO, R.A.S... **Desenvolvimento Front-End para um modelo de aplicação de fretes** / Rafael Araújo dos Santos Monteiro - 2022.

NESTJS. **Documentation | NestJS - A progressive Node.js framework.** Disponível em: https://docs.nestjs.com/

NODE.JS. **Documentation - Node.js.** Disponível em: https://nodejs.org/en/docs

POSTMAN API PLATFORM. **Postman API Platform.** Disponível em: https://www.postman.com/

RIAZ, U., HUSSAIN, S., & PATEL, H. **A Comparative Study of REST with SOAP.** In: Springer, 2021. Disponível em: https://link.springer.com/chapter/10.1007/978-3-030-82562-1_47

TOTVS DEVELOPERS. **O que é back-end e qual seu papel na programação?** Disponível em: https://www.totvs.com/blog/developers/back-end/

TYPESCRIPTLANG. **TypeScript: JavaScript With Syntax For Types.** Disponível em: https://www.typescriptlang.org/

USERGUIDING. **51 Estatísticas e tendências SaaS: Explorando o crescimento e a adoção de SaaS.** Disponível em: https://userguiding.com/pt-br/blog/estatisticas-e-tendencias-saas/. Acesso em: nov. 2022.

APÊNDICE A - TRECHOS DE CÓDIGO (CADASTRO DE USUÁRIO)

No trecho de código 1 é possível observar a entidade *User* estendida da classe *BaseEntity*, responsável por definir quais atributos e relações a entidade vai ter no sistema.

Trecho de Código 1: Cadastro de Usuário (user.entity.ts).

Código TypeScript

```typescript
@Entity('user')
export class User extends BaseEntity {
 @PrimaryGeneratedColumn()
 id: number;

 @Column({ name: 'username', type: 'varchar', length: 50, unique: true })
 username: string;

 @Column({ name: 'email', type: 'varchar', length: 50, unique: true })
 email: string;

 @Column({ name: 'password', type: 'varchar', length: 256 })
 password: string;

 @Column({ name: 'access', type: 'boolean', default: true })
 access: boolean;

 @OneToMany((type) => Order, (user) => User)
 orders: Order[];
}
```

Fonte: Elaborado pelo autor, 2023.

No trecho de código 2 é definido os atributos que o usuário do sistema precisa informar para a criação da entidade citada anteriormente, por *default*, o atributo *access* é definido como *true*.

Trecho de Código 2: Cadastro de Usuário (create-user.dto.ts).

Código TypeScript

```typescript
export class CreateUserDto {

 @ApiProperty()

 @IsNotEmpty()

 username: string;



 @ApiProperty()

 @IsNotEmpty()

 @IsEmail()

 email: string;



 @ApiProperty()

 @IsNotEmpty()

 @Matches(RegExHelper.password, { message: MessagesHelper.PASSWORD_VALID })

 password: string;



 @ApiProperty()

 @IsNotEmpty()

 @IsBoolean()

 access = true;

}
```

Fonte: Elaborado pelo autor, 2023.

No trecho de código 3 é definido a chamada da requisição para o cadastro do usuário, há uma rota padrão *Post*.

Trecho de Código 3: Cadastro de Usuário (users.controller.ts).

Código TypeScript

```typescript
@ApiTags('users')
@Controller('users')
@UseGuards(AuthGuard('jwt'))
export class UsersController {
 constructor(private readonly usersService: UsersService) {}

 @Post()
 async createUser(@Body() dto: CreateUserDto): Promise<User> {
   return this.usersService.createUser(dto);
 }
}
```

Fonte: Elaborado pelo autor, 2023.

No trecho de código 4 é possível observar a lógica da requisição de criação de usuário, é realizado uma encriptação utilizando *hash* (função responsável por transformar dados de entrada em um valor de *hash* único) na senha fornecida para que ela seja armazenada de forma segura no banco de dados.

Trecho de Código 4: Cadastro de Usuário (users.service.ts).

Código TypeScript

```typescript
@Injectable()
export class UsersService {
 constructor(
   @InjectRepository(User)
   private readonly userRepository: Repository<User>,
 ) {}


 async createUser(dto: CreateUserDto): Promise<User> {
   dto.password = await hash(dto.password, Number(process.env.SALT_OR_ROUNDS));
   const user = await this.userRepository.create(dto);
   await this.userRepository.save(user);


   return user;
 }
}
```

Fonte: Elaborado pelo autor, 2023.

No trecho de código 5 é possível observar o módulo do usuário, responsável por definir qual parte do código é pública para ser utilizada em outros lugares do projeto e também se há códigos de outro lugar do projeto que serão utilizados aqui.

Trecho de Código 5: Cadastro de Usuário (users.module.ts).

Código TypeScript

```typescript
@Module({
 imports: [TypeOrmModule.forFeature([User])],
 controllers: [UsersController],
 providers: [UsersService],
 exports: [UsersService],
})
export class UsersModule {}
```

Fonte: Elaborado pelo autor, 2023.

APÊNDICE B - TRECHOS DE CÓDIGO (LOGIN DE USUÁRIO)

No trecho de código 6 é definido o construtor com informações do jwt para configurações de encriptação e o método de validação de dados.

Trecho de Código 6: Login de Usuário (jwt.strategy.ts).

Código TypeScript

```typescript
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
 constructor() {
   super({
     jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
     ignoreExpiration: false,
     secretOrKey: process.env.JWT_SECRET_KEY,
   });
 }

 async validate(payload: any) {
   return { userId: payload.sub, username: payload.username };
 }
}
```

Fonte: Elaborado pelo autor, 2023.

No trecho de código 7 é definido a chamada da requisição para o login do usuário, há uma rota */login* do tipo *Post*.

Trecho de Código 7: Login de Usuário (auth.controller.ts).

Código TypeScript

```typescript
@ApiTags('auth')
@Controller('auth')
@UseGuards(LocalAuthGuard)
export class AuthController {
 constructor(private readonly authService: AuthService) {}

 @PublicRoute()
 @Post('/login')
 async login(@Request() req) {
   return this.authService.login(req.user);
 }
}
```

Fonte: Elaborado pelo autor, 2023.

No trecho de código 8 é possível observar a lógica da requisição de *login* de usuário, é realizada uma consulta ao banco de dados para verificar se o usuário existe, caso não exista, é lançada uma exceção. Em seguida é testado se o usuário está com acesso ativo através do atributo *access*, se não estiver, é lançada uma exceção; caso contrário, é realizado o login e retornado um *accessToken*.

Trecho de Código 8: Login de Usuário (auth.service.ts).

Código TypeScript

```
@Injectable()
export class AuthService {
 constructor(
   private readonly usersService: UsersService,
   private readonly jwtService: JwtService,
 ) {}


 async login(user: any) {
   const payload = { username: user.username, sub: user.userId };


   const dbUser = await this.usersService.getUserById(user.userId);
   if (!dbUser) {
     throw new HttpException(
       MessagesHelper.PASSWORD_OR_USER_INVALID,
       HttpStatus.NOT_FOUND,
     );
   }
   if (dbUser.access == false) {
     throw new HttpException('Unauthorized User', HttpStatus.UNAUTHORIZED);
   }
```

```
    try {
      const token = this.jwtService.sign(payload);
      return {
        accessToken: token,
      };
    } catch (e) {
      console.log(e);
    }
  }
}
```

Fonte: Elaborado pelo autor, 2023.

No trecho de código 9 é possível observar o módulo de autenticação, responsável por definir qual parte do código é pública para ser utilizada em outros lugares do projeto e também se há códigos de outro lugar do projeto que serão utilizados aqui. Além disso, é definido a origem do *jwt-secret-key* e o tempo de expiração do *accessToken*.

Trecho de Código 9: Login de Usuário (auth.module.ts).

Código TypeScript

```
@Module({
 imports: [
   UsersModule,
   PassportModule,
   JwtModule.register({
     secret: process.env.JWT_SECRET_KEY,
     signOptions: { expiresIn: '1h' },
   }),
 ],
 controllers: [AuthController],
 providers: [AuthService, LocalStrategy, JwtStrategy],
 exports: [AuthService],
})
export class AuthModule {}
```

Fonte: Elaborado pelo autor, 2023.

APÊNDICE C - TRECHOS DE CÓDIGO (RASTREIO DE ENCOMENDA)

No trecho de código 10 é definido a chamada da requisição para listar todas as encomendas do usuário, há uma rota */user/:userId* do tipo *Get*, que é necessário passar o id do usuário na rota para a identificação.

Trecho de Código 10: Rastreio de Encomenda (order.controller.ts).

Código TypeScript

```typescript
@ApiTags('order')
@Controller('order')
export class OrderController {
 constructor(private readonly orderService: OrderService) {}

 @Get('/user/:userId')
 async getAllOrdersFromUser(
   @Param('userId') userId: number,
 ): Promise<Order[]> {
   return this.orderService.getAllOrdersFromUser(userId);
 }
}
```

Fonte: Elaborado pelo autor, 2023.

No trecho de código 11 é possível observar a lógica da requisição de rastreio de encomenda, é realizada uma consulta ao banco de dados para verificar se o usuário existe, caso não exista, é lançada uma exceção. Em seguida é realizada uma consulta ao banco de dados para buscar as encomendas relacionadas a aquele usuário.

Trecho de Código 11: Rastreio de Encomenda (order.service.ts).

Código TypeScript

```typescript
@Injectable()
export class OrderService {
 constructor(
   @InjectRepository(Order)
   private readonly orderRepository: Repository<Order>,
   private readonly userService: UsersService,
 ) {}


 async getAllOrdersFromUser(userId: number): Promise<Order[]> {
   const user = await this.userService.getUserById(userId);
   if (user) {
     const orders = await this.orderRepository.find({
       where: { user: { id: userId } },
     });
     return orders;
   }
   throw new HttpException('User not found', HttpStatus.NOT_FOUND);
 }
}
```

Fonte: Elaborado pelo autor, 2023.

APÊNDICE D - TRECHOS DE CÓDIGO (SIMULADOR DE VALORES PARA FRETES)

No trecho de código 12 é definido a chamada da requisição para criar a simulação de valores para fretes, há uma rota padrão do tipo *Post*.

Trecho de Código 12: Simulador de Valores para Fretes (simulate-values.controller.ts).

Código TypeScript

```typescript
@ApiTags('simulate-values')

@Controller('simulate-values')

export class SimulateController {

 constructor(private readonly simulateService: SimulateService) {}



 @Post()

 async createSimulation(@Body() dto: CreateSimulationDto): Promise<number> {

   return this.simulateService.createSimulation(dto);

 }

}
```

Fonte: Elaborado pelo autor, 2023.

No trecho de código 13 é possível observar a lógica da requisição de simulação de valores para frete, é realizado um cálculo genérico utilizando os atributos altura, largura, comprimento e peso, disponibilizados pelo usuário.

Trecho de Código 13: Simulador de Valores para Fretes (simulate-values.service.ts).

Código TypeScript

```typescript
@Injectable()
export class SimulateService {
 constructor(
   @InjectRepository(Simulate)
   private readonly simulateRepository: Repository<Simulate>,
 ) {}


 async createSimulation(dto: CreateSimulationDto): Promise<number> {
   return (dto.height + dto.width + dto.length + dto.weight) / 4;
 }
}
```

Fonte: Elaborado pelo autor, 2023.