UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS ARARANGUÁ
CENTRO DE CIÊNCIAS, TECNOLOGIAS E SAÚDE (CTS)
TECNOLOGIAS DA INFORMAÇÃO E COMUNICAÇÃO (TIC)

Natalia Bortoli Vieira

Desenvolvimento de um Sistema de Backend Genérico para Aplicações de Software Comerciais

Araranguá

2024

Natalia Bortoli Vieira

Desenvolvimento de um Sistema de Backend Genérico para Aplicações de Software Comerciais

Trabalho de Conclusão de Curso submetido ao curso de Tecnologias da Informação e Comunicação do Centro de Ciências, Tecnologias e Saúde da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Tecnologias da Informação e Comunicação

Orientador: Prof. Dr. Fabrício Herpich

Araranguá

2024

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Bortoli Vieira, Natalia
Desenvolvimento de um Sistema de Backend Genérico para Aplicações de Software Comerciais / Natalia Bortoli Vieira ; orientador, Fabrício Herpich, 2024.
76 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Campus Araranguá, Graduação em Tecnologias da Informação e Comunicação, Araranguá, 2024.

Inclui referências.

1. Tecnologias da Informação e Comunicação. 2. Aplicações de Software Comerciais. 3. Backend Genérico. 4. APIs REST. 5. Arquitetura Multitenant. I. Herpich, Fabrício. II. Universidade Federal de Santa Catarina. Graduação em Tecnologias da Informação e Comunicação. III. Título.

Natalia Bortoli Vieira

Desenvolvimento de um Sistema de Backend Genérico para Aplicações de Software Comerciais

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel e aprovado em sua forma final pelo Curso de Graduação em Tecnologias da Informação e Comunicação.

Araranguá, 18 de Dezembro de 2024.

Documento assinado digitalmente
Fabricio Herpich
Data: 19/12/2024 16:28:24-0300
CPF: ***.335.260-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Fabrício Herpich, Dr.
Coordenador do Curso

Banca Examinadora:

Documento assinado digitalmente
Fabricio Herpich
Data: 19/12/2024 16:29:00-0300
CPF: ***.335.260-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Fabrício Herpich, Dr.
Orientador

Documento assinado digitalmente
Cristian Cechinel
Data: 19/12/2024 23:21:41-0300
CPF: ***.628.449-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Cristian Cechinel, Dr.
Universidade Federal de Santa Catarina

Documento assinado digitalmente
Thiago da Silva Fialho
Data: 20/12/2024 15:19:20-0300
CPF: ***.391.186-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Thiago da Silva Fialho, Mestrando
Universidade Federal de Santa Catarina

Araranguá, 2024

Resumo

No dinâmico cenário empresarial contemporâneo, marcado pela necessidade de agilidade e flexibilidade para atender às demandas de um mercado em constante transformação, a Transformação Digital (TD) e o uso intensivo de Tecnologias da Informação e Comunicação (TICs) desempenham um papel crucial na competitividade das empresas, impulsionando tanto a inovação quanto a eficiência operacional. Nesse contexto, o desenvolvimento ágil de aplicações surge como um desafio devido à sua complexidade, altos custos e demanda por profissionais altamente qualificados. Com o objetivo de contribuir para a redução de custos e esforços associados à construção da camada *backend*, este trabalho propõe o desenvolvimento de um sistema de *backend* genérico capacitado para fornecer, via *API REST*, recursos e funcionalidades para aplicações comerciais, com foco inicial no setor de serviços de entrega de produtos (*delivery*). A metodologia utilizada neste projeto foi a *Design Science Research Methodology* (*DSRM*), recomendada para o desenvolvimento de artefatos voltados à solução de problemas específicos. O sistema foi desenvolvido com *TypeScript* e *Nest.js*, e adotou uma arquitetura *multi-tenancy*, visando possibilitar o compartilhamento da infraestrutura tecnológica entre diferentes aplicações, sem riscos de interferência. A solução implementa princípios do padrão *REST*, proporcionando uma interface de comunicação uniforme e padronizada, tornando-a mais intuitiva e fácil de ser compreendida. Os resultados obtidos destacam o potencial do sistema em simplificar e padronizar processos no desenvolvimento de software, oferecendo funcionalidades reutilizáveis e adaptáveis a outros domínios comerciais, com possibilidades promissoras para expansão. Sugere-se a ampliação das funcionalidades para abranger novos contextos comerciais, a integração com um sistema de *frontend* para proporcionar uma solução completa e o desenvolvimento de recursos que aumentem a usabilidade e personalização. Adicionalmente, a inclusão de mecanismos que aprimorem o suporte a configurações específicas de negócios e a otimização do desempenho em ambientes de alta demanda são aspectos relevantes a serem explorados em futuras iterações.

Palavras-chave: *Backend* Genérico; *API REST* ; Transformação Digital; Desenvolvimento Ágil de Software; *Multi-tenancy*.

Abstract

In the dynamic contemporary business scenario, marked by the need for agility and flexibility to meet the demands of a constantly evolving market, Digital Transformation (DT) and the intensive use of Information and Communication Technologies (ICTs) play a crucial role in enhancing companies' competitiveness, driving both innovation and operational efficiency. In this context, the agile development of applications emerges as a challenge due to its complexity, high costs, and the demand for highly skilled professionals. Aiming to reduce the costs and efforts associated with building the backend layer, this study proposes the development of a generic system designed to provide, through a *REST API*, resources and functionalities for commercial applications, with an initial focus on the product delivery services. The methodology employed in this project was the *Design Science Research Methodology* (*DSRM*), recommended for developing artifacts aimed at solving specific problems. The system was developed using *TypeScript* and *Nest.js* and adopted a multi-tenancy architecture, enabling the technological infrastructure to be shared among different applications without risks of interference. The solution implements *REST* principles, providing a uniform and standardized communication interface, making it more intuitive and easier to understand. The results highlight the system's potential to simplify and standardize processes in software development, offering reusable and adaptable functionalities for other commercial domains, with promising possibilities for expansion. Suggested improvements include extending the functionalities to encompass new commercial contexts, integrating with a frontend system to provide a complete solution, and developing features that enhance usability and customization. Additionally, incorporating mechanisms to improve support for specific business configurations and optimizing performance in high-demand environments are relevant aspects to be explored in future iterations.

Keywords: Generic Backend; REST API; Digital Transformation; Agile Software Development; Multi-tenancy.

Lista de Figuras

Lista de Tabelas

Sumário

1 INTRODUÇÃO

Atualmente as empresas estão introduzidas em um ambiente global constantemente volátil e, por conta disso, se torna inquestionável o fato de que as organizações precisam responder de forma ágil e flexível às exigências do mercado para que consigam atender aos requisitos variáveis do ambiente ao qual estão inseridas (SANCHIS et al., 2019).

Para Sommerville (2011), esse cenário não só exige respostas ágeis em relação às oportunidades de mercado, mas também requer que as empresas se adaptem de forma eficiente em relação à concorrência, de forma a estarem sempre preparadas para ajustar suas estratégias diante do surgimento de novos produtos que geram uma vantagem competitiva aos seus concorrentes.

A transformação digital (TD), que diz respeito ao processo individual de cada organização utilizar tecnologias digitais a fim de se redefinir como um negócio digital, reflete o esforço das empresas em alcançarem uma posição competitiva vantajosa perante às mudanças do mercado. Como destaca Fabio Correa Xavier, mestre em Ciência da Computação, professor e colunista da MIT Technology Review Brasil,

> A transformação digital tem grande importância para as organizações, pois está diretamente relacionada à capacidade de competir em um mercado cada vez mais revolucionário. As empresas que adotam a TD podem obter vantagens competitivas em relação às suas concorrentes, seja por meio de redução de custos, aumento da eficiência, melhoria na qualidade do produto ou serviço ou mesmo por meio da criação de novos modelos de negócios (XAVIER, 2023, n.p).

Assim, as organizações que conseguem responder às demandas do mercado que estão inseridas com agilidade e eficiência têm maior chance de se destacar frente aos concorrentes. Tal perspectiva se aproxima da significação do termo agilidade por Baran e Woznyj (2020), como sendo um pré-requisito essencial para as organizações gerenciarem eficientemente a volatilidade do mercado.

Segundo Junkes (2014) *apud* Cassiolato e Silva (1995) o uso intensivo das Tecnologias da Informação e Comunicação (TICs) é um dos fatores que impacta diretamente na competitividade das organizações. As TICs também são consideradas fundamentais nos processos de transformação digital, pois além de proporcionarem uma vantagem competitiva, impulsionam a inovação e permitem que as empresas se adaptem de forma ágil diante das mudanças que podem ocorrer na esfera à qual estão inseridas.

De acordo com Sommerville (2011), os produtos de software, por estarem inseridos na ampla maioria dos processos de negócio das organizações, precisam que o seu desenvolvimento e entrega sejam rápidos para que seja possível aproveitar novas oportunidades e atender às

pressões competitivas do mercado. Dessa forma, a agilidade no desenvolvimento de produtos de software se torna um fator crucial e que requer novas soluções, dado que a abordagem tradicional de codificação manual é considerada um processo lento, bastante complexo e que gera muitos custos em termos de tempo, dinheiro e esforço (ALSAADI et al., 2021).

Nesse contexto, ferramentas que viabilizam o desenvolvimento rápido e eficaz de softwares tornam-se relevantes, pois além de contribuírem com fatores de redução de tempo, custos e complexidade relativos ao desenvolvimento, também favorecem as estratégias relacionadas à transformação digital das corporações ao agilizarem a introdução de novos produtos no mercado, proporcionando uma vantagem competitiva para o negócio em questão.

1.1 JUSTIFICATIVA

Nos últimos anos, a sociedade tem testemunhado avanços significativos no campo da tecnologia da informação e comunicação (TIC). Esse processo de avanço tecnológico evidenciou que a integração e a consolidação das aplicações de software na sociedade não só beneficia seus consumidores, mas melhora a eficiência operacional das organizações, bem como constituem um negócio para quem as desenvolve.

Assim, surge uma crescente demanda pela adoção e pelo desenvolvimento destas aplicações e, nesse contexto, a exigência relativa a agilidade no desenvolvimento e na entrega de aplicações de software emerge, tornando este um fator crucial para que seja possível responder às demandas dos usuários, aproveitar novas oportunidades e atender às pressões competitivas do mercado simultaneamente.

Tendo isso em conta, o processo de desenvolvimento da ampla maioria das aplicações nos dias de hoje inclui essencialmente o planejamento e a criação de um sistema de *backend* (NGUYEN, 2016). O *backend* é a camada do software responsável por alimentar a aplicação, fornecendo dados e recursos específicos que dão suporte ao seu funcionamento no contexto em que estão inseridas.

No entanto, o desenvolvimento de um sistema de *backend* exige tomadas de decisões críticas que impactam diretamente a performance, segurança e confiabilidade da aplicação. Tais medidas envolvem desde a definição de regras e processos, que orientam o funcionamento da aplicação, até o armazenamento, a recuperação e a manutenção de dados. Sendo assim, o processo de criação dessa camada tende a consumir mais tempo, principalmente caso esta seja personalizada, como as aplicações de comércio eletrônico. Além disso, é necessário contar com desenvolvedores *backend* qualificados, o que representa um obstáculo não só em relação à disponibilidade de mão de obra qualificada, mas também por conta de que esses profissionais costumam ter um salário médio superior ao de outros desenvolvedores, como os de *frontend*, por exemplo.

Dessa forma, o desenvolvimento de um sistema de *backend* envolve altos custos relativos ao desenvolvimento, tanto em termos financeiros, quanto em relação ao tempo gasto para o desenvolvimento do sistema. À luz disso, soluções capazes de aliviar ou cessar os esforços referentes ao desenvolvimento da camada *backend* são cada vez mais procuradas, tornando-se cada vez mais relevantes e essenciais nesse contexto.

De forma a responder a estas necessidades, o presente trabalho propõe o desenvolvimento do protótipo de um sistema de *backend* genérico, capaz de fornecer, via *API REST*, recursos e funcionalidades aplicáveis ao desenvolvimento de aplicações comerciais, com foco inicial no domínio de serviços de entrega (*delivery*). O propósito é agilizar a elaboração e a entrega desses produtos, reduzindo fatores como o tempo e o custo associados ao desenvolvimento, permitindo que o foco no desenvolvimento de aplicações seja direcionado exclusivamente ao desenvolvimento da camada *frontend*, a qual constitui definições associadas às interações do usuário e ao design das interfaces da aplicação.

Adicionalmente, a camada *frontend* oferece o ponto de contato direto com o usuário final do produto e, por conta disso, é a parte da aplicação onde diferenciais competitivos podem ser implementados, como uma experiência de uso mais intuitiva ou um design de interface mais atraente.

1.2 OBJETIVOS

Os objetivos do trabalho estão divididos em objetivo geral e objetivos específicos.

1.2.1 Objetivo Geral

Este projeto visa desenvolver um sistema de *backend* genérico, capaz de fornecer, via *API REST*, recursos e funcionalidades aplicáveis ao desenvolvimento de aplicações de software comerciais, com foco inicial em plataformas de serviços de entrega *delivery*. O propósito central é desenvolver o protótipo de uma ferramenta com capacidade de agilizar a criação e entrega de aplicações de software no âmbito comercial, reduzindo fatores como tempo e custos associados ao desenvolvimento.

1.2.2 Objetivos Específicos

- Explorar os fundamentos do padrão arquitetural *REST*, com ênfase em seus princípios mais relevantes para o desenvolvimento de *web services*.
- Realizar o levantamento dos requisitos para o desenvolvimento do sistema, os diagramas de caso de uso e o modelo de dados.

- Descrever o fluxo das requisições relativa ao uso da *API* do sistema para o desenvolvimento de aplicações de *delivery*.
- Verificar a aplicabilidade dos recursos implementados no sistema para serem reutilizados em outros domínios de aplicações comerciais.

2 REFERENCIAL TEÓRICO

Neste capítulo, serão apresentados os conceitos fundamentais de *APIs* e do padrão arquitetural *REST*, detalhando seus princípios e práticas. Esses conceitos fornecem a base teórica para o desenvolvimento do sistema proposto, pois orientam a construção de uma *API* eficiente, padronizada e capaz de atender às necessidades de interoperabilidade, característica essencial para um *backend* genérico e reutilizável.

2.1 INTERFACE DE PROGRAMAÇÃO DE APLICAÇÕES *(API)*

Uma *API* — sigla em inglês para *Application Program Interface* — é uma aplicação cliente-servidor que, através de um conjunto de definições e protocolos, permite a interoperabilidade entre sistemas, ou seja, viabiliza a comunicação direta entre aplicações. Ian Sommerville, em seu livro Engenharia de Software, 9ª edição, define uma *API* como

> Uma interface, normalmente especificada como um conjunto de operações, que permite acesso a uma funcionalidade da aplicação. Isso significa que essa funcionalidade pode ser chamada diretamente por outros programas e não apenas acessada através da interface de usuário (SOMMERVILLE, 2011)

.

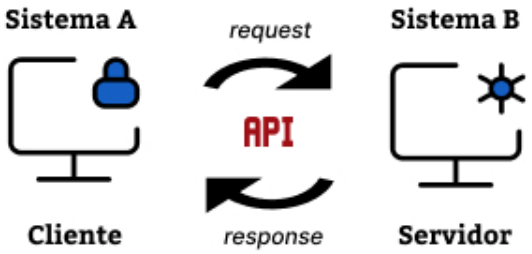
As *APIs* ditam as regras e os métodos de interação, atuando como um contrato para a comunicação de dois sistemas e possibilitando que os desenvolvedores integrem informações e funcionalidades de um sistema em outro, sem que precisem conhecer os detalhes internos de como esses recursos foram implementados.

Em termos práticos, um sistema disponibiliza uma *API* com *endpoints* que são pontos de acesso identificados por *URIs (Uniform Resource Identifier)* específicas, geralmente na forma de *URLs (Uniform Resource Locator)*. Esses *endpoints* são responsáveis por receber solicitações de outro sistema, processar a solicitação e fornecer os recursos ou serviços solicitados, geralmente na forma de dados estruturados.

A Figura 1 ilustra esse conceito. Nela, o sistema A representa uma aplicação desenvolvida por um programador e é denominado como Cliente. Por outro lado, o sistema B, denominado Servidor, contém as informações ou funcionalidades que o sistema A requer.

Em essência, o cliente solicita os recursos que deseja obter ou manipular através de uma requisição (*request*) em uma *URI* específica disponibilizada pelo servidor por meio de uma *API*. Sendo assim, essa interface programável atua como um intermediador, processando a solicitação do cliente, buscando as informações no servidor e fornecendo os recursos solicitados por ele através de uma resposta (*response*). Esse mecanismo permite que um sistema acesse e interaja

Figura 1 – Cliente-Servidor



Fonte: A autora (2024)

com os recursos disponíveis de outro sistema, representando o papel central de uma *API* na prática. Assim, podemos dizer que o funcionamento de uma *API* é como um 'serviço de *delivery*' de informações e funcionalidades.

Contudo, o papel das *APIs* se baseia na abstração da complexidade das funcionalidades de um sistema subjacente, de forma a ocultar os detalhes internos e complexos de funcionamento e separar a infraestrutura que implementa os serviços do sistema que os utiliza. Dessa forma, as *APIs* otimizam tempo no desenvolvimento de aplicações ao possibilitarem que recursos de um determinado sistema sejam utilizados por outros, permitindo que os desenvolvedores se concentrem apenas em como consumi-las e na utilização e organização dos recursos obtidos por meio delas.

Além disso, dependendo dos recursos oferecidos por uma *API*, os desenvolvedores podem direcionar seus esforços exclusivamente para o desenvolvimento da interface do usuário (*frontend*), priorizando a experiência do usuário. Podemos utilizar a solução proposta neste trabalho para exemplificar essa abordagem, visto que o sistema desenvolvido conta com uma *API* baseada nos princípios *REST*, projetada para fornecer funcionalidades aplicáveis ao desenvolvimento de softwares comerciais. Esse sistema pode ser visto como uma ferramenta capaz de eliminar a necessidade dos desenvolvedores de se preocuparem com a criação e estruturação da camada *backend* desses tipos de aplicações.

2.2 PADRÃO *REST*

Como conceituado no tópico anterior, as *APIs* são interfaces responsáveis por permitir e garantir a interoperabilidade entre sistemas distintos. No entanto, tanto na hora do desenvolvimento, quanto na hora do consumo, é importante que as *APIs* sejam construídas seguindo as melhores práticas e os melhores padrões.

Atualmente, o padrão arquitetural - acrônimo para *Representation State Transfer*, ou,

em português, Transferência de Estado Representacional - é um dos modelos arquiteturais mais utilizados no desenvolvimento de *APIs* por estabelecer princípios que incluem boas práticas de design e implementação que facilitam a comunicação entre aplicações. Esse padrão foi criado por Roy Fielding, no ano 2000, em sua tese de doutorado. Roy é um dos principais autores sobre as especificações do protocolo *HTTP (Hypertext Transfer Protocol)*, esteve envolvido no desenvolvimento do *HTML (HyperText Markup Language)* e de *URIs (Uniform Resource Identifier)*, e também foi o fundador do *Apache HTTP Server*, um dos servidores web mais populares e amplamente utilizados no mundo todo.

Na prática, a transferência de estado representacional ocorre a partir de uma solicitação *REST* enviada pelo cliente ao servidor, o qual responde a solicitação com uma representação dos estados que foram solicitados. No entanto, uma solicitação *REST* não serve apenas para consultar e enviar informações, mas também permite que ações sejam realizadas, como a criação de novos registros ou a exclusão de registros existentes. Assim, esse conceito abrange tanto a recuperação de dados quanto operações de manipulação de recursos.

Como mencionado anteriormente, este padrão conta com diversos princípios que visam estabelecer boas práticas de implementação e conceitos estruturais relacionados à organização e ao design da arquitetura. Contudo, ao desenvolver uma *API*, pode-se optar por implementar todos ou apenas alguns dos princípios *REST*. No contexto do presente trabalho, para o desenvolvimento do protótipo do sistema proposto, foram considerados e implementados alguns dos principais princípios que esse padrão abrange, que incluem: a arquitetura cliente-servidor, o paradigma *stateless* e a interface uniformizada. Esse último abrange quatro conceitos, dos quais optou-se por adotar a identificação dos recursos da aplicação, a representação desses recursos e a utilização adequada dos métodos *HTTP*. Tais princípios foram discutidos nas seções seguintes, divididos em princípios *REST* de arquitetura (subseção 2.2.1) e princípios *REST* de implementação (subseção 2.2.2).

2.2.1 Princípios *REST* de Arquitetura

Fielding (2000) definiu princípios fundamentais em sua tese que formam a base da arquitetura *REST*. Alguns desses princípios foram considerados no desenvolvimento do sistema proposto neste trabalho e estão especificados nos tópicos a seguir.

2.2.1.1 Arquitetura Cliente-Servidor

A arquitetura cliente-servidor é uma das restrições associadas ao padrão *REST*. Segundo Fielding (2000) o princípio por trás dessa restrição arquitetural consiste na separação de responsabilidades entre os sistemas cliente e servidor, de forma que as preocupações relacionadas à interface do usuário (cliente) sejam separadas das preocupações relacionadas ao armazena-

mento de dados (servidor), tornando os componentes presentes em cada um desses sistemas independentes e possibilitando que evoluam de forma autônoma.

Fielding (2000) destaca que essa abordagem possibilita aumentar a portabilidade da interface do usuário, de forma que esta seja capaz de funcionar em diferentes ambientes ou sistemas operacionais, sem precisar fazer modificações significativas no design da interface, o qual pode ser simplesmente adaptado ou reutilizado facilmente em diversas plataformas. Por outro lado, simplificar e modularizar os componentes do servidor aumenta a escalabilidade, ficando mais fácil lidar com um maior volume de requisições e de usuários simultâneos.

2.2.1.2 Paradigma *Stateless*

O paradigma *stateless* — em português, sem estado — é um tipo de "restrição" que refere-se à independência das requisições do cliente. Em outras palavras, cada requisição é interpretada de forma isolada pelo servidor, o qual deve considerar apenas o estado atual dos recursos presentes e nunca deve armazenar qualquer estado ou qualquer informação sobre a solicitação em si (RICHARDSON; RUBY, 2007).

Nesse contexto, segundo Fielding (2000) é indispensável que qualquer requisição de um cliente para um servidor contenha todas as informações necessárias para que o servidor possa compreender não só o tipo da solicitação, mas também as informações que ele deve retornar como resposta.

Em suma, o paradigma *stateless*, que também é conhecido como comunicação *stateless*, possibilita que uma aplicação seja escalável e performática, de forma a suportar requisições de múltiplos clientes.

Vale ressaltar que, de acordo com Richardson e Ruby (2007), caso o cliente precise que algum estado seja ponderado pelo servidor, ele pode interagir com o estado do recurso por meio do envio de representações em solicitações do tipo *PUT* ou *POST*, enquanto o servidor responde ao estado do cliente através de representações enviadas em solicitações *GET*, fundamentando assim o conceito de "Transferência de Estado Representacional".

2.2.1.3 Interface Uniforme

A relevância dada a uniformização da interface é considerada a característica principal que diferencia o estilo arquitetural *REST* de outros. De acordo com Fielding (2000) para alcançar uma interface uniforme associada ao *REST* é preciso seguir algumas restrições que orientam o comportamento dos componentes de uma aplicação.

Sendo assim, três das quatro principais restrições definidas por Fielding (2000) incluem a identificação de recursos, a manipulação desses recursos por meio de suas representações e

o uso de mensagens autodescritivas. Essas restrições serão apresentadas nas subseção 2.2.2.1, subseção 2.2.2.1.1 e subseção 2.2.2.2.

2.2.2 Princípios *REST* de Implementação

Os princípios de implementação *REST* são orientações que ditam alguns padrões arquiteturais devem ser seguidos e implementados para tornar a comunicação entre sistemas mais direta e eficiente. Alguns desses princípios foram implementados no desenvolvimento da *API* proposta neste trabalho e serão discutidos nos tópicos a seguir.

2.2.2.1 Identificação dos Recursos

Ao desenvolver uma aplicação baseada nos princípios *REST*, uma das regras fundamentais a serem seguidas consiste na identificação dos recursos existentes e na definição de *URIs* únicas para cada recurso identificado, com o intuito de que de que a aplicação seja capaz de definir qual deles deve ser manipulado em uma determinada requisição *HTTP*. O princípio que aborda a definição de *URIs* únicas será discutido de forma mais ampla na subseção 2.2.2.1.1

Segundo Richardson e Ruby (2007), um recurso pode ser caracterizado como algo que, por possuir relevância o suficiente, pode ser considerado uma entidade independente e, além disso, devem ser passíveis de serem armazenados em um computador e reproduzidos por um fluxo de bits. Os autores citam exemplos de recursos como sendo "um documento, uma linha em um banco de dados ou o resultado da execução de um algoritmo".

Adicionalmente, de forma mais simplificada, os recursos podem ser definidos como uma abstração das informações administradas por uma aplicação, como serviços, dados ou qualquer entidade que outro sistema, o cliente, pode interagir. Por exemplo, uma aplicação de *delivery*, assim como uma aplicação de *e-commerce*, administram recursos como produtos, clientes, pedidos, menus, entregas, entre outros.

2.2.2.1.1 Separação dos Recursos em *URIs* únicas

Este princípio se aplica à definição de *URIs* responsáveis por identificar um determinado recurso da aplicação. Em outras palavras, o cliente deve efetuar solicitações através de *URIs* específicas que indicam o recurso a ser manipulado em cada requisição. Dessa forma, será possível acessar ou manipular um recurso apenas por meio da URI que o identifica, de forma a também definir como a estrutura da aplicação é organizada.

Dentro dos múltiplos documentos *Request for Comments* (*RFC*) publicados pela *Internet Engineering Task Force*, existe um documento intitulado *Uniform Resource Identifier (URI): Generic Syntax*, que foi elaborado por Berners-Lee, Fielding e Masinter (2005). Esse documento

também é conhecido por *RFC* 3986, e define a sintaxe e a semântica das *URIs*, fornecendo padrões para construí-las e utilizá-las.

Contudo, de acordo com o padrão *REST* e o *RFC* 3986, a sintaxe para criação de *URIs* se baseia em uma sequência hierárquica dos componentes a seguir:

URI = esquema ":" "//" **autoridade** "/" **caminho** ["?" **query**] ["#" **fragmento**]

- O **esquema**, ou *schema*, refere-se ao protocolo que deve ser utilizado para acessar um recurso na aplicação, como o *HTTP* ou *HTTPs*. O esquema é sempre seguido por dois pontos e duas barras "://".
- A **autoridade**, ou *authority*, refere-se ao nome do domínio e da porta do protocolo para a qual serão enviadas as requisições de acesso ao recurso. O domínio indica o servidor que hospeda o recurso e pode ser representado tanto por um nome de domínio registrado, quanto por um endereço de IP. A definição da porta é opcional, no caso de ela não ser definida, a porta padrão definida pelo protocolo é utilizada.
- O **caminho**, ou *path*, indica o nome do recurso que será manipulado, geralmente de forma hierárquica e pode ser dividido em segmentos separados por barras "/".
- A **query** e o **fragmento** são componentes opcionais que podem ser utilizados para indicar informações adicionais a serem consideradas na solicitação.

  – A **query** é um conjunto de parâmetros que podem ser utilizados para alterar ou filtrar a solicitação e é precedida por um ponto de interrogação "?".

  – O **fragmento**, também conhecido como âncora, é utilizado para identificar uma parte específica de um recurso e é precedido pelo símbolo "#".

2.2.2.2 Representação dos Recursos

Richardson e Ruby (2007) definem a representação de um recurso a partir da concepção de que um servidor web não é capaz de fornecer uma ideia, mas sim uma série de bytes, em um formato e em uma linguagem específica. A partir disso, afirmam que uma representação é simplesmente qualquer dado/informação útil sobre o estado atual de um recurso.

Dessa forma, este princípio *REST* consiste na especificação do formato de representação dos recursos de uma aplicação, em outras palavras, é uma negociação entre o cliente e o servidor, onde o cliente define os tipos de entrada e de retorno dos dados que serão transferidos em uma solicitação. As aplicações de software suportam diversos formatos de representação de recursos, como *JSON*, *XML* e *HTML*.

2.2.2.3 Métodos *HTTP*

O *HTTP* é um protocolo amplamente utilizado para transferência de dados entre sistemas em redes computacionais, que opera em conjunto com o protocolo de transporte *TCP/IP*. Este protocolo dispõe de métodos de interação que descrevem as ações que podem ser realizadas tanto para a manipulação quanto para a transferência de representações de recursos entre clientes e servidores.

De acordo com Fielding (2000) em sua tese de doutorado, o *HTTP* é o único protocolo desenvolvido especificamente para a transferência de representações de recursos. O documento RFC 7231, intitulado *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*, define os métodos do protocolo *HTTP* como sendo a fonte principal que dá significado para uma solicitação ao indicarem o objetivo da operação requisitada pelo cliente e o resultado que se é esperado. Dessa forma, esse protocolo fornece uma interface padronizada para a interação com os diferentes tipos de recursos de uma aplicação, através da manipulação e transferência de representações. (FIELDING; RESCHKE, 2014). Sendo assim, cada método *HTTP* possui um significado semântico específico, e utilizá-los corretamente faz parte do princípio de interface uniforme do padrão *REST*.

Devido à limitação das operações que podem ser realizadas em recursos disponíveis na web, o *HTTP* disponibiliza quatro métodos principais para as ações mais comuns, que são: *GET*, *POST*, *PUT* e *DELETE* (RICHARDSON; RUBY, 2007).

Com base nas definições de Allamaraju (2010), a seguir, serão apresentadas breves explicações sobre cada um destes métodos, abrangendo suas funções, os componentes necessários para realizar operações com estes métodos, bem como as respostas que podem ser obtidas após a sua execução. Adicionalmente, será abordado o método *HTTP PATCH*, que foi implementado no sistema desenvolvido neste trabalho e deve ser utilizado para efetuar determinadas solicitações.

- **GET**: é utilizado para obter uma representação do recurso especificado no *URI* da solicitação. Para efetuar este tipo de operação, é necessário incluir informações no cabeçalho da requisição, como *tokens* de autenticação e formatos de representação. O resultado geralmente é retornado em um corpo de resposta que contém uma representação dos dados solicitados.

- **POST**: permite a execução diferentes ações em um servidor, como a criação de novos recursos, a atualização de recursos que já existem e a realização de alterações em um ou mais recursos simultaneamente. Para efetuar essa operação, é preciso incluir uma representação do recurso no corpo da requisição, descrevendo os dados de criação ou de atualização propostos. A resposta obtida geralmente contém uma representação do recurso

recém-criado ou atualizado, e pode incluir instruções de redirecionamento, como a *URI* de acesso ao novo recurso.

• **PUT**: permite atualizar completamente ou substituir um recurso existente no sistema, e também pode ser utilizado para criar um novo recurso. Para efetuar uma operação do tipo *PUT*, é preciso incluir uma representação do recurso no corpo da requisição, que pode ou não ser igual ao recebido em uma solicitação *GET* subsequente.A resposta obtida pode conter um status da atualização, bem como uma representação completa do recurso atualizado no corpo da resposta. Vale ressaltar que, alguns sistemas podem exigir a inclusão de apenas algumas propriedades mutáveis específicas do recurso em questão.

• **DELETE**: permite que um cliente exclua um recurso no sistema, de acordo com a *URI* utilizada para efetuar a requisição de exclusão. Para realizar esse tipo de operação, é necessário incluir apenas informações no cabeçalho da requisição, sem corpo. A resposta pode ser de sucesso ou falha, e o corpo da resposta pode conter uma mensagem indicando o estado da operação, como “sucesso” ou “fracasso”.

• **PATCH**: diferentemente do método *PUT*, que é utilizado para a atualização ou substituição total de um recurso no sistema, o método *PATCH* suporta atualizações parciais ou únicas, aplicando modificações específicas a um recurso já existente. O formato do envio e da resposta de uma requisição do tipo *PATCH* segue os mesmo do método PUT, apresentando anteriormente.

3 METODOLOGIA

A metodologia adotada neste estudo refere-se a *Design Science Research Methodology* (*DSRM*), proposta por Peffers et al. (2007), que orienta a condução de pesquisas voltadas não apenas para contribuições teóricas, mas também para a criação de soluções práticas aplicáveis a problemas reais. A *DSRM* propõe uma abordagem estruturada em seis etapas, que são definidas sequencialmente como: identificação do problema e motivação, definição dos objetivos da solução, design e desenvolvimento, demonstração, avaliação e comunicação.

3.1 IDENTIFICAÇÃO DO PROBLEMA E MOTIVAÇÃO

Nesta etapa, foram identificadas os desafios, esforços e custos associados ao desenvolvimento da camada *backend* de uma aplicação de software, como já mencionado na seção 1.1, bem como as necessidades de agilizar o desenvolvimento de aplicações de software, apresentados no capítulo 1. Para identificar essas necessidades e desafios, foram realizadas atividades que incluíram:

- Identificação das demandas por agilidade no desenvolvimento de software em um cenário global volátil, por meio de pesquisas bibliográficas sobre transformação digital e competitividade nas organizações.
- Identificação dos desafios associados ao desenvolvimento da camada *backend* de software, por meio de experiências pessoais e acadêmicas da autora nesta área de conhecimento.

3.2 DEFINIÇÃO DOS OBJETIVOS PARA UMA SOLUÇÃO

Com base na identificação dos desafios associados ao desenvolvimento de aplicações de software, especialmente os apresentados na construção da camada *backend*, foram estabelecidas metas gerais e específicas para a criação de uma solução eficaz. O objetivo principal deste trabalho é desenvolver um sistema de *backend* genérico, que forneça funcionalidades essenciais via *API REST*, voltado inicialmente para aplicações no âmbito de serviços de entrega (*delivery*).

A escolha do domínio inicial para construção da solução foi fundamentada na observação de que a ampla maioria das funcionalidades essenciais de uma aplicação de *delivery*, como o gerenciamento de produtos, cardápios, categorias e pedidos, pode ser reutilizada em outros sistemas de aplicações comerciais, tais como plataformas de *e-commerces*, tornando o domínio de serviços de entrega (*delivery*) uma base estratégica para o desenvolvimento do protótipo do sistema proposto.

Essa solução busca padronizar tarefas comuns no desenvolvimento da camada *backend* das aplicações de software do âmbito comercial, como a autenticação e o gerenciamento de dados e processos, e viabilizar a reutilização de funcionalidades essenciais, como o gerenciamento de produtos, categorias, pedidos, menus de produtos, clientes, entre outros.

Dessa forma, pretende-se contribuir com a agilidade no desenvolvimento e na entrega desses tipos de aplicações, reduzindo os esforços, o tempo e os custos envolvidos, permitindo que os desenvolvedores concentrem exclusivamente na criação de interfaces intuitivas e na melhoria da experiência do usuário.

3.3 PROJETO E DESENVOLVIMENTO

Nessa etapa ocorreu o planejamento e o desenvolvimento do protótipo do sistema proposto neste trabalho. O processo de criação iniciou-se com um *brainstorm* para analisar e levantar os recursos comuns entre aplicações comerciais, permitindo a compreensão das necessidades e especificações funcionais do sistema. Com base nessa análise e nos modelos de especificação da *Unified Modeling Language* (*UML*), foram elaborados diagramas de caso de uso para cada tipo de usuário do sistema, a partir dos quais foram analisados e especificados os requisitos funcionais e não funcionais, assim como as regras de negócio. Em seguida, o modelo de dados foi estruturado, envolvendo a definição das entidades genéricas e suas propriedades, bem como a identificação dos recursos da aplicação. Após essa fase, procedeu-se à codificação das funcionalidades definidas, juntamente com a definição de *URIs* exclusivas cada recurso, de *endpoints*, parâmetros, cabeçalhos e *payloads* necessários para realizar as operações sob os recursos implementados.

3.4 DEMONSTRAÇÃO

Para demonstrar as funcionalidades implementadas, foram criados diagramas que representam os seis fluxos de atividades mais relevantes previstos na utilização do sistema para o desenvolvimento de aplicações voltadas aos serviços de entrega (*delivery*). Esses fluxos descrevem requisições realizadas à interface do sistema, enquanto as raias dos diagramas indicam os módulos da aplicação onde cada requisição deve ser processada.

Além da representação gráfica, cada fluxo é acompanhado de um detalhamento das requisições ilustradas, evidenciando a conformidade com os princípios do design arquitetônico descritos no capítulo 2. Esse detalhamento inclui informações como a *URI* necessária para manipular o recurso correspondente, o método *HTTP* apropriado, os cabeçalhos requeridos, o conteúdo do corpo da requisição (quando aplicável) e a resposta esperada.

3.5 AVALIAÇÃO

Para avaliar a eficácia da ferramenta, foi elaborada uma documentação abrangente da *API*, detalhando as informações necessárias para interagir com o sistema. O sistema também foi implantado em um ambiente de produção, permitindo a realização de testes práticos. Esses testes foram conduzidos utilizando o *Postman*, com o objetivo de verificar a conformidade do sistema em relação aos requisitos funcionais e não funcionais definidos nas etapas iniciais do projeto.

Além disso, foi avaliada a aplicabilidade das funcionalidades desenvolvidas no sistema, com o objetivo de verificar seu potencial de reutilização em outros domínios de aplicações comerciais, destacando a versatilidade e a capacidade de expansão da solução proposta.

3.6 COMUNICAÇÃO

Nesta etapa do estudo, foram disseminados os resultados obtidos ao longo das etapas anteriores da pesquisa, que resultaram na elaboração do protótipo do sistema de *backend* proposto. Essa fase envolve a divulgação acadêmica do presente documento, que detalha todo o processo de desenvolvimento, e também a apresentação à banca avaliadora, durante a defesa deste Trabalho de Conclusão de Curso. Essa etapa desempenha um papel fundamental, pois permite que as contribuições teóricas e práticas sejam avaliadas e reconhecidas. Além disso, a comunicação dos resultados abre novas possibilidades de colaborações futuras.

4 PROJETO E DESENVOLVIMENTO

A etapa de *Projeto e Desenvolvimento* descreve o processo que vai desde a coleta de requisitos da ferramenta de software até a sua implementação prática. Neste capítulo, essas atividades serão apresentadas de forma detalhada.

4.1 ANALISE E ESPECIFICAÇÃO DE REQUISITOS

No contexto da *Engenharia de Software*, a elicitação de requisitos é uma prática inicial considerada fundamental para o desenvolvimento de uma aplicação. Essa etapa refere-se ao processo de descobrir, analisar e documentar, de forma textual, as funcionalidades, propriedades e limitações que determinado software exige, a fim de compreender e validar os padrões e especificações que devem ser supridos com base no domínio escolhido para o sistema.

Os requisitos são as descrições das funcionalidades, características e restrições que o sistema deve atender para satisfazer as necessidades dos usuários ou objetivos do projeto, sendo classificados em funcionais e não funcionais. Os requisitos funcionais, de acordo com Chichinelli (2017, p. 225) "dizem respeito à definição das funções que um sistema ou um componente de sistema deverá fazer, ou seja, as entradas que deverão ser transformadas e as saídas que deverão ser produzidas". Por outro lado, os requisitos não funcionais definem as limitações e as ações que o sistema precisará cumprir.

4.1.1 Requisitos Funcionais

Nesta seção são apresentados os requisitos funcionais do sistema. Estes foram organizados em tabelas distintas, correspondendo aos perfis de usuários presentes na aplicação, sendo eles: administrador do sistema, administrador da organização, administrador da loja e clientes.

Algumas das tabelas de requisitos funcionais foram categorizados em duas vertentes: aqueles comuns entre aplicações de softwares comerciais (exibidos na Tabela 1, Tabela 3, Tabela 5 e na Tabela 6) e aqueles específicos para as aplicações de software no âmbito de serviços de entrega (*delivery*), exibidos na Tabela 2 e na Tabela 4).

Tabela 1 – Requisitos funcionais gerais dos clientes

Código	Descrição
RF[00]	Podem se cadastrar no sistema.

RF[01]	Podem realizar *login*.
RF[02]	Podem visualizar o(s) menu(s) de produtos de uma loja.
RF[03]	Podem listar as categorias de produtos de uma loja.
RF[04]	Podem visualizar os produtos de uma loja por categoria.
RF[05]	Podem listar as informações de um produto, visualizando as opções e os adicionais disponíveis para o produto.
RF[06]	Podem adicionar um produto a sacola de compras.
RF[07]	Podem alterar as opções de um produto adicionado ao seu carrinho de compras.
RF[08]	Podem remover um produto do seu carrinho de compras.
RF[09]	Podem criar, listar, excluir e editar um ou mais endereços de entrega.
RF[10]	Podem listar as formas de pagamento disponíveis de uma loja.
RF[11]	Podem enviar um pedido para uma loja.
RF[12]	Podem visualizar as informações de seus pedidos, como status.

Fonte: A autora (2024)

Tabela 2 – Requisitos funcionais dos clientes - *delivery*

Código	**Descrição**
RF[00]	Podem adicionar uma observação ao produto.
RF[01]	Podem editar a observação de um produto em sua sacola de compras.
RF[02]	Podem editar os adicionais de um item na sua sacola de compras.
RF[03]	Podem visualizar a taxa de entrega aplicada ao seus endereços.

Fonte: A autora (2024)

Tabela 3 – Requisitos funcionais dos administradores da loja

Código	**Descrição**
RF[01]	Podem criar, listar, editar e excluir formas de pagamento.
RF[02]	Podem criar, listar, editar e excluir endereços e taxas de entrega.
RF[03]	Podem editar, listar e excluir horários de funcionamento da loja.
RF[04]	Podem criar, listar, editar e excluir produtos.
RF[05]	Podem criar, listar, editar e excluir opções de produtos, que representam variações do produto, como "Tamanho", "Cor"e "Modelo".
RF[06]	Podem criar, listar, editar e excluir adicionais de produtos.
RF[07]	Podem criar, listar, editar e excluir categorias de produtos.
RF[08]	Podem criar, listar, editar e excluir um conjunto de produtos, representando um menu, cardápio, ou catálogo.
RF[09]	Podem adicionar itens ao conjunto de produtos.
RF[10]	Podem abrir um caixa.
RF[11]	Podem listar os pedidos.
RF[12]	Podem atualizar o status de um pedido.
RF[13]	Podem listar os pedidos com base no status.
RF[14]	Podem criar, listar, editar e excluir usuários.
RF[15]	Podem criar, listar, editar e excluir os papéis de usuários, que agrupam um conjunto de permissões.
RF[16]	Podem vincular um papel a um usuário.

Fonte: A autora (2024)

Tabela 4 – Requisitos funcionais dos usuários da loja - *delivery*

Código	Descrição
RF[00]	Podem iniciar o funcionamento do *delivery*.
RF[01]	Podem encerrar o funcionamento do *delivery*.
RF[02]	Podem aceitar e recusar pedidos.

Fonte: A autora (2024)

Tabela 5 – Requisitos funcionais dos administradores da organização.

Código	Descrição
RF[00]	Podem criar, listar, editar e excluir lojas.
RF[01]	Podem criar e listar usuários administradores de uma determinada loja.

Fonte: A autora (2024)

Tabela 6 – Requisitos funcionais dos administradores do sistema

Código	Descrição
RF[00]	Podem criar, listar, editar e excluir organizações.
RF[01]	Podem criar e listar usuários administradores de uma organização.

Fonte: A autora (2024)

4.1.2 Requisitos Não-Funcionais

Neste tópico são apresentados os requisitos não funcionais, que foram identificados para restringir as funcionalidades do sistema proposto. Estes foram organizados em duas tabelas distintas, correspondendo aos requisitos não funcionais gerais do sistema (exibidos na Tabela 7) e aos requisitos não funcionais relativos aos usuários do sistema (exibidos na Tabela 8).

Tabela 7 – Requisitos não funcionais gerais do sistema

Código	Descrição
RF[00]	O sistema deve autenticar os usuários vinculados às organizações (como os usuários com papel de administrador da organização e administrador da loja, e clientes) garantindo que estes tenham acesso apenas aos dados associados ao *tenant* da organização a qual pertencem.
RF[01]	O sistema deve autenticar os usuários não associados a uma organização (usuários administradores do sistema) garantindo que estes tenham acesso aos dados do *tenant* público.
RF[02]	O sistema deve garantir que os dados persistidos não sejam deletados do banco de dados (*soft delete*).
RF[03]	O sistema deve garantir que, ao criar uma nova organização, um *tenant* exclusivo seja gerado automaticamente para ela, garantindo que cada organização possua um *tenant* próprio no sistema.
RF[04]	O sistema deve garantir que, ao criar uma nova organização, um usuário com o papel de administrador da organização seja gerado automaticamente.
RF[05]	O sistema deve garantir que, ao criar uma nova loja, um usuário com o papel de administrador da loja seja gerado automaticamente.
RF[06]	O sistema deve garantir que um pedido seja enviado apenas se este incluir um ou mais produtos.
RF[07]	O sistema deve garantir que um pedido seja enviado apenas se este incluir um endereço e um tipo de pagamento associado.
RF[08]	O sistema deve possibilitar que os pedidos seja aprovado manualmente.

Fonte: A autora (2024)

Tabela 8 – Requisitos não funcionais dos usuários do sistema

Código	**Descrição**
RF[00]	O sistema deve garantir que os clientes realizem o login o antes de enviar um pedido.
RF[01]	O sistema deve garantir que apenas os clientes possam adicionar, visualizar e gerenciar itens em sua sacola de compras.
RF[02]	O sistema deve garantir que apenas usuários com papel de administrador do sistema, ou aqueles com as permissões adequadas, possam criar organizações.
RF[03]	O sistema deve garantir que apenas usuários com papel de administrador geral do sistema, ou aqueles com as permissões adequadas, possam criar os usuários com papel de administrador da organização.
RF[04]	O sistema deve garantir que apenas usuários com papel de administrador da organização, ou aqueles com as permissões adequadas, possam criar as lojas.
RF[05]	O sistema deve garantir que apenas usuários com o papel de administrador da organização, ou aqueles com as permissões adequadas, possam criar usuários com papel de administrador da loja.
RF[06]	O sistema deve garantir que os usuários de uma loja realizem o login para acessar a aplicação.
RF[07]	O sistema deve garantir que somente usuários com papel de administrador da loja, ou aqueles com as permissões adequadas, possam criar, listar e gerenciar os produtos e categorias.
RF[08]	O sistema deve garantir que somente usuários com papel de administrador da loja ou aqueles com as permissões adequadas possam criar, listar e gerenciar caixas e tipos de pagamento.
RF[09]	O sistema deve garantir que somente usuários com papel de administrador da loja, ou aqueles com as permissões adequadas, possam criar, listar e gerenciar menus.

RF[10]	O sistema deve garantir que somente usuários com papel de administrador da loja, ou aqueles com as permissões adequadas, possam criar, listar e gerenciar opções do produto e os valores associados as opções do produto.
RF[11]	O sistema deve garantir que somente usuários com papel de administrador da loja, ou aqueles com as permissões adequadas, possam criar, listar e gerenciar os adicionais associados aos produtos.
RF[12]	O sistema deve garantir que somente usuários com papel de administrador da loja, ou aqueles com as permissões adequadas, possam gerenciar o status de um pedido.
RF[13]	O sistema deve possibilitar que os usuários com papel de administrador da loja, ou aqueles com as permissões adequadas, possam gerenciar o status de pagamento de um pedido.
RF[14]	O sistema deve garantir que apenas os usuários com papel de administrador da loja, ou aqueles com as permissões adequadas, possam criar, listar e gerenciar os bairros disponíveis para entrega.

Fonte: A autora (2024)

4.1.3 Casos de Uso

Os casos de uso são uma técnica da engenharia de software utilizada para descrever as interações entre os usuários (atores) e o sistema, com o objetivo de capturar e especificar os requisitos funcionais do software. Cada caso de uso representa um cenário específico em que um ator realiza uma sequência de ações para atingir um objetivo em relação ao sistema (BOOCH, 2005).

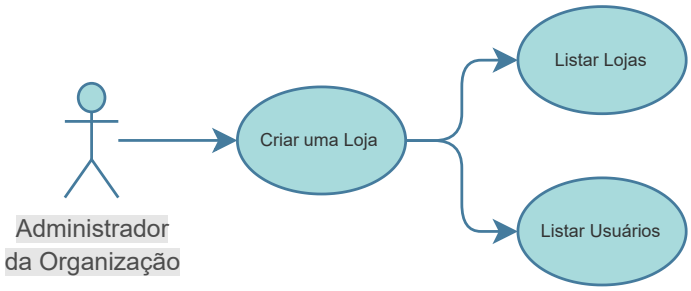
Essa técnica é amplamente empregada por sua capacidade de fornecer uma visão clara e detalhada das funcionalidades esperadas, facilitando o entendimento entre os envolvidos no desenvolvimento do projeto e promovendo uma documentação estruturada dos requisitos (SOMMERVILLE, 2011).

Os diagramas a seguir ilustram os casos de uso do sistema. Os casos de uso foram organizados em diagramas separados, correspondendo a cada perfil de usuário existente no sistema, sendo eles: Administrador do Sistema, Administrador da Organização, Administrador da Loja e Clientes.

Figura 2 – Diagrama de casos de uso dos administradores do sistema

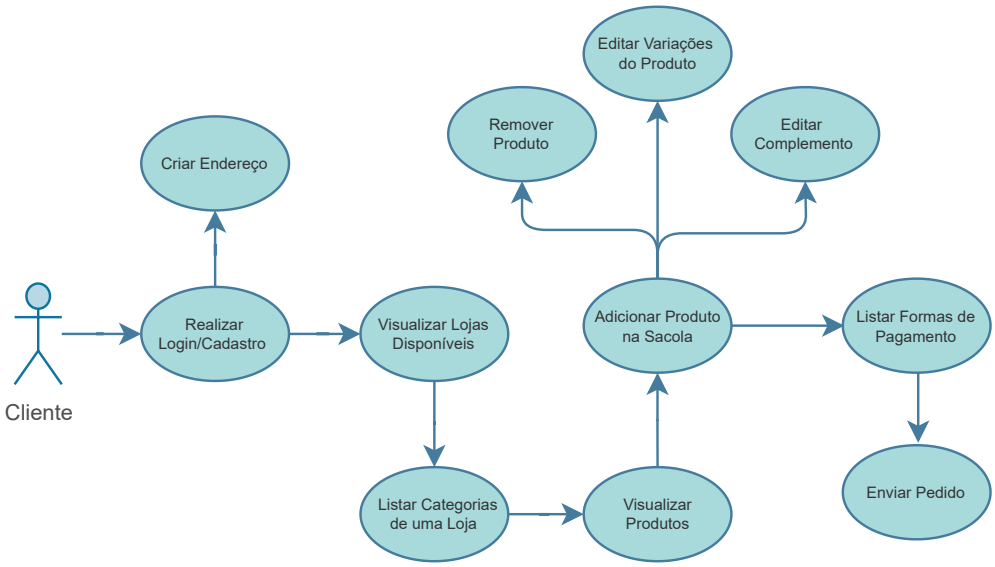
Fonte: A autora (2024)

Figura 3 – Diagrama de casos de uso dos administradores da organização



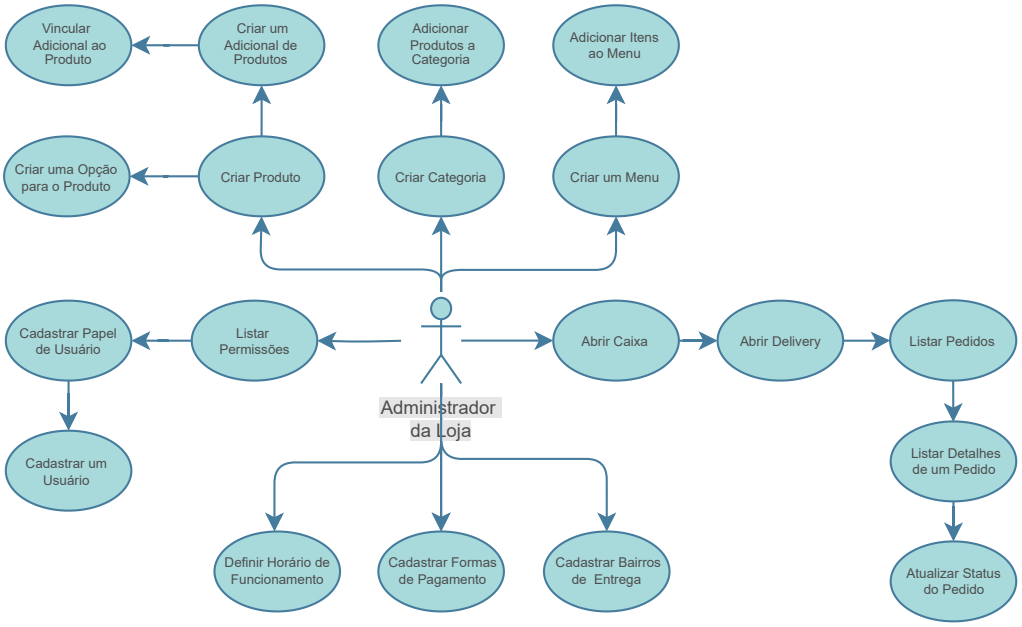
Fonte: A autora (2024)

Figura 4 – Diagrama de casos de uso dos clientes



Fonte: A autora (2024)

Figura 5 – Diagrama de casos de uso dos administradores da loja



Fonte: A autora (2024)

4.2 PLANEJAMENTO DO SISTEMA

Ao decorrer da fase de *Planejamento do Sistema*, foram concretizadas decisões sobre diversas características do sistema, abrangendo a arquitetura do sistema, a seleção da linguagem de programação e *framework*, e o planejamento do modelo de dados.

4.2.1 Modelo de Dados da Aplicação

Nesta seção será apresentado o modelo de dados associado a este projeto, descrevendo a função principal de cada uma das entidades definidas. Para cada entidade, será fornecida uma tabela com a descrição de suas propriedades específicas. A aplicação final é composta por 30 entidades genéricas, projetadas inicialmente para atender às necessidades de aplicações de serviços de entrega (*delivery*), mas que também podem ser utilizadas em outros domínios de aplicações do contexto comercial. Essas entidades são:

- **Organization**: representa um utilizador do sistema, como desenvolvedores autônomos ou vinculados a uma empresa. Tem como função principal definir a organização que está a utilizar os recursos do sistema.

- **Address**: representa informações detalhadas sobre os endereços de um cliente e de uma loja.

- **Store**: representa as lojas pertencentes a uma organização. Sua função é fornecer uma estrutura central para gerenciar os elementos relacionados à operação de uma loja.

- **StoreAddress**: atua como uma entidade associativa que associa a loja, representada pela entidade *Store*, a um endereço, definido pela entidade *Address*.

- **User**: representa os usuários do sistema, com exceção dos clientes, e tem como função permitir a criação de uma conta para cada usuário. Os campos '*userName*' e '*password*' são utilizadas para realizar o *login* e acessar diferentes áreas da aplicação, que variam conforme os níveis de permissão do usuário.

- **Role**: representa um papel ou uma função que contém um conjunto de permissões, as quais são definidas a nível de código. Tem como função verificar se um usuário possui permissão para efetuar uma determinada ação. Por exemplo, um papel de usuário pode ser "Gerente"ou "Atendente", cada um contendo um conjunto específico de permissões. Alguns *roles* (papéis) são definidos programaticamente, como os associados aos usuários administradores. Portanto, cada projeto individual deverá ter seus *roles* definidos manualmente, personalizando-os de acordo com as necessidades específicas da aplicação.

- **UserRole**: atua como uma entidade associativa ao representar um relacionamento entre as entidades *User* e *Role*. Tem como principal função estabelecer os papéis de um usuário do sistema.

- **RolePermission**: tem a função de associar permissões específicas a um papel (*role*) dentro do sistema. Ela define quais ações podem ser realizadas em um determinado módulo da aplicação, vinculando cada papel a suas permissões correspondentes.

- **Customer**: representa o usuário final na aplicação, ou seja, o cliente de uma loja. Sua função principal é permitir a criação de uma conta para cada usuário cliente, a qual posteriormente poderá ser utilizada para efetuar login.

- **CustomerAddress**: representa um relacionamento entre as entidades Customer e Address. Sua função é associar um cliente a um ou mais endereços.

- **CashRegister**: representa uma caixa registradora de uma loja. Sua principal função é registrar vendas.

- **Category**: representa uma categoria de produtos de uma determinada loja, permitindo a organização e classificação de produtos.

- **Product**: representa os produtos oferecidos por uma loja no sistema. Esta entidade armazena informações como o nome do produto e seu preço padrão, e está associada a diversas outras entidades, como itens de conjuntos, categorias, complementos e opções de produto. Seu papel central é definir e organizar os produtos comercializados pela loja.

- **ProductCategory**: atua como uma entidade associativa, estabelecendo o relacionamento entre produtos e categorias. Sua principal função é vincular um produto a uma ou mais categorias.

- **ProductSet**: representa um conjunto de produtos agrupados, definindo os produtos que fazem parte do menu de uma loja.

- **ProductSetItem**: atua como uma entidade associativa, associando produtos (*Product*) a um conjunto de produtos (*ProductSet*). Seu papel é definir quais produtos fazem parte de um determinado conjunto, permitindo também a definição de um preço específico para o produto dentro desse conjunto.

- **ProductOption**: representa as diferentes opções ou variações que podem ser associadas a um produto. As opções permitem que o produto tenha escolhas, como tamanhos, cores e ingredientes, e podem ser configuradas como sendo obrigatórias ou não. Além disso, a entidade define o tipo de opção e armazena os valores possíveis dessas variações.

- **ProductOptionValue**: representa os valores específicos de uma determinada opção de produto, permitindo que as variações definidas por uma opção tenham nomes e valores distintos. Por exemplo, em uma opção "Tamanho", os valores poderiam ser "Pequeno", "Médio", ou "Grande".

- **AddOn**: representa um item adicional que pode ser oferecido como um complemento para um produto de uma loja. Por exemplo, um adicional poderia ser "Bacon", "Maionese"ou "Cebola".

- **ProductAddOn**: atua como uma entidade associativa ao vincular produtos aos seus respectivos adicionais (*AddOns*). Ela permite que itens adicionais sejam associados a produtos específicos, oferecendo mais opções de personalização ao cliente.

- **Bag**: representa uma sacola de compras, agrupando os itens que um cliente deseja adquirir antes de finalizar a compra.

- **BagItem**: representa um item específico dentro da sacola de compras. Cada *BagItem* armazena informações sobre um produto individual, incluindo sua quantidade, preço unitário e as opções ou complementos associados, permitindo um controle detalhado dos itens que o cliente pretende comprar.

- **BagItemOption**: representa as opções específicas associadas a um item dentro da sacola de compras (*BagItem*), permitindo o gerenciamento das escolhas do cliente relativas às opções disponíveis para cada item na sacola.

- **BagItemAddOn**: representa os complementos adicionais associados a um item na sacola de compras (**BagItem**), permitindo o gerenciamento das escolhas do cliente relativas aos complementos disponíveis para cada item na sacola.

- **Order**: representa um pedido realizado por um cliente, encapsulando todas as informações relacionadas a essa transação. Um pedido é vinculado a uma sacola de compras (*Bag*), permitindo que o sistema registre os itens comprados, o caixa responsável pela transação e os pagamentos associados ao pedido.

- **Payment**: representa um pagamento associado a um determinado pedido, estabelecendo uma relação entre o tipo de pagamento utilizado e o pedido correspondente, permitindo o registro e controle financeiro das transações realizadas em uma aplicação. Essa entidade é essencial para garantir que os pagamentos sejam associados corretamente aos pedidos.

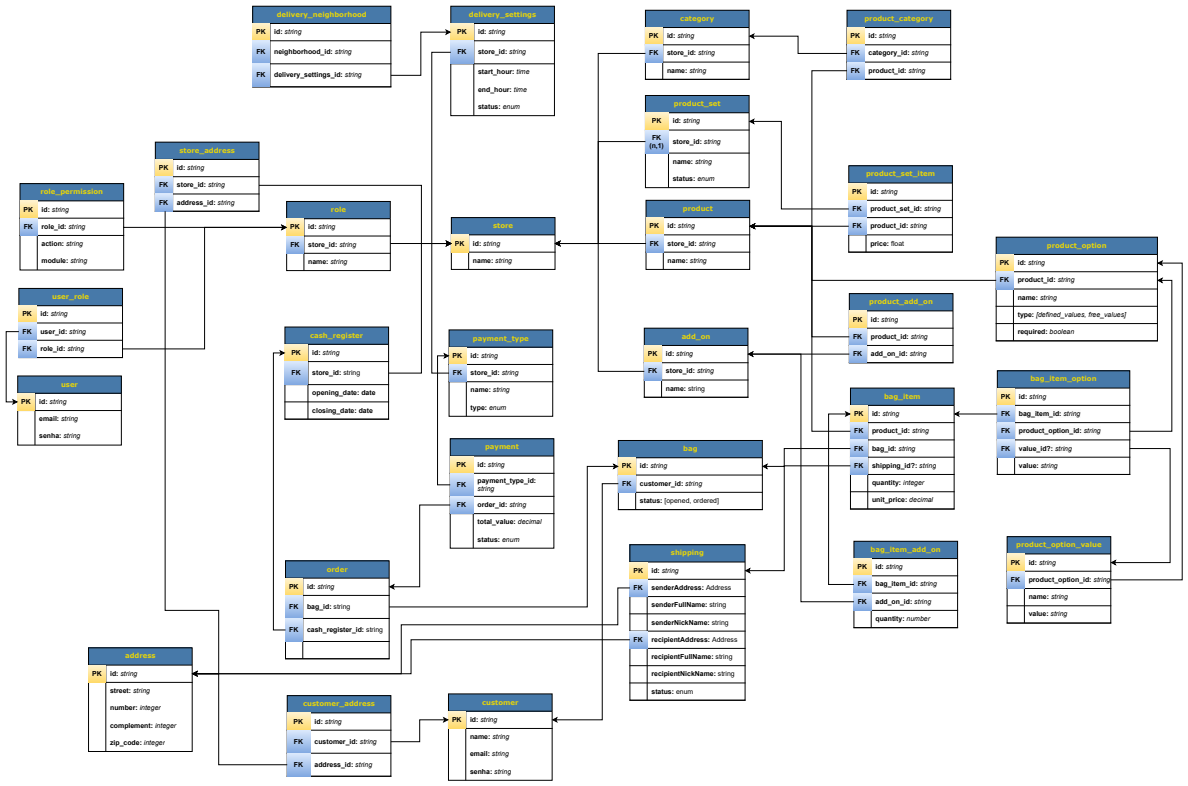
- **PaymentType**: representa os diferentes métodos de pagamento disponíveis, permitindo que os operadores de uma loja configurem e gerenciem suas opções de recebimento. Os tipos de pagamento são definidos programaticamente, sendo: "*CASH*", "*CREDIT*", "*DEBIT*"ou "*PIX*".

- **Shipping**: representa o processo de envio de produtos. Tem como função fornecer o gerenciamento dos detalhes de um envio, como as informações de nome e de endereço tanto do remetente, quanto do destinatário, e o status do envio, que pode variar entre "aguardando", "em andamento"e "concluído".

- **DeliverySettings**: é um modelo específico para as aplicações de serviços de entrega (*delivery*). Sua função principal é permitir que as lojas configurem os horários de funcionamento e definam quando o serviço de entrega está aberto ou fechado, garantindo que os clientes solicitem pedidos apenas enquanto o status estiver definido como "aberto".

- **DeliveryNeighborhood**: representa os bairros de entrega de uma loja. Tem como função definir os bairros nos quais uma loja oferece os seus serviços de entrega de produtos.

Figura 6 – Modelo Lógico do Sistema



Fonte: A autora (2024)

4.3 DESENVOLVIMENTO

Esta seção abrange a implementação prática do sistema, abordando os aspectos relacionados a estrutura da aplicação e a persistência dos dados.

4.3.1 Estrutura da Aplicação

Para o desenvolvimento do sistema, escolheu-se utilizar o *NestJS*, que é um dos principais frameworks do *Node.js* para construção de *backends* e *APIs*. Este framework oferece uma estrutura arquitetural que facilita a criação de aplicações modulares, testáveis e de fácil manutenção, além de proporcionar tipagem estática ao código ao adotar a linguagem de programação *TypeScript* por padrão. A estrutura do *Nest* possui alguns conceitos fundamentais que são essenciais para o desenvolvimento de uma aplicação básica.

A seguir, os principais conceitos foram mencionados e descritos, com base na leitura da documentação do NestJS (2024).

- ***NestCLI*** - O *NestCLI* é uma ferramenta de interface de linha de comando que auxilia os desenvolvedores na inicialização, no desenvolvimento e na manutenção da aplicação.

- ***Controllers*** - Os *controllers*, ou controladores, são classes da aplicação encarregadas de receber as requisições dos clientes e retornar as respostas adequadas a essas solicitações, geralmente anotadas pelo *Nest* com o decorador "*@Controller()*". Em aplicações Nest, cada recurso pode ter um ou mais controladores dedicados, vinculados a módulos específicos. Esses controladores definem uma ou mais rotas de acesso e as operações que cada rota executa, conforme o contexto em que são utilizados. O mecanismo de roteamento do *Nest* é responsável por direcionar as solicitações dos clientes aos controladores designados para processá-las, com base nas *URLs* e nos métodos *HTTP* utilizados nas requisições.

- ***Providers* e *Dependency Injection*** - Os *providers*, ou provedores, são classes anotadas com o decorador "*@Injectable()*" que possuem como propósito central proporcionar a injeção de dependências entre classes, possibilitando que os objetos criem múltiplos relacionamentos entre si. Diversas classes de uma aplicação desenvolvida com *Nest* podem ser ajustadas para serem um provedor (como serviços e repositórios) e devem ser declaradas como "*providers*" nos arquivos "*module.ts*".

- ***Services*** - Os *services*, ou serviços, são classes responsáveis por encapsular a lógica de negócio da aplicação e fornecer funcionalidades aos controladores e aos módulos. Como mencionado anteriormente, os serviços podem ser definidos como um provedor, a fim de permitir que suas funcionalidades sejam reutilizáveis em diferentes partes do sistema.

- ***Modules*** - Os *modules*, ou módulos, são classes anotadas com o decorador "*@Module()*", o qual retorna metadados que são utilizados pelo *Nest* para organizar a estrutura da aplicação. A ampla maioria das aplicações desenvolvidas com *Nest* resulta em uma arquitetura que emprega diferentes módulos, cada um sendo responsável por agrupar recursos que estão relacionados entre si de forma lógica e coesa. Através desta estrutura modular, o *Nest* proporciona organização ao código tornando-o mais fácil de entender, simplificando a gestão e a manutenção.

4.3.1.1 Módulos da Aplicação

Como mencionado no tópico anterior, o framework *NestJS* adota uma arquitetura modular, na qual diferentes módulos agrupam, de forma lógica e coesa, recursos relacionados entre si. Sendo assim, optou-se por organizar a estrutura do sistema em cinco módulos distintos, são eles:

- *Auth* - destinado à todos os usuários do sistema. Agrupa funcionalidades associadas a autenticação, como login e verificação do usuário atual logado.

- *Admin* - destinado aos usuários que operam recursos do sistema, como o administrador do sistema. Agrupa operações de acordo com os requisitos funcionais apresentados na Tabela 6.

- *Organization* - destinado aos usuários de uma organização. Agrupa um conjunto de operações que podem ser realizadas pelos usuários de uma organização, conforme apresentado na Tabela 5 de requisitos funcionais.

- *Store* - destinado aos usuários vinculados a uma loja. Agrupa um conjunto de operações que podem ser realizadas sobre os recursos da loja, conforme apresentado na Tabela 3 de requisitos funcionais.

- *Customer* - destinado aos clientes de uma loja (*store*). Agrupa as operações que um cliente pode realizar em uma determinada loja do sistema, de acordo com os requisitos funcionais dos clientes apresentados na Tabela 1.

O código-fonte do sistema desenvolvido está disponível publicamente e pode ser acessado por meio do repositório no *GitHub*: <https://github.com/nataliavieirab/api-tcc-ufsc.git>

4.3.2 Persistência de Dados

A persistência dos dados do sistema foi implementada utilizando o *TypeORM*, uma biblioteca de *Object-Relational Mapping* (*ORM*) em *TypeScript*, que permite o mapeamento das entidades da aplicação para tabelas no banco de dados relacional. Com o *TypeORM*, as entidades do sistema foram definidas em classes que especificam os atributos e relacionamentos dos dados que serão armazenados no banco, permitindo que os dados sejam manipulados de maneira eficiente e que as tabelas sejam geradas automaticamente no banco. A utilização dessa biblioteca simplificou a criação, consulta, atualização e exclusão de informações ao eliminar a necessidade de escrever consultas *SQL* de forma manual.

Além do mapeamento das entidades, foi implementado no sistema a arquitetura *multi-tenant*, que é um modelo de design de software no qual uma única instância de uma aplicação serve a vários "*tenants*"(inquilinos). Este tipo de arquitetura possibilita que cada instância da aplicação, representada pela entidade ***Organization*** (organização), possua seu próprio ambiente de dados isolado, chamado de *tenant*. Sendo assim, ao criar uma organização no sistema, um *tenant* é criado automaticamente e vinculado a ela. Dessa forma, o sistema permite que as operações de leitura e gravação no banco de dados sejam segregadas por organização, aumentando a segurança e garantindo que cada organização e seus respectivos usuários acessem apenas seus próprios dados em um ambiente compartilhado.

5 DEMONSTRAÇÃO

Conforme o exposto na seção 3.4 *Demonstração*, o presente capítulo tratará de demonstrar os principais fluxos de atividades previstos na utilização do sistema desenvolvido neste trabalho. Para isso, foram elaborados diagramas *BPMN* (*Business Process Model and Notation*), que representam visualmente diferentes sequências de atividades realizadas por diferentes usuários para a execução de seis funcionalidades essenciais do sistema. Tais funcionalidades incluem:

1. Criação de uma organização e de uma loja;
2. Gerenciamento de usuários de uma loja;
3. Gerenciamento de produtos, categorias e menus de uma loja;
4. Personalização de configurações específicas de uma loja;
5. Realização de um pedido por um cliente;
6. Gerenciamento de pedidos de uma loja.

Os fluxos de atividades ilustram etapas sequenciais que representam solicitações que podem ser realizadas à *API* do sistema para execução integral das funcionalidades oferecidas. Sendo assim, serão detalhados os formatos específicos de requisição correspondentes a cada uma dessas etapas operacionais.

5.1 REQUISITOS GERAIS

Nesta seção, serão apresentados os requisitos gerais para realizar solicitações à interface do sistema, com o propósito de evitar redundâncias nas explicações dos fluxos de atividades posteriores. Os requisitos estabelecidos definem um processo padrão para interação, demonstrando a consistência e a conformidade com os princípios arquitetônicos do padrão *REST* adotado no desenvolvimento da *API*.

- Seleção do método *HTTP* adequado
  - A operação a ser realizada deve incluir o método *HTTP* apropriado. A aplicação suporta os seguintes métodos:
    - GET: para obter dados de um recurso;
    - POST: para criar um novo recurso;
    - PUT: para atualizar um recurso existente;

    - DELETE: para remover um recurso;
    - PATCH: para atualizações parciais de um recurso.

- Definição da *URI* do recurso
  - É necessário incluir a *URI* associada ao recurso que será manipulado. A *URI* deve apontar para o módulo correto da aplicação, garantindo que a solicitação seja encaminhada ao contexto apropriado.

- Envio de dados no corpo da requisição
  - Para operações que requerem envio de informações adicionais, como a criação ou atualização de recursos, os dados devem ser fornecidos no corpo (*body*) da requisição, utilizando o formato de representação *JSON*. O conteúdo do *JSON* deve atender à estrutura esperada pela *API*, detalhada em cada caso específico.

- Autenticação e Permissões
  - O cabeçalho (*header*) da requisição deve conter um *token* de autenticação válido do usuário. Esse *token*, obtido por meio das respostas das requisições de login, é obrigatório para autorizar o acesso aos recursos e operações da *API*. O usuário autenticado deve possuir as permissões necessárias para realizar a operação desejada.

- Login de usuários e clientes
  - Para realizar o login de um usuário no sistema, é necessário enviar uma requisição com o método *HTTP POST* para a *URI* de autenticação "***/auth/login***". O corpo da requisição deve conter um objeto *JSON* com os atributos ***userIdentification*** — que pode ser representado por um e-mail ou nome de usuário — e ***password***. Essa requisição retornará um *token* de autenticação que deverá ser usado nas requisições subsequentes.
  - Para o login de usuários pertencentes a uma organização (todos os usuários, exceto administradores do sistema), também é necessário incluir no cabeçalho um campo denominado ***X_API_TOKEN***. Este campo deve conter o ID da organização a qual o usuário pertence.

5.2 FLUXOS DE ATIVIDADES

Dado que o sistema de *backend* desenvolvido neste trabalho disponibiliza suas funcionalidades por meio de uma *API REST*, nas subseções seguintes serão apresentados diagramas

previamente mencionados, que evidenciam operações-chave que garantem o funcionamento eficiente do sistema em seu contexto.

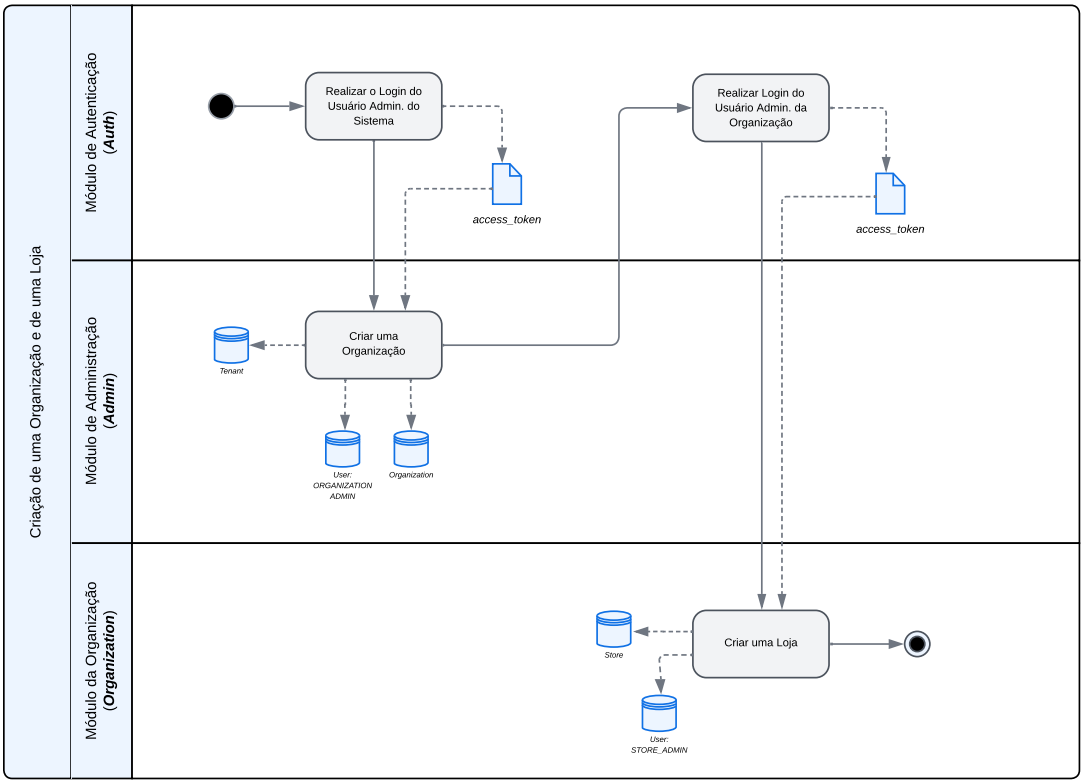
Nas representações das respostas das solicitações demonstradas nas seções seguintes, optou-se por omitir os dados de *timestamps* (datas de criação, atualização e deleção) de cada recurso, apesar de tais informações estarem efetivamente presentes no retorno prático de cada solicitação. A exclusão desses elementos decorreu da necessidade de simplificar a visualização, evitando representações excessivamente longas. De maneira similar, nas operações de busca de dados, os metadados relacionados à paginação (como número da página, total de registros e tamanho da página) também foram ocultados nas respostas demonstradas. Entretanto, esses dados foram integralmente implementados no sistema, garantindo sua funcionalidade em cenários reais, e podem ser consultados na documentação técnica elaborada para a *API*.

Por meio do link público no *SwaggerHub*, a documentação da *API* também serve como um guia abrangente para a compreensão e execução das operações implementadas no sistema (link de acesso à documentação).

5.2.1 Criação de uma organização e de uma loja

Uma organização representa um usuário dos recursos fornecidos pelo sistema, como um desenvolvedor ou uma empresa. Por sua vez, uma loja pode representar, por exemplo, um serviço de entrega (*delivery*) ou uma loja online (*e-commerce*). Nesse contexto, o fluxo de atividades abaixo demonstra, de forma sequencial, as primeiras requisições que devem ser realizadas para fazer o uso das funcionalidades oferecidas, que envolvem primeiramente a criação de uma organização e, em seguida, de uma loja. Essas operações são pré-requisitos para a execução das atividades apresentadas nos fluxos subsequentes.

Figura 7 – Fluxo de Atividades 1



Fonte: A autora (2024)

A primeira etapa ilustrada no fluxo diz respeito a realização do *login* de um usuário com papel de administrador do sistema. Após efetuar essa solicitação à *API*, um *token* de acesso associado a este usuário é retornado como resposta. Dessa forma, é possível executar a etapa subsequente, relativa à criação de uma organização, efetuando uma solicitação com o formato demonstrado no bloco de código abaixo.

Os campos *userName* e *userPassword*, representados no elemento *body* da requisição, referem-se às credenciais do usuário administrador da organização, o qual será gerado automaticamente ao realizar a operação em questão.

- **Criar uma organização:**

Formato da requisição para criar uma organização

```
POST {host}/admin/organizations

Header: "Authorization": {token_system_admin}

Body: {
   "name": "FoodDelivery",
   "email": "admin@foodelivery.com",
   "userName": "admin",
   "userPassword": "kijkf_9Iddd"
}

Response: 201 {
   "name": "FoodDelivery",
   "email": "admin@foodelivery.com",
   "id": "UUID"
}
```

A etapa seguinte ilustra a realização do *login* do usuário com papel de administrador da organização que foi gerado na etapa anterior. Ao obter o *token* deste usuário, é possível executar a próxima etapa, efetuando uma solicitação de criação de uma loja no formato especificado abaixo.

De forma semelhante a operação anterior, os campos *userName* e *userPassword* referem-se as credenciais do usuário administrador da loja, que também é gerado automaticamente ao executar a operação de criação de uma loja no sistema.

- **Criar uma loja:**

Formato da requisição para criar uma loja no sistema

```
POST /organization/stores

Header: "Authorization:" {organization_admin_token}

Body: {
 "name": "string",
 "userName": "string",
 "userPassword": "string"
}

Response: 201 {
   "name": "string",
   "userName": "string",
   "id": "UUID"
}
```

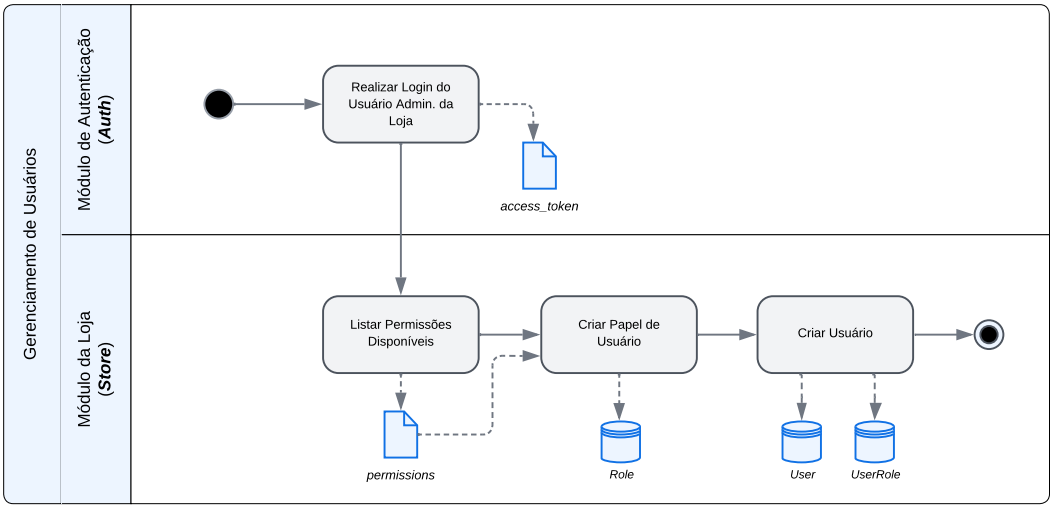
A execução dessas etapas no sistema geram registros nas entidades *Organization* e *Store*, as quais foram previamente apresentadas nasubseção 4.2.1 *Modelo de Dados*.

5.2.2 Gerenciamento de Usuários

O sistema fornece recursos abrangentes de gerenciamento de usuários operadores de uma loja, permitindo que os administradores criem, editem, listem e removam usuários vinculados à sua loja. Nesse contexto, um recurso que se destaca é a personalização de papéis de usuário. Essa funcionalidade permite definir conjuntos específicos de permissões que podem ser atribuídas quando novos usuários são cadastrados. As configurações das permissões são implementadas de forma programática no arquivo '*permissions.ts*' do projeto, garantindo um controle preciso e seguro dos acessos.

A seguir, o fluxo de atividades apresentado ilustra detalhadamente as etapas envolvidas no processo de gestão de usuários, demonstrando a sistemática de administração e configuração de acessos implementada no sistema.

Figura 8 – Fluxo de Atividades 2



Fonte: A autora (2024)

Ao executar a solicitação de *login* do usuário administrador da loja, um *token* de acesso exclusivo vinculado a sua conta é obtido. Com esta autenticação estabelecida, é possível prosseguir com a execução das três etapas subsequentes ilustradas no diagrama, iniciando pela listagem de permissões, posteriormente a criação de papéis de usuários e por fim a criação dos usuários em si. Os elementos que compõem os formatos das solicitações esperados pela aplicação para a execução dessas operações no sistema são apresentados a seguir.

- **Listar permissões disponíveis:**

Formato da requisição para listar permissões

```
GET /store/{store_id}/permissions

Response: 200 {[
     "createCashRegister",
     "createCategory",
     "createProduct",
     ....
   ]}
```

- **Criar papel de usuário:**

Elementos da requisição de criação dos papéis de usuário

```
POST /store/{store_id}/roles

Body: {
   "name": "string",
   "permissions": [
     "string"
   ]}

Response: 201 {
   "name": "string",
   "id": "UUID"
}
```

- **Criar usuário:**

Elementos da requisição de criação de usuários

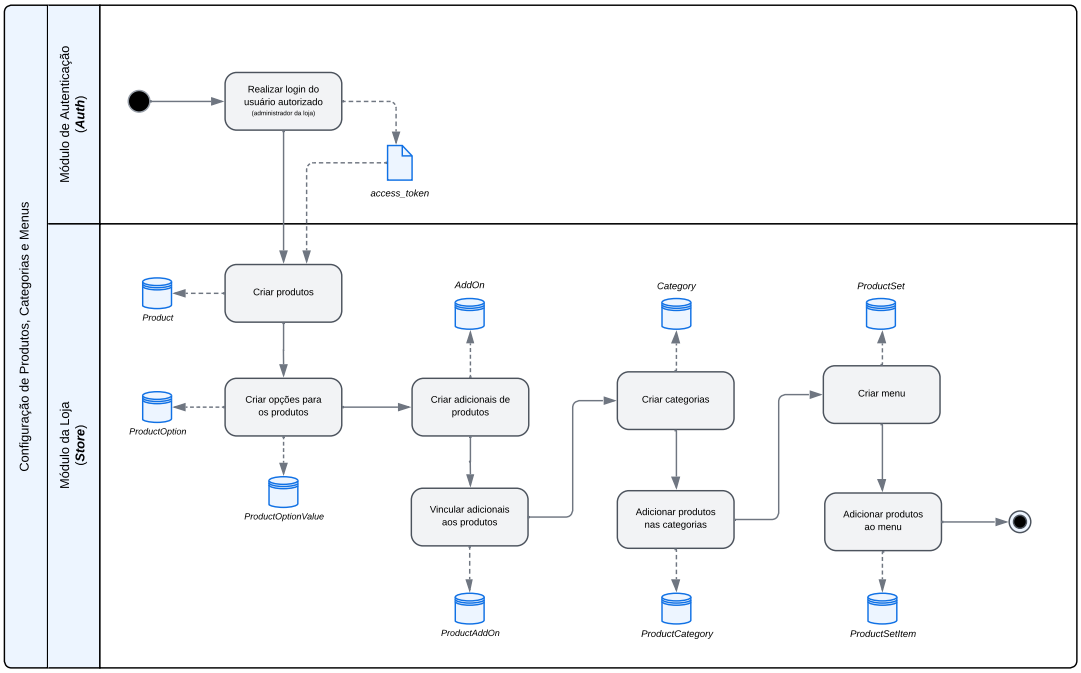
```
POST /store/{store_id}/users

Body: {
   "userName": "string",
   "password": "string",
   "roles": ["UUID"]
}

Response: 201 {
   "userName": "string",
   "id": "UUID"
}
```

A execução dessas etapas no sistema geram registros nas entidades *Role*, *User* e *UserRole*, discutidas anteriormente nasubseção 4.2.1 *Modelo de Dados*

5.2.3 Gerenciamento de Produtos, Categorias e Menus

O fluxo de atividades demonstrado na Figura 9 ilustra as atividades que podem ser realizadas pelos usuários administradores ou operadores de uma loja para o cadastro de produtos, categorias e menus.

Figura 9 – Fluxo de Atividades 3



Fonte: A autora (2024)

Conforme demonstrado, a primeira etapa envolve a realização do login de um usuário autorizado (podendo este ser administrador ou operador da loja). Posteriormente, com o *token* de acesso deste usuário obtido como retorno da requisição de login, é possível efetuar uma solicitação de criação de um produto, no formato apresentado a seguir.

- **Criar um produto:**

Formato da requisição de criação de um produto

```
POST  /store/{store_id}/products

Body: {
   "name": "string"
}

Response: 201 {
   "name": "string",
   "id": UUID,
   "defaultPrice": nul
```

Após a criação de um ou mais itens, é possível efetuar operações para cadastrar opções e adicionais de produtos. As opções correspondem a variações, como tamanhos, cores ou ingredientes, enquanto os adicionais incluem complementos que podem ser escolhidos pelo cliente, como bacon, queijo, molhos, entre outros. Esses complementos são associados manualmente a cada produto da loja pelo próprio usuário.

A seguir, serão apresentados os devidos formatos esperados pela aplicação para a execução dessas operações.

- **Criar uma opção para o produto:**

Formato da requisição de criação de opções de produtos

```
POST
/store/{store_id}/products/{product_id}/options

Body: {
   "name": "string",
   "type": "string",
   "required": boolean,
   "values": [
      {"name": "string", "value": "string",
         "price": number},
   ]}

Response: 201 {
   "name": "string",
   "type": "string",
   "required": boolean,
   "product": {
      "id": "UUID",
      "name": "string",
      "defaultPrice": null
   },
   "id": "UUID"
}
```

- **Criar adicionais de produtos:**

Formato da requisição de criação de adicionais

```
POST
/store/{store_id}/add-ons

Body: {
   "name": "string"
}

Response: 201 {
   "name": "string",
   "id": "UUID"
}
```

- **Vincular adicional ao produto:**

Formato da requisição para vincular adicionais aos produtos

```
POST
/store/{store_id}/products/{product_id}/add-ons

Body: {
   "addOnId": "UUID",
   "price": number
}

Response: 201 {
   "id": "UUID",
   "name": "string",
   "defaultPrice": null
}
```

Com a execução dessas operações, é possível executar as operações representadas pelas quatro últimas etapas ilustradas no fluxo, que abrangem a criação de categorias e dos menus de produtos, conforme demonstrado a seguir.

- **Criar uma categoria:**

Formato da requisição para criação de categorias

```
POST /store/{store_id}/categories

Body: {
   "name": "string",
}

Response: 201 {
   "name": "string",
   "id": "UUID"
}
```

- **Adicionar produtos a categoria:**

Formato da requisição para adicionar produtos as categorias

```
POST /store/{store_id}/categories/{category_id}
/products

Body: {
   "productId": "UUID"
}

Response: 201 {
   "product": {
      "id": "UUID",
      "name": "string,
      "defaultPrice": null
   },
   "category": {
      "id": "UUID",
      "name": "string"
   },
   "id": "UUID"
}
```

- **Criar um menu:**

Formato da requisição para criação do menu

```
POST /store/{store_id}/product-sets

Body: {
   "name": "string",
   "status": "active"
}

Response: 201 {
   "name": "string",
   "status": "string",
   "store": {
      "id": "UUID"
   }}
```

- **Adicionar itens ao menu:**

Formato da requisição para adicionar itens no menu

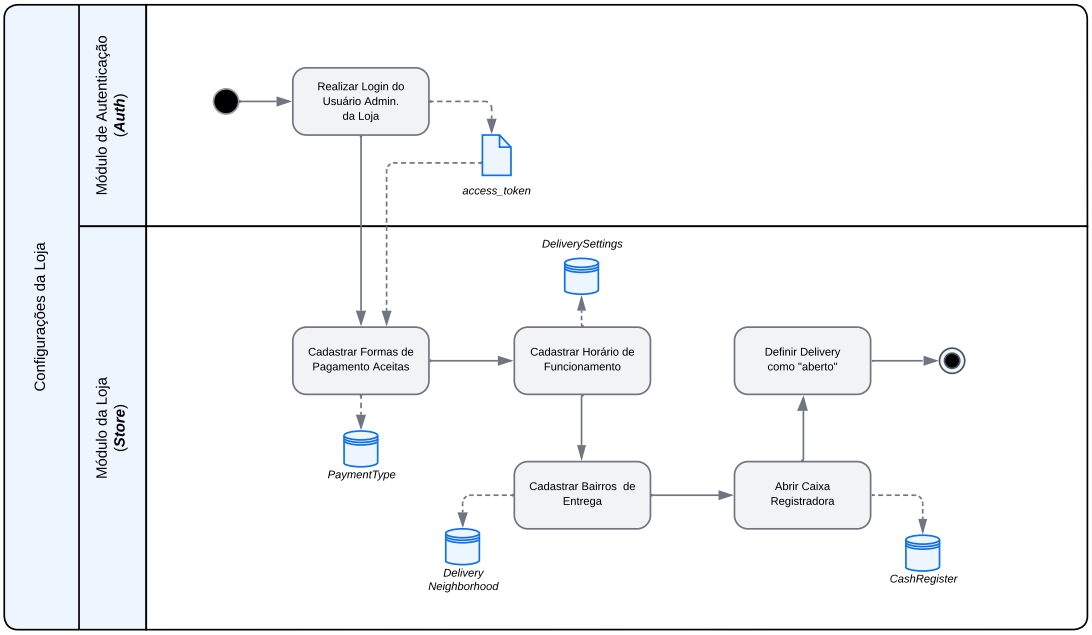
```
POST /store/{store_id}/productsets
/{product_set_id}/products

Body: {
   "productId": "UUID",
   "price": number
}

Response: 201 {
   "price": number,
   "productSet": {
      "id": "UUID"
   },
   "product": {
      "id": "UUID"
   },
   "id": "UUID"
}
```

Essas operações geram registros nas entidades *ProductOption*, *ProductOptionValue*, *AddOn* e *ProductAddOn*, discutidas anteriormente nasubseção 4.2.1 *Modelo de Dados*.

5.2.4 Personalização de Configurações Específicas da Loja

A Figura 10 ilustra o fluxo de atividades para personalizar configurações específicas de uma loja, incluindo as formas de pagamento aceitas, os horários de funcionamento e os bairros disponíveis para entrega.

Figura 10 – Fluxo de Atividades 4



Fonte: A autora (2024)

Após realizar o login de um usuário administrador da loja, ou de qualquer usuário com as permissões adequadas, é possível realizar as quatro etapas subsequentes ilustradas no diagrama intituladas '*Cadastrar Formas de Pagamento*', '*Cadastrar Horários de Funcionamento*", "*Cadastrar Bairros de Entrega*', *Abrir Caixa Registradora* e "*Abrir Delivery*".

- **Definir horário de funcionamento:**

Formato da requisição para definir horário

```
PATCH /store/{store_id}/delivery-settings

Body: {
   "startHour": "2024-12-09 18:30:23.739",
   "endHour": "2024-12-09 23:30:23.739"
}

Response: 201 {
   "id": "UUID",
   "startHour": "2024-12-09 18:30:23.739",
   "endHour": "2024-12-09 23:30:23.739",
   "status": "string"
}
```

- **Cadastrar formas de pagamento:**

As formas de pagamento disponíveis são definidas programaticamente no arquivo intitulado '*payment-type.entity.ts*' do projeto, sendo necessário seguir o padrão estabelecido nesse arquivo.

Formato da requisição de cadastro de formas de pagamento

```
POST /store/{store_id}/payment-types
Body: {
   "name": "string",
   "type": "string"
}

Response: 201 {
   "name": "string",
   "type": "string",
   "store": {
      "id": "UUID"
   },
   "id": "UUID"
}
```

- **Cadastrar bairros de entrega:**

Formato da requisição de cadastro dos bairros de entrega

```
POST
/store/{store_id}/delivery-settings/neighborhoods

Body: {
   "neighborhoodName": "string",
   "neighborhoodCode": "string",
   "deliveryFee": number
}

Response: 201 {
   "neighborhoodCode": "string",
   "neighborhoodName": "string",
   "deliveryFee": number,
   "deliverySettings": {
      "id": "UUID"
   },
   "id": "UUID"
}
```

- **Abrir *delivery*:**

Formato da requisição para definir *delivery* como aberto

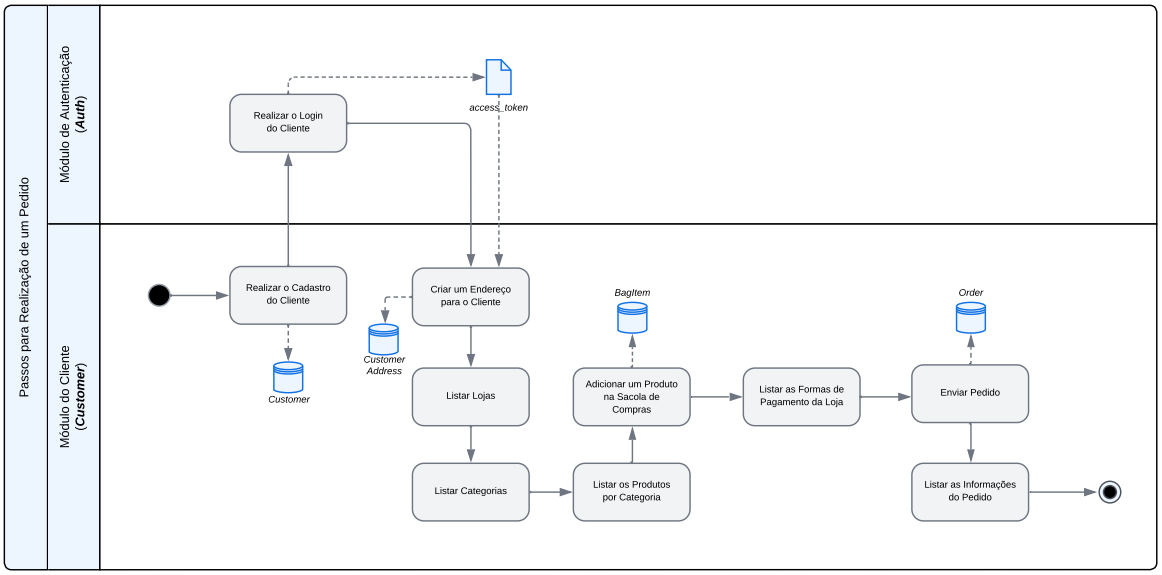
```
PATCH
/store/{store_id}/delivery-settings/open

Response: 200 OK
```

A execução de todas essas etapas geram registros nas entidades *PaymentType*, *Delivery-Settings*, *DeliveryNeighborhood* e *CashRegister*, as quais foram previamente apresentadas e discutidas nasubseção 4.2.1 *Modelo de Dados*.

5.2.5 Realização de um pedido por um Cliente

O diagrama de fluxo de atividades demonstrado na Figura 11 ilustra, de forma sequencial, as etapas previstas na realização de um pedido por um cliente em uma loja do sistema.

Figura 11 – Fluxo de Atividades 5



Fonte: A autora (2024)

Na primeira etapa intitulada *Realizar o Cadastro do Cliente*', uma conta para o cliente é criada no sistema. Subsequentemente, o login do cliente é realizado, gerando um *token* de acesso exclusivo vinculado a ele, o qual serve para executar as próximas etapas ilustradas no diagrama.

- **Cadastrar uma nova conta de cliente:**

Formato da requisição de cadastro do cliente

```
POST /customer/accounts

Body: {
   "name": "string",
   "email": "string",
   "password": "string"
}

Response 201 {
   "name": "string",
   "email": "string",
   "id": "UUID"
}
```

A etapa subsequente envolve a realização do login do cliente, e após a sua execução é possível criar um endereço para ele, o qual será posteriormente utilizado na penúltima etapa, intitulada '*Enviar pedido*'. Vale destacar que um cliente pode ter mais de um endereço vinculado a ele.

- **Vincular um endereço a conta do cliente:**

Formato da requisição de criação de endereços de clientes

```
POST /customer/accounts/addresses

Body: {
   "street": "string",
   "number": "number",
   "zipCode": "string",
   "neighborhoodCode": "string"
}

Response: 201 {
   "street": "string",
   "number": "number",
   "zipCode": "string",
   "neighborhoodCode": "string",
   "id": "UUID",
   "complement": null
}
```

As próximas etapas envolvem a navegação pelos produtos da loja organizados por categorias, a adição de itens à sacola de compras, a consulta das formas de pagamento disponíveis e, por fim, o envio do pedido, seguido pela visualização das informações do pedido, como o status.

- **Listar lojas disponíveis para o cliente:**

Formato da requisição para listar as lojas

```
GET /customer/stores

Response: 200 {
   [{
      "id": "UUID"
      "name": "string"
      "startHour": "2024-12-09T21:30:23.739Z",
      "endHour": "2024-12-10T02:30:23.739Z",
      "status": "open"
   }]
}
```

- **Listar categorias da loja:**

Formato da requisição para listar categorias

```
GET /customer/stores/{store_id}/categories

Response: 200 {
   [{
      "id": "UUID"
      "name": "string"
   }]
}
```

- **Visualizar produtos da categoria:**

Formato da requisição para listar produtos por categoria

```
GET /customer/stores/{store_id}/categories
/{category_id}/products

Response: 200 {
    "id": "UUID",
    "name": "string",
    "price": number,
    "addOns": [{
       "id": "UUID",
       "name": "string",
       "price": number
     }],
    "options": [{
       "id": "UUID",
       "name": "string",
       "type": "string",
       "required": boolean,
       "values": [{
          "id": "UUID",
          "name": "string",
          "value": "string",
          "price": number }]
     }]
  }
```

- **Adicionar produto à sacola de compras:**

Formato da requisição de adição de itens à sacola de compras

```
POST /customer/stores/{store_id}/bag/items

Body: {
 "productSetItemId": "string",
 "quantity": number,
 "options": [{
    "productOptionId": "string",
    "optionValueId": "string"
  }],
 "addOns": [{
    "productAddOnId": "string",
    "quantity": number
  }]
}

Response: 201 OK
```

- **Listar formas de pagamento:**

Formato da requisição para listar formas de pagamento

```
GET /customer/stores/{store_id}}/payment-types

Response: 200 {
   "content": [{
        "id": "UUID",
        "name": "string",
        "type": "string"
     }]
```

- **Enviar pedido:**

Formato da requisição de envio de um pedido

```
POST /customer/stores/{store_id}/orders

Body: {
   "addressId": "UUID",
   "preferredPaymentTypeId": "UUID",
   "observation": "string"
}

Response: 200 OK
```

- **Visualizar as informações do pedido:**

Formato da requisição de visualização das informações do pedido

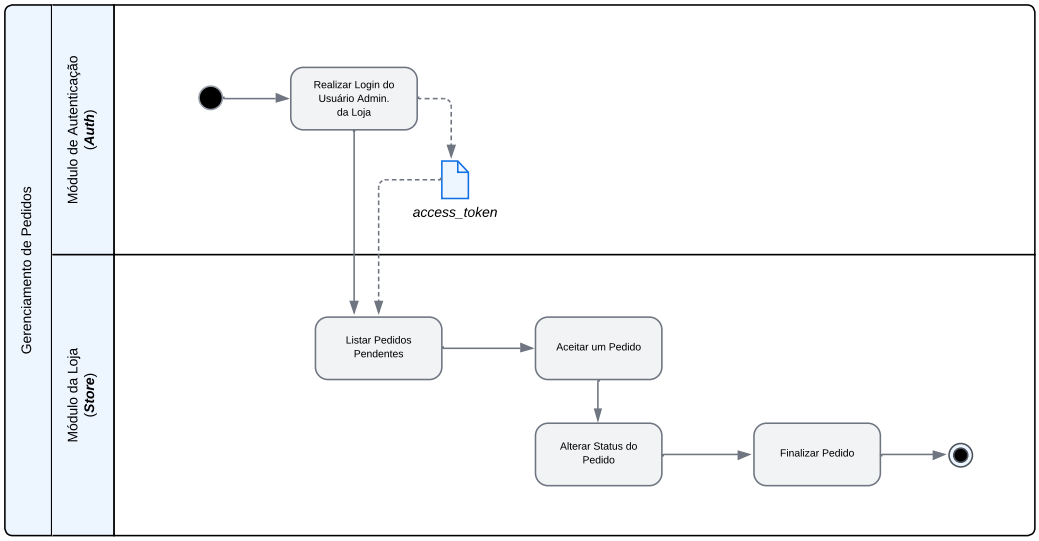
```
GET /customer/stores/{store_id}/orders

Response: 200 {
   "content": [
      {
         "id": "UUID",
         "date": "Date",
         "bagPrice": number,
         "shippingPrice": number,
         "totalPrice": number,
         "observation": "string",
         "status": "string",
         "payments": [],
         "shippings": [{
               "id": "6c209649-d607...",
               "recipientName": "string",
               "status": "string",
               "price": number
            }]
         }]
      }
```

A execução dessas etapas no sistema geram registros nas entidades *Customer*, *CustomerAddress*, *BagItem* e *Order*, as quais foram previamente apresentadas e discutidas na subseção 4.2.1 *Modelo de Dados*.

5.2.6 Gerenciamento de Pedidos da Loja

A Figura 12 apresenta o fluxo de atividades relacionado ao gerenciamento de pedidos de uma loja, abrangendo operações como a listagem de pedidos, aceitação de pedidos e atualização do status de um pedido.

Figura 12 – Fluxo de Atividades 6



Fonte: A autora (2024)

Conforme demonstrado, a primeira etapa envolve a realização do login de um usuário autorizado (podendo este ser administrador ou operador da loja). Posteriormente, com o *token* de acesso deste usuário obtido por meio da resposta da requisição de login, é possível executar as quatro etapas subsequentes intituladas ’Listar Pedidos Pendentes’, ’Aceitar um Pedido’, ’Alterar Status do Pedido’ e ’Finalizar Pedido’. Essas operações podem ser realizadas seguindo os formatos apresentados a seguir.

- **Listar pedidos**

Formato da requisição para listar os pedidos pendentes

```
GET /store/{store_id}/orders

Body: {
   "status": "PENDING"
}

Response: 200 {
   "content": [{
         "id": "UUID",
         "date": Date,
         "bagPrice": number,
         "shippingPrice": number,
         "totalPrice": number,
         "observation": "string",
         "status": "string"
      }]
}
```

- **Listar detalhes de um pedido:**

Formato da requisição para listar os detalhes de um pedido

```
GET /store/{store_id}/orders/{order_id}

Response: 200 {
   "id": "UUID",
   "bagPrice": number,
   "shippingPrice": number,
   "totalPrice": number,
   "observation": "string",
   "status": "string",
   "paymentType": "string",
   "items": [{
      "id": "UUID",
      "name": "string",
      "quantity": number,
      "unitPrice": number,
      "options": [{
         "id": "UUID",
         "name": "string",
         "value": {
            "id": "UUID",
            "name": "string",
            "price": number }
      }],
      "addOns": [{
         "id": "UUID",
         "quantity": number,
         "name": "string"
      }]
   }]
}
```

- **Aceitar pedido:**

Formato da requisição de aceitação de pedidos

```
GET /store/{store_id}/orders/{order_id}/accept

Body: {
   "cashRegisterId": "UUID"
}

Response: 200 {
   "id": "UUID",
   "date": "Date",
   "bagPrice": number,
   "shippingPrice": number,
   "totalPrice": number,
   "observation": "string",
   "status": "string",
   "cashRegister": {
      "id": "UUID",
      "openingDate": "Date",
      "closingDate": null }
}
```

- **Atualizar status do pedido como "entregando":**

Formato da requisição de atualização do status de um pedido

```
PATCH /store/{store_id}/orders/{order_id}/shipping

Response: 200 {
   "id": "UUID",
   "date": "Date",
   "bagPrice": number,
   "shippingPrice": number,
   "totalPrice": number,
   "observation": "string",
   "status": "string"
}
```

- **Definir pedido como "finalizado":**

Formato da requisição para finalizar um pedido

```
PATCH /store/{store_id}/orders/{order_id}/finish

Body: {
 "payments": [{
   "paymentTypeId": "string",
   "value": number
  }]
}

Response: 200 OK
```

6 RESULTADOS E DISCUSSÕES

Neste capítulo, serão apresentados os principais resultados obtidos com a criação do protótipo do sistema proposto neste trabalho.

6.1 ANÁLISE GERAL DO SISTEMA

Na etapa de planejamento, o processo de coleta de requisitos desempenhou um papel fundamental, garantindo uma sólida compreensão das necessidades essenciais de diferentes aplicações de software comerciais. A execução dessa etapa possibilitou que o desenvolvimento do protótipo do sistema seguisse uma base bem estruturada, facilitando as etapas subsequentes e reduzindo os riscos de desvios no escopo.

Para o desenvolvimento, optou-se por utilizar o framework *NestJS*, que se destaca por facilitar a criação de aplicações modulares devido ao seu design baseado em módulos. Essa abordagem organiza o código de maneira estruturada e desacoplada, permitindo que cada módulo seja desenvolvido, mantido e escalado de forma independente. Essa característica do sistema foi essencial para atender ao objetivo geral do trabalho, pois contribui para a criação de um código de *backend* de fácil manutenção, proporcionando um ambiente favorável para extensibilidade de suas funcionalidades. Por outro lado, a definição das entidades de forma genérica também foi um fator determinante para garantir que o sistema pudesse ser reutilizado em diversos domínios.

A implementação de um modelo de dados *multi-tenancy* destaca-se como um dos aspectos significativos para atingir o propósito do sistema de atender a múltiplas aplicações comerciais. Esse recurso permite o compartilhamento eficiente da infraestrutura entre diferentes aplicações e, dessa forma, garante que diferentes aplicações possam fazer o uso dos recursos disponibilizados pela mesma infraestrutura de maneira independente, sem riscos de interferência.

Adicionalmente, a adoção dos padrões *REST* no desenvolvimento da *API* trouxe benefícios importantes ao proporcionar uma interface uniforme e padronizada, tornando-a mais intuitiva e fácil de ser compreendida. Essa padronização não apenas melhora a experiência de quem utiliza a *API*, mas também facilita que outros desenvolvedores contribuam para a expansibilidade do sistema e seu desenvolvimento futuro.

Em suma, essas características fundamentais do sistema demonstram que o seu desenvolvimento foi realizado de forma alinhada ao objetivo de criar uma ferramenta capaz de agilizar a criação e entrega de aplicações comerciais. Nas seções seguintes, serão discutidos os recursos implementados e sua aplicabilidade a outros contextos comerciais, evidenciando o potencial expansivo do sistema.

6.2 FUNCIONALIDADES IMPLEMENTADAS

O desenvolvimento do protótipo resultou em um conjunto robusto de recursos e funcionalidades voltados ao suporte de diferentes tipos de aplicações comerciais, apesar do foco inicial no domínio de *delivery*. Essas funcionalidades foram projetadas para serem reutilizáveis em outros contextos comerciais, reforçando a versatilidade e a escalabilidade do sistema.

O sistema inclui um gerenciamento completo de usuários, permitindo administrar diferentes tipos, como administradores, operadores de lojas e clientes finais. De modo complementar, foi implementada uma funcionalidade para possibilitar o gerenciamento de papéis de usuários, permitindo que permissões específicas sejam agrupadas e posteriormente vinculadas aos usuários ao serem criados. Dessa forma, o sistema permite a personalização de permissões de acordo com as necessidades específicas de cada organização ou loja do sistema. A autenticação dos usuários é realizada por meio de métodos seguros baseados em *tokens JWT*, garantindo controle de acesso eficiente e seguro às funcionalidades do sistema.

No módulo de organização (*organization*), foram implementadas funcionalidades relativas ao gerenciamento de lojas, possibilitando a criação e configuração de múltiplas entidades comerciais de maneira flexível e escalável. Já o módulo da aplicação referente às lojas (*store*), foram implementadas funções associadas ao gerenciamento e administração de recursos essenciais para o funcionamento online de um estabelecimento comercial, como produtos, categorias, menus e pedidos. Além disso, oferece ferramentas para personalizar configurações específicas de cada estabelecimento, incluindo formas de pagamento, cadastro de endereços de entrega e definição de horários de funcionamento.

Para os clientes, o sistema agrupa funcionalidades como o cadastro, a navegação por menus e categorias de produtos, a personalização de pedidos e envio de solicitações às lojas.

Essas funcionalidades, quando integradas, evidenciam a capacidade do sistema de atender a diferentes contextos comerciais, cumprindo seu objetivo de oferecer uma solução genérica e adaptável às necessidades de variados domínios comerciais.

6.2.1 Contextos Comerciais Adicionais e Recursos Faltantes

Com base nas funcionalidades que foram implementadas, é possível identificar outros domínios comerciais nos quais o sistema pode ser aplicado, bem como as lacunas que precisam ser preenchidas para atender integralmente às necessidades de cada contexto.

No segmento de *e-Commerces*, várias funcionalidades do sistema podem ser reutilizadas, como o gerenciamento de usuários e autenticação, o gerenciamento de recursos e configurações específicas de uma loja e o gerenciamento de recursos pelos clientes. Contudo, para atender plenamente às demandas desse contexto, seria necessário implementar funcionalidades adicionais,

como gestão de estoque com notificações de baixa quantidade, integração com *gateways* de pagamento (como *PayPal* e *Stripe*), cálculo de frete baseado em peso e dimensões, além de ferramentas para gestão de promoções e cupons de desconto personalizados.

Já para o domínio de *Restaurantes e Serviços de Alimentação*, as funcionalidades reutilizáveis incluem o gerenciamento de usuários e autenticação, bem como o gerenciamento de recursos e configurações específicas de uma loja. No entanto, para atender completamente às necessidades desse segmento, seria importante implementar recursos como o gerenciamento de mesas para pedidos realizados no local e a integração com dispositivos de cozinha, como impressoras de comandas.

No caso de *Supermercados e Minimercados Online*, o sistema também apresenta um conjunto de funcionalidades reutilizáveis, como o gerenciamento de usuários e autenticação, o gerenciamento de recursos e configurações de lojas, e o gerenciamento de recursos pelos clientes finais. Para adaptar o sistema a esse segmento, seriam necessárias implementações adicionais, como uma ferramenta de análise de compras recorrentes para sugestões automáticas, integração com *gateways* de pagamento e gestão de promoções e cupons de desconto personalizados.

Esses novos contextos e funcionalidades destacam o potencial expansivo do sistema para além do domínio inicial de *delivery*. Apesar de algumas lacunas pontuais, a arquitetura genérica, baseada em *multi-tenancy* e nos princípios *REST*, torna o sistema facilmente extensível para atender a novos segmentos de aplicações de software comerciais. Essa flexibilidade reafirma o valor do sistema como sendo um *backend* genérico e adaptável, capaz de suportar uma ampla gama de aplicações comerciais.

7 CONSIDERAÇÕES FINAIS

Tendo em vista o objetivo geral do presente trabalho, que consiste em "*desenvolver o protótipo de um sistema de backend genérico, capaz de fornecer, via API REST, recursos e funcionalidades aplicáveis ao desenvolvimento de aplicações de software comerciais, com foco inicial em plataformas de delivery*", pode-se afirmar, fundamentando-se no exposto no capítulo 5 Demonstração e no capítulo 6 Resultados e Discussões, que este objetivo foi atingido.

A seguir, ao que se refere aos objetivos específicos que orientaram o desenvolvimento do sistema:

- *Explorar os fundamentos do padrão arquitetural REST, com ênfase em seus princípios mais relevantes para o desenvolvimento de web services* - O objetivo proposto foi alcançado, com suas evidências demonstradas no capítulo 2 *Referencial Teórico*, demonstradas no capítulo 5 *Demonstração* e corroboradas pela documentação da *API*, que se encontra publicamente disponível no *SwaggerHub*.

- *Realizar o levantamento dos requisitos para o desenvolvimento do sistema, os diagramas de caso de uso e o modelo de dados* - Conforme a descrição dos requisitos apresentada no capítulo 4, *Projeto e Desenvolvimento*, entende-se que este objetivo foi alcançado;

- *Descrever os principais fluxos de requisições previstos na utilização da ferramenta* - Verifica-se que o objetivo foi atingido com a apresentação e o detalhamento dos fluxos propostos no capítulo 5 *Demonstração*;

- *Verificar a aplicabilidade dos recursos implementados no sistema para outros domínios de aplicações de software comerciais* - O objetivo foi alcançado e suas evidências podem ser conferidas com o exposto no seção 6.2 *Funcionalidades Implementadas*, bem como na seção subsequente reservada para o tema.

O sistema de *backend* genérico para aplicações de software comerciais desenvolvido neste trabalho representa uma contribuição significativa para o desenvolvimento ágil de softwares voltados ao mercado comercial. A implantação do sistema em um ambiente de produção, aliada à disponibilização pública da documentação da *API* do sistema, possibilita a reutilização dos recursos e funcionalidades implementadas(os). Dessa forma, o sistema pode servir como uma base sólida para o funcionamento de interfaces de usuário desenvolvidas especificamente para o domínio comercial de *delivery*, reduzindo fatores como o tempo e os custos associados ao desenvolvimento completo de soluções comerciais integradas.

O sistema também apresenta grande potencial para continuidade e expansão, permitindo a implementação de novas funcionalidades que atendam a outros domínios comerciais de aplicações de software. Esse potencial de evolução é viabilizado pela arquitetura modular e escalável do sistema, aliada à documentação detalhada da *API*, que fornece uma base clara e acessível para desenvolvedores interessados em contribuir para o projeto. A implantação do sistema em um ambiente de produção garante que essas futuras implementações possam ser testadas e integradas em um contexto realista, assegurando sua eficácia e funcionalidade. Dessa forma, o sistema se posiciona como uma solução não apenas para o domínio inicial de *delivery*, mas também como uma ferramenta adaptável e expansível para diferentes segmentos comerciais.

Embora o sistema tenha demonstrado eficácia em cumprir seu propósito como uma solução de *backend* genérica e expansível, entende-se que a ausência de uma integração com um sistema de *frontend* restringiu a possibilidade de avaliação em cenários mais realistas. Essa limitação impediu uma análise aprofundada da interação usuário-sistema, que seria fundamental para avaliar a usabilidade e a eficiência do sistema em condições práticas de utilização. A integração com um *frontend* permitiria explorar de forma mais abrangente o comportamento do sistema quando utilizado por usuários finais, identificando potenciais melhorias tanto na experiência do usuário quanto na comunicação entre as camadas de software. Essa é uma sugestão relevante para trabalhos futuros que possibilita uma avaliação mais completa e detalhada da solução proposta.

Entende-se, ainda, que há oportunidades para aprimorar a eficácia do sistema no contexto de aplicações de *delivery* em trabalhos futuros, por meio da implementação de funcionalidades adicionais que agregariam ainda mais valor. Entre elas, destaca-se o gerenciamento de estoque e vendas, juntamente com a inclusão de *dashboards* gerenciais, que poderiam apresentar informações financeiras, dados de vendas e indicadores-chave de desempenho (*KPI's*), como receita total, volume médio de pedidos, produtos mais vendidos e comparativos de vendas. Esses recursos ofereceriam *insights* estratégicos valiosos, facilitando a tomada de decisões baseadas em dados e otimizando as operações comerciais.

Adicionalmente, algumas sugestões de melhorias futuras para abranger novos domínios comerciais incluem a integração com sistemas de logística em tempo real, permitindo o rastreamento detalhado de entregas, e a implementação de mecanismos avançados de personalização, como recomendações baseadas no histórico de pedidos dos clientes. Essas implementações posicionariam o sistema como uma solução ainda mais versátil e competitiva para diferentes mercados de aplicações comerciais.

Referências

ALLAMARAJU, S. *Restful web services cookbook: solutions for improving scalability and simplicity*. "O'Reilly Media, Inc.", 2010. Disponível em: <https://pepa.holla.cz/wp-content/uploads/2016/01/RESTful-Web-Services-Cookbook.pdf>. Citado na página 20.

ALSAADI, H. A. et al. Factors that affect the utilization of low-code development platforms: survey study. *Romanian Journal of Information Technology & Automatic Control/Revista Română de Informatică și Automatică*, v. 31, n. 3, 2021. Disponível em: <https://pdfs.semanticscholar.org/4f18/561e75945c083924b014d6cfe297687f55f5.pdf>. Citado na página 11.

BARAN, B. E.; WOZNYJ, H. M. Managing vuca: The human dynamics of agility. *Organizational dynamics*, Elsevier, 2020. Disponível em: <https://pmc.ncbi.nlm.nih.gov/articles/PMC7439966/>. Citado na página 10.

BERNERS-LEE, T.; FIELDING, R.; MASINTER, L. *Uniform resource identifier (URI): Generic syntax*. [S.l.], 2005. Disponível em: <https://datatracker.ietf.org/doc/html/rfc3986#autoid-19>. Citado na página 18.

BOOCH, G. *The Unified Modeling Language User Guide*. [S.l.]: Pearson Education, 2005. (The Addison-Wesley object technology series). ISBN 9788131715826. Citado na página 31.

CASSIOLATO, J. E.; SILVA, A. L. G. da. *Telecomunicações, globalização e competitividade*. [S.l.]: Papirus Editora, 1995. Citado na página 10.

CHICHINELLI, M. A importância das técnicas de levantamento de requisitos no processo de desenvolvimento de software. *Revista Empreenda UniToledo Gestão, Tecnologia e Gastronomia*, v. 1, n. 1, 2017. Citado na página 25.

FIELDING, R.; RESCHKE, J. *Hypertext transfer protocol (HTTP/1.1): Semantics and content*. [S.l.], 2014. Disponível em: <https://datatracker.ietf.org/doc/html/rfc7231#section-4.3.1>. Citado na página 20.

FIELDING, R. T. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000. Disponível em: <https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf>. Citado 3 vezes nas páginas 16, 17 e 20.

JUNKES, G. d. S. Evolução da tecnologia da informação e comunicação (tic) e seus benefícios para as empresas. 2014. Disponível em: <http://repositorio.unesc.net/handle/1/2879>. Citado na página 10.

NestJS. *NestJS Documentation*. [S.l.], 2024. Disponível em: <https://docs.nestjs.com/>. Citado na página 37.

NGUYEN, P. Mobile backend as a service: the pros and cons of parse. Lahden ammattikorkeakoulu, 2016. Citado na página 11.

PEFFERS, K. et al. A design science research methodology for information systems research. *Journal of management information systems*, Taylor & Francis, v. 24, n. 3, p. 45–77, 2007. Disponível em: <https://doi.org/10.2753/MIS0742-1222240302>. Citado na página 22.

RICHARDSON, L.; RUBY, S. *RESTful Web Services*. Gravenstein Highway North, Sebastopol, CA: O'Reilly Media, Inc, 2007. Disponível em: <https://pepa.holla.cz/wp-content/uploads/2016/01/RESTful-Web-Services.pdf>. Citado 4 vezes nas páginas 17, 18, 19 e 20.

SANCHIS, R. et al. Low-code as enabler of digital transformation in manufacturing industry. *Applied Sciences*, MDPI, v. 10, n. 1, p. 12, 2019. Disponível em: <https://www.mdpi.com/2076-3417/10/1/12>. Citado na página 10.

SOMMERVILLE, I. *Engenharia de software, 9a*. Pearson Education do Brasil, 2011. Disponível em: <https://www.facom.ufu.br/~william/Disciplinas%202018-2/BSI-GSI030-EngenhariaSoftware/Livro/engenhariaSoftwareSommerville.pdf>. Citado 3 vezes nas páginas 10, 14 e 31.

XAVIER, F. C. *Transformação Digital: uma jornada para a competitividade e inovação*. 2023. Disponível em: <https://mittechreview.com.br/transformacao-digital-uma-jornada-para-a-competitividade-e-inovacao/#:~:text=A%20Transforma%C3%A7%C3%A3o%20Digital%20pode%20trazer,a%20novos%20modelos%20de%20neg%C3%B3cios>. Citado na página 10.