Desenvolvimento de uma API REST para Gestão de Vendas com Controle de Débitos e Pagamentos em Microcomércios

Antonio Ferreira [*1], **Douglas Nemézio** [†1], **Elton Silva** [‡1], **Julia Alves** [§1], **Matheus Bastos** [¶ 1], **Angela Peixoto** [‖1*]

[1] Análise e Desenvolvimento de Sistemas
Escola de Tecnologias
Universidade Católica do Salvador (UCSAL)
Av. Prof. Pinto de Aguiar, 2589 Pituaçu, CEP: 41740-090
Salvador/BA, Brasil
[1] *{antoniovitor.ferreira,douglas.sousa,eltonsilva.santos,, julia.costa,matheusbastos.barbosa}@ucsal.edu.br*
[1*] *{angela.santana,coorientador}@pro.ucsal.edu.br*

Junho 2026

[*]antoniovitor.ferreira@ucsal.edu.br
[†]douglas.sousa@ucsal.edu.br
[‡]eltonsilva.santos@ucsal.edu.br
[§]julia.costa@ucsal.edu.br
[¶]matheusbastos.barbosa@ucsal.edu.br
[‖]angela.santana@pro.ucsal.br

Resumo

Este trabalho surgiu a partir da identificação de dificuldades enfrentadas no controle manual de vendas, débitos de clientes e faturamento, o que tornava o processo mais demorado e sujeito a falhas. O trabalho apresenta o desenvolvimento de um sistema *web* para gerenciamento de vendas, débitos de clientes e faturamento em microcomércios. A aplicação foi desenvolvida utilizando Java, Spring Boot e Angular, com arquitetura baseada em *API REST*. O sistema possibilita o cadastro de clientes, cadastro de produtos, registro de vendas, controle de débitos e acompanhamento financeiro, oferecendo maior organização e apoio à gestão do negócio.

Palavras-chave: microcomércio; controle de vendas; sistema *web*; débitos; gestão financeira

Abstract

This study emerged from the identification of difficulties faced in the manual control of sales, customer debts, and revenue, which made the process more time-consuming and prone to errors. The work presents the development of a *web* system for managing sales, customer debts, and revenue in microbusinesses. The application was developed using Java, Spring Boot, and Angular, with an architecture based on a *REST API*. The system enables customer registration, product registration, sales recording, debt control, and financial monitoring, providing better organization and support for business management.

Keywords: microbusiness; sales control; web system; customer debts; financial management.

1 Introdução

O trabalho propõe o desenvolvimento de um sistema para auxiliar no controle de dívidas de clientes em um microcomércio, além de permitir o registro e acompanhamento das vendas realizadas e do faturamento obtido no mês. A proposta surgiu a partir da identificação de um problema real enfrentado por uma empreendedora, que encontra dificuldades e demora para calcular o valor mensal das dívidas de seus clientes, bem como para manter um controle organizado das vendas realizadas.

Em um cenário em que a tecnologia tem sido cada vez mais utilizada para apoiar a gestão de pequenos negócios, muitos microcomércios ainda realizam o controle de vendas e dívidas de forma manual, utilizando cadernos, anotações ou planilhas simples. Essa prática pode dificultar a visualização das informações financeiras e comprometer a tomada de decisões, especialmente quando há um volume maior de clientes e transações. Nesse contexto, o uso de sistemas de informação torna-se uma alternativa importante para otimizar processos, reduzir falhas e proporcionar maior eficiência no gerenciamento do negócio.

A ausência de um sistema para registrar vendas e acompanhar as dívidas dos clientes pode gerar dificuldades na organização financeira do negócio, além de aumentar a possibilidade de erros nos cálculos e no controle das informações. Dessa forma, torna-se importante a utilização de ferramentas que auxiliem na gestão dessas informações de maneira mais rápida e organizada.

Diante desse cenário, este trabalho tem como objetivo desenvolver um sistema *web* capaz de registrar vendas, controlar dívidas de clientes e fornecer informações sobre o faturamento do microcomércio, facilitando o acompanhamento das transações realizadas.

Nesse sentido, a adoção de um sistema contribui para a organização e o controle das informações do negócio. A ideia é desenvolver uma aplicação *web* utilizando *Java, Spring Boot, API REST e Angular*, que permita registrar as vendas de forma mais simples e acompanhar as dívidas dos clientes de forma mais clara. Com isso, a pessoa empreendedora consegue ter mais controle sobre o que acontece no dia a dia, evita erros nos cálculos e ainda economiza tempo, já que não precisa fazer tudo manualmente. Além disso, o sistema facilita a visualização do faturamento, ajudando na tomada de decisões para melhorar o desempenho do negócio.

Além dessa introdução, o artigo está organizado da seguinte forma: a seção 2 apresenta a revisão da literatura; a seção 3 apresenta o desenvolvimento do sistema e a seção 4 aponta a conclusão e trabalhos futuros.

2 Fundamentação Teórica

O presente capítulo tem por finalidade apresentar os fundamentos teóricos que sustentam a realização da pesquisa, abordando os conceitos de sistemas de informação voltados ao comércio, as arquiteturas modernas de desenvolvimento *web* e as tecnologias específicas utilizadas na implementação da solução proposta. A compreensão desses elementos é essencial para fundamentar as decisões técnicas tomadas ao longo do desenvolvimento do projeto de *e-commerce*.

2.1 Sistemas de informação de Microcomercios

O *e-commerce* é um termo eletrônico, no qual as vendas de produtos e serviços de uma organização são realizadas através dos meios digitais. Ainda nesse projeto, Comércio Eletrônico é todo processo de negociação em um ambiente eletrônico, por meio da utilização intensa das tecnologias de comunicação e de informação. Essa comercialização acontece de forma fácil e rápida e busca atender as necessidades de uma pessoa ou sociedade em qualquer lugar do mundo (TWARDOWSKI; RIDER et al., 2023).

O comércio eletrônico já apresentava um crescimento consistente antes da pandemia de COVID-19, porém esse avanço foi intensificado a partir desse período, impulsionado pela praticidade oferecida tanto a vendedores quanto a consumidores no acesso a produtos e serviços (NUNES, 2026). Os sistemas de informação dentro dos microcomércios têm um papel fundamental, já que ele permite o controle de estoque, de vendas e toda a estrutura organizacional dentro de um microcomércio.

Nesse mesmo contexto, (TWARDOWSKI; RIDER et al., 2023) afirma que o *e-commerce* possibilitou ao comércio atual uma nova dimensão à medida que proporcionou a comercialização de produtos pelo meio digital. Esse modelo de negócio, permite que pequenos comércios atendam uma quantidade grande de clientes, em diversos locais, em qualquer hora, sem limitação de horários e distância, facilitando a trajetória de compras aos consumidores.

O acesso ao mercado no ambiente do comércio eletrônico brasileiro configura-se como um fenômeno ambivalente: ao mesmo tempo em que amplia horizontes e oportunidades para negócios de diferentes perfis e localizações, também exige um nível crescente de sofisticação técnica, capacidade de adaptação e estratégias para sobrevivência em um ambiente extremamente competitivo e volátil. A construção de um ecossistema digital mais inclusivo, inovador e resiliente depende, assim, tanto da atuação estratégica dos próprios empreendedores quanto do suporte de políticas públicas integradas e da redução das assimetrias tecnológicas e informacionais que ainda caracterizam o cenário brasileiro (MONTEIRO, 2025).

2.2 Arquitetura de *software*

A arquitetura de *software* é a estrutura organizacional de um sistema de *software*, incluindo seus componentes e a relação entre eles. Ela define os padrões e diretrizes para o desenvolvimento de sistemas, garantindo que os componentes funcionem de maneira coesa e eficiente (RAPÔSO et al., 2025).

Segundo Varoto (2002), o processo de arquitetura de *software* compreende as atividades complementares que são necessárias para a construção do *software*, entendendo que tais atividades servirão para realizar o projeto (*design*), implementar e evoluir o sistema. O produto arquitetural representa o resultado final das atividades que acontecem em cada etapa do processo de definição de uma arquitetura de *software*.

Uma arquitetura bastante usada atualmente é a arquitetura monolítica, que se refere a um modelo de *software* no qual todos os componentes da aplicação são estruturados e implantados como uma única e indivisível unidade. Ainda de acordo com esse projeto, essa abordagem é caracterizada por sua simplicidade conceitual e operacional nos estágios iniciais de um projeto. Uma vez que todo o código-fonte, a lógica de negócio e as funcionalidades estão contidos em um único processo, atividades como o desenvolvimento, a depuração e a implantação inicial são significativamente simplificadas (VALE, 2025).

2.3 API

Uma *API (Application Programming Interface)* é um mecanismo que permite a comunicação entre sistemas, permitindo que aplicativos ou serviços acessem um recurso dentro de outro aplicativo, serviço ou banco de dados. O aplicativo ou serviço que acessa os recursos é o cliente.

De acordo com (TAQUIWI; DIAS; BARBOSA, 2024), as *APIs* são "*interfaces* de programação que tornam os programas capazes de se comunicar com o banco de dados", reforçando seu papel como intermediadoras no acesso e manipulação de informações.

O desempenho das *APIs* também depende da maneira como os dados são modelados e disponibilizados, sendo essencial pensar na definição dos recursos, no versionamento adequado e na escolha dos formatos de mensagem que priorizem a leveza e a eficiência no tráfego, e nesse sentido, o padrão *JSON (JavaScript Object Notation)* vem sendo amplamente adotado por sua simplicidade e compatibilidade com diferentes linguagens, sendo essa prática reconhecida como uma das responsáveis pela disseminação do uso de *REST (Representational State Transfer)* em detrimento de abordagens mais pesadas como *SOAP (Simple Object Access Protocol)*, conforme aponta o estudo de Almeida, ao abordar a experiência da Embrapa com *APIs* (ALMEIDA, 2025).

2.3.1 *API REST - Representational State Transfer*

A interface de comunicação *API REST* é uma tecnologia que utiliza *HTTP (HyperText Transfer Protocol)* e disponibiliza serviços *web*, permitindo a comunicação entre sistemas e dispositivos com um forte poder de interoperabilidade (LEITE; JÚNIOR; FONSECA,).

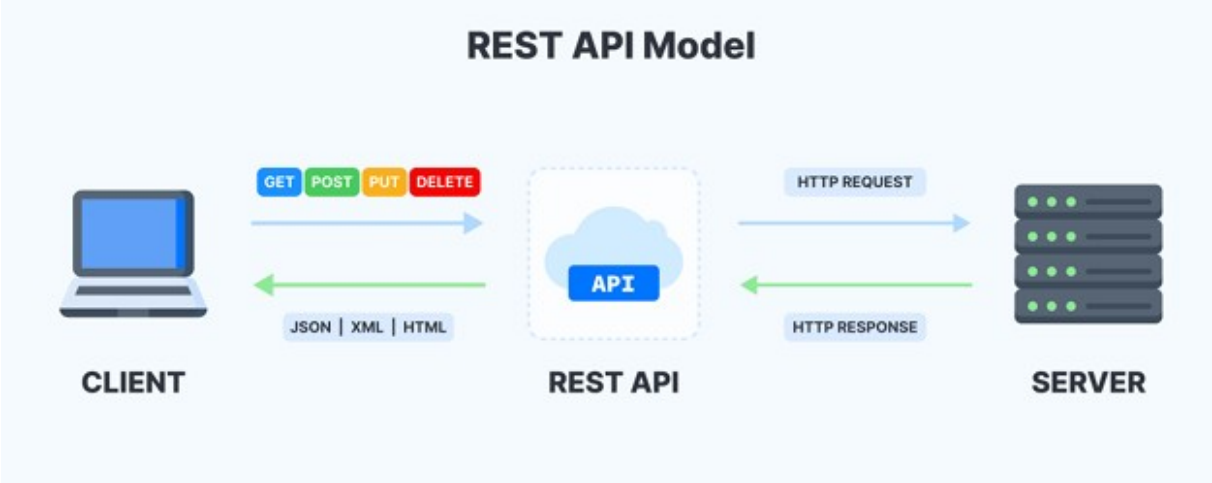
O princípio fundamental do *REST* é o conceito de recurso, que representa qualquer entidade significativa para o sistema, sendo acessada por meio de uma *URI(Uniform Resource Identifier)* única e manipulada por operações *HTTP* que refletem ações sobre esse recurso, o que promove clareza na organização da interface e facilita a documentação, testes e reutilização de código, permitindo que diferentes consumidores interajam com os mesmos dados de forma padronizada e segura, conforme exemplificado no estudo de Sousa ao descrever a organização dos *endpoints* da *API* de *e-commerce* com base na lógica *REST* e nos princípios de separação entre responsabilidades (ALMEIDA, 2025).

Os métodos *HTTP* são definidos como: *GET, POST, PUT, DELETE E PATCH.*

1. ***GET***: usado para recuperar um registro.
2. ***POST***: para criar um novo registro.
3. ***PUT***: para atualizar totalmente um registro.
4. ***PATCH***: para atualizar parcialmente um registro.
5. ***DELETE***: para deletar um registro.

O cliente realiza uma solicitação por meio das operações do *REST* (*GET, POST, PUT, PATCH e DELETE*), a *API* recebe a informação do que foi solicitado e então processa no servidor apontando o responsável por prover as informações da resposta, em seguida a *API* recebe a resposta e a envia para o cliente por meio de um formato selecionado, seja *JSON*, *XML*, *HTML* ou outro (PINHEIRO, 2022). A Figura 1 ilustra o modelo arquitetural *REST*.

Figura 1 – Serviço *web* em um modelo arquitetural *REST*



Autor: (PINHEIRO, 2022)

2.3.2 *API SOAP*

O *SOAP* é um protocolo baseado em *XML*, padronizado pela W3C, que garante forte tipagem, suporte a transações complexas e alto nível de segurança. Apesar de mais rígido que *REST*, o *SOAP* é amplamente empregado em domínios que exigem confiabilidade e segurança, como sistemas financeiros e governamentais. Embora mais complexo, o *SOAP* continua sendo uma escolha relevante em contextos nos quais a padronização e a robustez das transações são fatores determinantes. (GONÇALVES, 2025).

Baseia-se em *XML*, que consiste em três partes: um envelope que define uma estrutura para descrever o que está em uma mensagem e como processá-la, um conjunto de regras de codificação para expressar instâncias de tipos de dados definidos pelo aplicativo e uma convenção para representar chamadas e respostas de procedimentos remotos (PONTES, 2022).

Figura 2 – Corpo de um documento *XML* para mensagens *SOAP*

```xml
<?xml version="1.0"?>

<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope/" soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
  ... Envelope define que o documento XML é uma mensagem SOAP
  <soap:Header>
    ... Utilizado para configurações da mensagem
  </soap:Header>

  <soap:Body>
    ... Utilizado para mensagens em si
    <soap:Fault>
      ... Utilizado para mensagens de erro
    </soap:Fault>
  </soap:Body>

</soap:Envelope>
```

Autor: (RIBEIRO, 2021)

2.3.3 *API GraphQL*

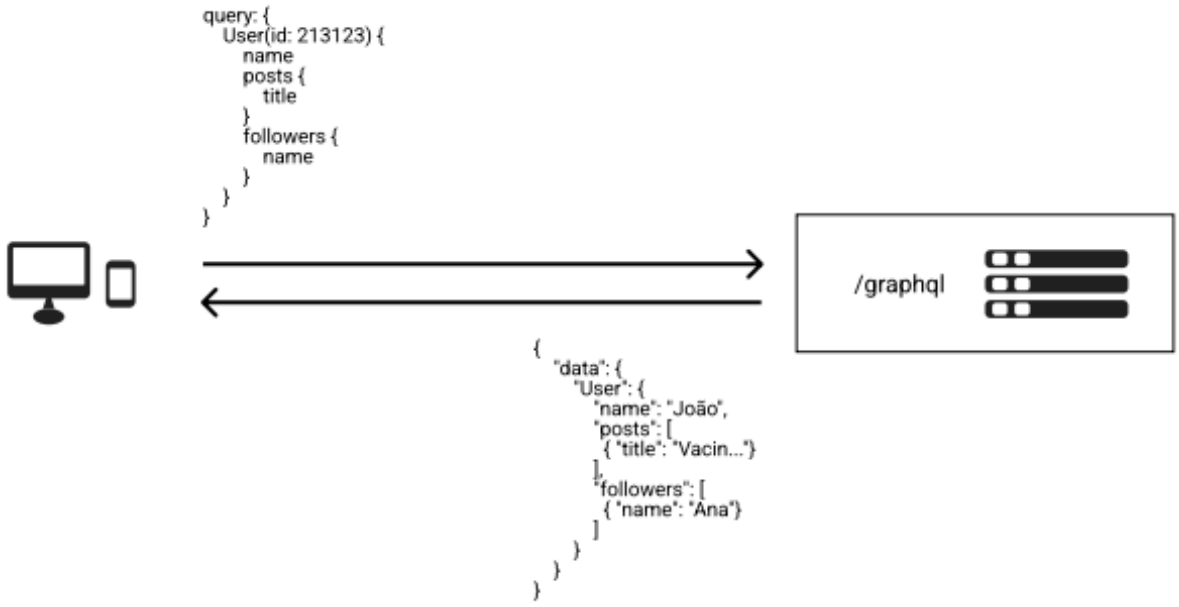
GraphQL é uma linguagem de consulta para *APIs* que permite que os clientes solicitem exatamente os dados de que precisam. Diferente do *REST*, onde o servidor define a estrutura dos dados retornados, no *GraphQL*, o cliente tem controle sobre a consulta, o que pode resultar em menos dados transferidos e menos chamadas de *API* (FILHO, 2025).

É importante ressaltar que a linguagem *GraphQL* tem como prioridade o fornecimento de dados, tendo sido desenvolvida especificamente para busca e tratamento dos mesmos. Sendo assim, ela é ideal para projetos que trabalham com requisições complexas, como serviços *Web* ou móveis, e que precisam lidar eficientemente com as solicitações de dados. Além disso, por ser uma linguagem fortemente tipada, o tratamento e validação de dados é otimizado, seja para o cliente ou para o servidor (ARAÚJO, 2025).

Para realizar uma operação utilizando *GraphQL*, pode-se ter uma consulta (operação de leitura), ou uma mutação (operação de escrita, alteração e exclusão). Para ambos os casos, deve haver uma string em que a camada de execução interpreta e executa no servidor, então respondendo em um formato específico, sendo este o *JSON*, o formato mais utilizado por aplicações *mobile* e *web* (RIBEIRO, 2021).

Na Figura 3, podemos ver que existe apenas um *endpoint* responsável por qualquer tipo de requisição GraphQL. Assim, o que difere o que a aplicação retornará de dados é o que o corpo da própria requisição pedir, no exemplo apresentado pedimos os dados da entidade "*User*" especificando o seu id e logo após definimos que dados queremos receber dessa requisição (OLIVEIRA, 2022).

Figura 3 – Recebimento e envio de mensagens pelo padrão GraphQL



Autor: (OLIVEIRA, 2022)

2.3.4 Segurança em *APIs*

A Segurança da Informação tem o objetivo de minimizar as ocorrências e a gravidade de incidentes como desastres, erros (intencionais ou não) e manipulação não autorizada de informações ou sistemas. A base de um sistema seguro se constitui a partir de princípios essenciais da segurança da informação como confidencialidade, integridade e disponibilidade que devem ser garantidos para que a informação permaneça livre de qualquer intervenção externa (NAKAGAWA, 2017).

A segurança das *APIs REST* depende da aplicação de boas práticas capazes de proteger os dados trafegados e controlar o acesso aos recursos do sistema. Entre essas práticas, destacam-se a autenticação por meio de *tokens*, a utilização de conexões seguras com *HTTPS*, o controle de acesso baseado em papéis, conhecido como *Role-Based Access Control* (RBAC), e a validação das entradas fornecidas pelos usuários, a fim de reduzir riscos de ataques como injeção de comandos e acessos não autorizados. Essas medidas são importantes para garantir que apenas usuários autenticados e autorizados possam acessar determinadas funcionalidades da aplicação, contribuindo para a confidencialidade, integridade e segurança das informações manipuladas pela *API* (ALMEIDA, 2025).

Uma das formas mais usadas atualmente para evitar acessos indevidos é através da autenticação e autorização. São feitas por meio de tecnologias como *JSON Web Tokens* (*JWT*), *OAuth2* e dessa forma protegem as *APIs*, garantindo que apenas usuários autenticados e autorizados tenham acesso ao sistema.

2.4 Testes de *Software*

Os testes de *software* têm como principais objetivos verificar a conformidade do sistema com os requisitos funcionais e não funcionais, identificar defeitos ainda durante o processo de desenvolvimento e validar se o comportamento da aplicação corresponde ao esperado em diferentes cenários de uso. Essa atividade é essencial para a garantia da qualidade do *software*, pois atua como um mecanismo de verificação e validação, complementando outras práticas adotadas no desenvolvimento do sistema (REZENDE et al., 2025).

Além de avaliar o funcionamento das funcionalidades implementadas, os testes contribuem para assegurar que o sistema atenda às necessidades dos usuários e cumpra seu propósito em situações próximas ao uso real. Dessa forma, tornam-se fundamentais para aumentar a confiabilidade da aplicação, reduzir falhas e apoiar a entrega de um produto mais estável e adequado ao contexto em que será utilizado (JUNIOR, 2025).

2.5 Trabalhos Relacionados

Nesta seção, apresentam-se estudos e projetos recentes que guardam similaridade com a proposta deste trabalho, servindo como base comparativa e metodológica.

O trabalho E-commerce Invertido: Valorizando Microempreendedores (MORAIS et al., 2024) serviu como referência para a estruturação da aplicação em camadas, autenticação de usuários e organização do fluxo entre interface e *API*, aspectos que também foram considerados no desenvolvimento deste sistema.

No trabalho de Análise de Desempenho de *APIs* em *E-commerce* dos autores Oliveira e Rodrigues (2025): Este estudo realizou um *benchmarking* entre diferentes *frameworks* para sistemas de vendas. Os autores destacam que, embora *frameworks* em *Go* e *Rust* apresentem alta performance bruta, ecossistemas baseados em *Java* com *Spring Boot* oferecem maior robustez e facilidade de manutenção para regras de negócio complexas de *e-commerce*, justificando a escolha tecnológica deste trabalho.

Sistema RENOVAR (REZENDE et al., 2025): Embora voltado para a gestão de obras, o trabalho de Rezende detalha a implementação de *APIs RESTful* utilizando *Kotlin* e *Spring Boot* com foco em alta cobertura de testes *(JUnit e MockK)*. Este trabalho serve de referência para a estruturação das camadas de serviço e a garantia de qualidade de dados aplicada ao fluxo de pedidos deste projeto.

Diferentemente dos trabalhos apresentados, este projeto tem como foco principal a aplicação prática em um microcomércio, buscando resolver problemas do dia a dia relacionados ao controle de dívidas e registro de vendas. Nesse contexto, destaca-se por propor uma solução acessível e prática, integrando conceitos de desenvolvimento *web* para apoiar a gestão do negócio. Além disso, o sistema foi pensado para ser simples de utilizar, atendendo às necessidades reais de pequenos comerciantes, sem exigir conhecimentos técnicos avançados.

Por fim, o presente trabalho acrescenta uma aplicação prática voltada para microcomércios, reunindo em um único sistema funcionalidades como registro de vendas, controle de débitos, gerenciamento de produtos, estoque e acompanhamento do faturamento.

3 Desenvolvimento

Este capítulo tem como objetivo apresentar as etapas de concepção, planejamento e desenvolvimento do sistema de Gestão de Vendas com Controle de Débitos e Pagamentos em Microcomércios, descrevendo as principais decisões técnicas, arquiteturais e funcionais adotadas durante a construção da aplicação.

3.1 Introdução

O sistema foi desenvolvido utilizando a linguagem *Java* e o *framework Spring Boot* no *backend*, responsáveis pela construção da *API REST* e pela implementação das regras de negócio da aplicação. Para a interface do usuário, foi utilizado o *framework Angular*, que consome os *endpoints* disponibilizados pela *API*, permitindo a interação dos usuários

com as funcionalidades do sistema. A persistência dos dados foi realizada por meio de um banco de dados relacional.

Além disso, são apresentadas as principais decisões arquiteturais adotadas, bem como as tecnologias e ferramentas utilizadas durante o desenvolvimento da aplicação.

3.2 Escolha das Tecnologias e Plataformas para o Projeto

As tecnologias utilizadas no projeto foram definidas com o objetivo de desenvolver uma aplicação robusta, escalável, segura e de fácil manutenção. A escolha considerou a integração entre as ferramentas, sua ampla utilização no mercado e sua adequação ao desenvolvimento de sistemas *web* baseados em *APIs REST*.

Para o desenvolvimento do *backend*, foi utilizada a linguagem *Java* em conjunto com o *framework Spring Boot*, que facilita a criação de aplicações *web*, a exposição de *endpoints REST*, a organização das camadas da aplicação e a implementação das regras de negócio.

A persistência dos dados foi realizada por meio do sistema gerenciador de banco de dados *PostgreSQL*, responsável pelo armazenamento das informações utilizadas pela aplicação, como usuários, produtos, pedidos, pagamentos e registros relacionados ao controle financeiro.

No *frontend*, foi utilizado o *framework Angular*, juntamente com a linguagem *TypeScript*, possibilitando a criação de interfaces organizadas, responsivas e integradas à *API REST*. Essa combinação permite que os usuários interajam com as funcionalidades do sistema de forma mais clara e eficiente.

Dessa forma, a escolha dessas tecnologias contribuiu para a construção de uma solução adequada aos objetivos do projeto, favorecendo a separação de responsabilidades, a manutenção do código e a evolução futura da aplicação.

3.3 Requisitos Funcionais

Os requisitos funcionais descrevem as funcionalidades e comportamentos esperados do sistema, ou seja, definem o que o sistema deve fazer para atender às necessidades dos usuários e dos objetivos do negócio. Esses requisitos são essenciais para garantir que todas as operações principais sejam devidamente implementadas e executadas de forma correta.

A Tabela 1 apresenta os requisitos funcionais definidos para o correto funcionamento do sistema, descrevendo as principais funcionalidades que devem ser implementadas. Entre elas, estão o cadastro e autenticação de usuários, o registro e acompanhamento de compras, o controle de dívidas dos clientes, além das funcionalidades do administrador, como gestão de produtos, controle de estoque, visualização de relatórios e faturamento mensal.

Tabela 1 – Requisitos Funcionais

Código	**Descrição**
RF01	O sistema deve permitir o cadastro de clientes, sujeito à aprovação do administrador.
RF02	O sistema deve permitir a autenticação de usuários.
RF03	O sistema deve permitir a alteração de senha pelo cliente autenticado.
RF04	O sistema deve permitir a visualização dos produtos disponíveis para o cliente.
RF05	O sistema deve permitir que o cliente selecione um produto, informe a quantidade desejada e registre a compra, adicionando o valor ao seu débito.
RF06	O sistema deve permitir a visualização do débito atual pelo cliente.
RF07	O sistema deve permitir a visualização do histórico de compras pelo cliente.
RF08	O sistema deve permitir o encerramento da sessão do usuário.
RF09	O sistema deve permitir a visualização dos clientes pelo administrador.
RF10	O sistema deve permitir a visualização do faturamento mensal atual e anterior pelo administrador.
RF11	O sistema deve permitir a exclusão de usuários pelo administrador.
RF12	O sistema deve permitir o cadastro de produtos, incluindo nome e preço, pelo administrador.
RF13	O sistema deve permitir a atualização do estoque de produtos pelo administrador.
RF14	O sistema deve permitir a visualização do histórico de compras de cada cliente pelo administrador.
RF15	O sistema deve permitir a visualização de relatório geral contendo informações de todos os clientes pelo administrador.
RF16	O sistema deve permitir o registro de pagamentos de clientes pelo administrador.
RF17	O sistema deve permitir a edição e exclusão de produtos pelo administrador.
RF18	O sistema deve impedir a realização de compras quando a quantidade solicitada for maior que o estoque disponível.
RF19	O sistema deve permitir a visualização dos produtos cadastrados pelo administrador.
RF20	O sistema deve permitir que o administrador aprove o cadastro de novos clientes.

Autoria Própria

3.4 Requisitos Não Funcionais

Os requisitos não funcionais estão relacionados à qualidade do sistema, como desempenho, segurança, usabilidade e confiabilidade. Eles definem como o sistema deve se comportar durante sua execução.

Nesse sistema, foram considerados aspectos como segurança (uso de *JWT* e *hash* de senhas), desempenho (tempo de resposta), usabilidade (interface responsiva) e confiabilidade, além da adoção de boas práticas de desenvolvimento, padrão *REST* e documentação da *API*.

A Tabela 2 apresenta os requisitos não funcionais definidos para o sistema.

Tabela 2 – Requisitos Não Funcionais

Código	**Descrição**
RNF01	O sistema deve possuir disponibilidade mínima de 99% do tempo.
RNF02	O sistema deve realizar a autenticação de usuários por meio de *token JWT*.
RNF03	O *token JWT* deve possuir tempo de expiração configurado, garantindo a segurança da sessão.
RNF04	O sistema deve armazenar as senhas utilizando algoritmo de *hash* seguro.
RNF05	O sistema deve seguir uma arquitetura em camadas, promovendo a separação de responsabilidades entre os componentes da aplicação.
RNF06	A *API* deve ser devidamente documentada.
RNF07	A *API* deve seguir o padrão arquitetural *REST*.
RNF08	A comunicação entre cliente e servidor deve ocorrer por meio de requisições *HTTP*.
RNF09	O sistema deve responder às requisições da *API* em tempo inferior a 2 segundos, em condições normais de uso.

Autoria Própria

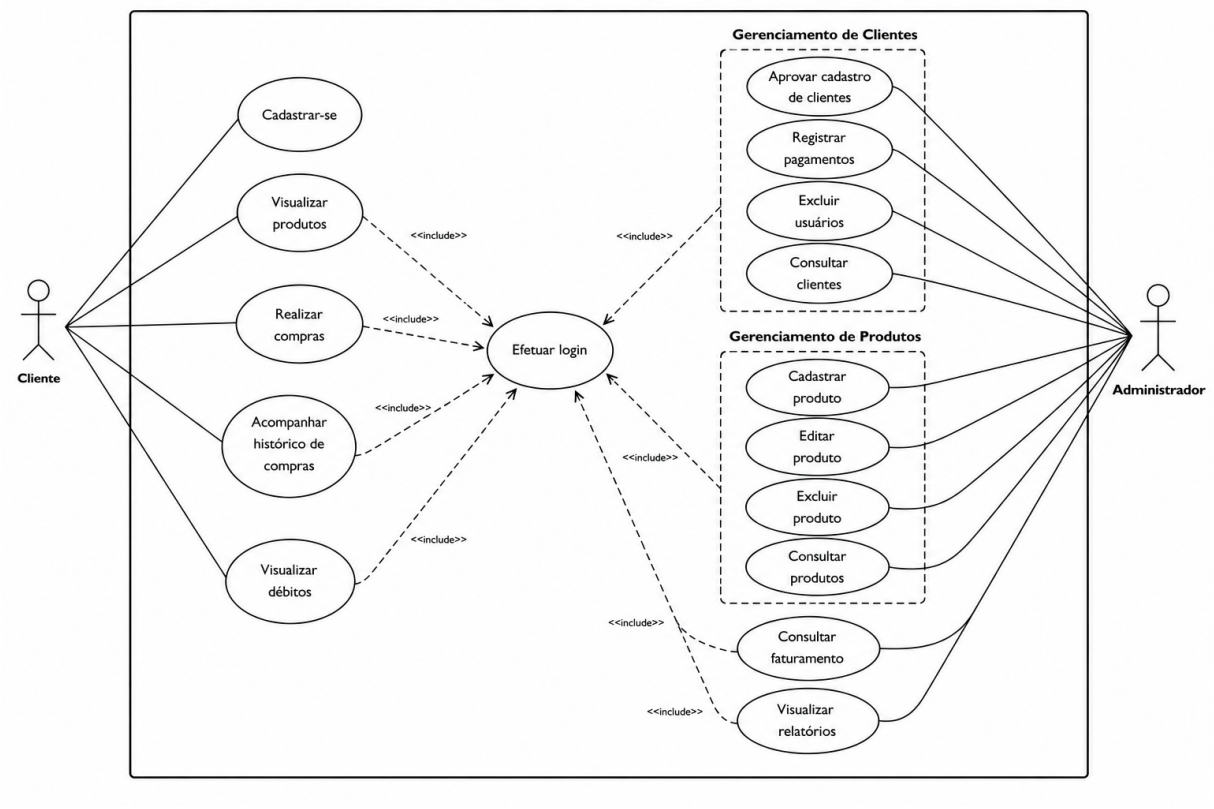
3.5 Caso de Uso

Com o objetivo de representar de forma organizada as funcionalidades do sistema e a interação dos usuários com a aplicação, foi elaborado o Diagrama de Caso de Uso. Esse tipo de diagrama permite visualizar os atores envolvidos e as principais ações que podem ser realizadas no sistema, contribuindo para a compreensão dos requisitos funcionais da aplicação.

No sistema proposto, foram definidos dois atores principais: o cliente e o administrador. O cliente interage com a aplicação para realizar seu cadastro, efetuar *login*, visualizar os produtos disponíveis, realizar compras, acompanhar o histórico de compras e consultar seus débitos. Essas funcionalidades permitem que o usuário acompanhe suas operações e tenha maior controle sobre suas pendências financeiras.

O administrador, por sua vez, possui acesso às funcionalidades de gerenciamento do sistema. Entre suas responsabilidades, estão a aprovação de cadastros de clientes, o registro de pagamentos, a consulta e exclusão de usuários, além do gerenciamento de produtos, incluindo cadastro, edição, exclusão e consulta. O administrador também pode consultar o faturamento e visualizar relatórios, auxiliando no acompanhamento financeiro do microcomércio. A Figura 4 ilustra o caso de uso da aplicação.

Figura 4 – Diagrama de Caso de Uso



Autoria própria

Dessa forma, o Diagrama de Caso de Uso apresenta uma visão geral das funcionalidades disponíveis para cada perfil de usuário, evidenciando a separação entre as ações do cliente e as ações administrativas.

3.6 Modelo da arquitetura - C4

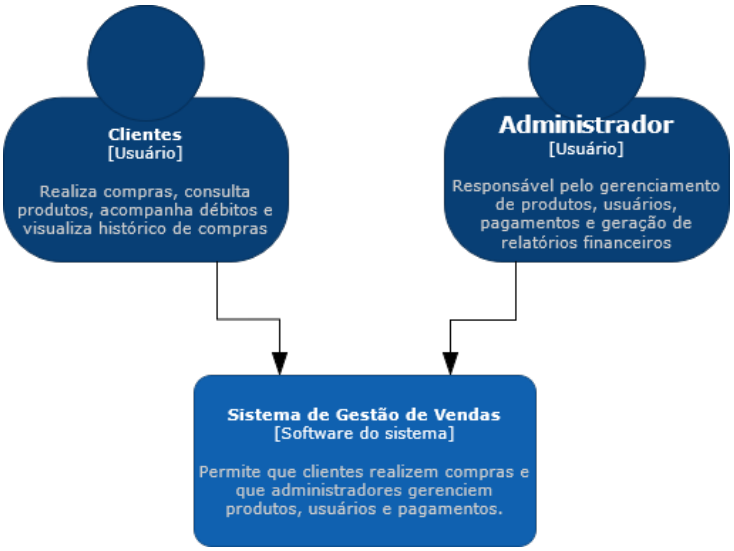
O modelo C4 é uma forma de representar a arquitetura de um sistema de um jeito mais organizado e fácil de entender. Ele divide o sistema em níveis, começando por uma visão mais geral e indo até partes mais específicas. Assim, fica mais simples visualizar como o sistema funciona e como tudo está conectado. Esses níveis são conhecidos como C1 (Contexto), C2 (*Contêiner*), C3 (Componentes) e C4 (Código).

3.6.1 Modelo C1

O diagrama de contexto (C4 Nível 1) apresenta uma visão geral do sistema de vendas, destacando os principais usuários e suas interações com o sistema. O sistema é utilizado por dois usuários principais, clientes e administradores. Os clientes são responsáveis por realizar compras, consultar produtos, acompanhar seus débitos e visualizar o histórico de compras. Os administradores possuem acesso ao gerenciamento, incluindo o controle dos produtos, usuários, pagamentos e a capacidade de geração de relatórios financeiros. O

diagrama põe em evidência a relação direta entre os usuários e o sistema demonstrando como ambos interagem exercendo suas funcionalidades, sem detalhar aspectos técnicos ou estruturais. A Figura 5 tem a estrutura desse nível.

Figura 5 – Modelo C1



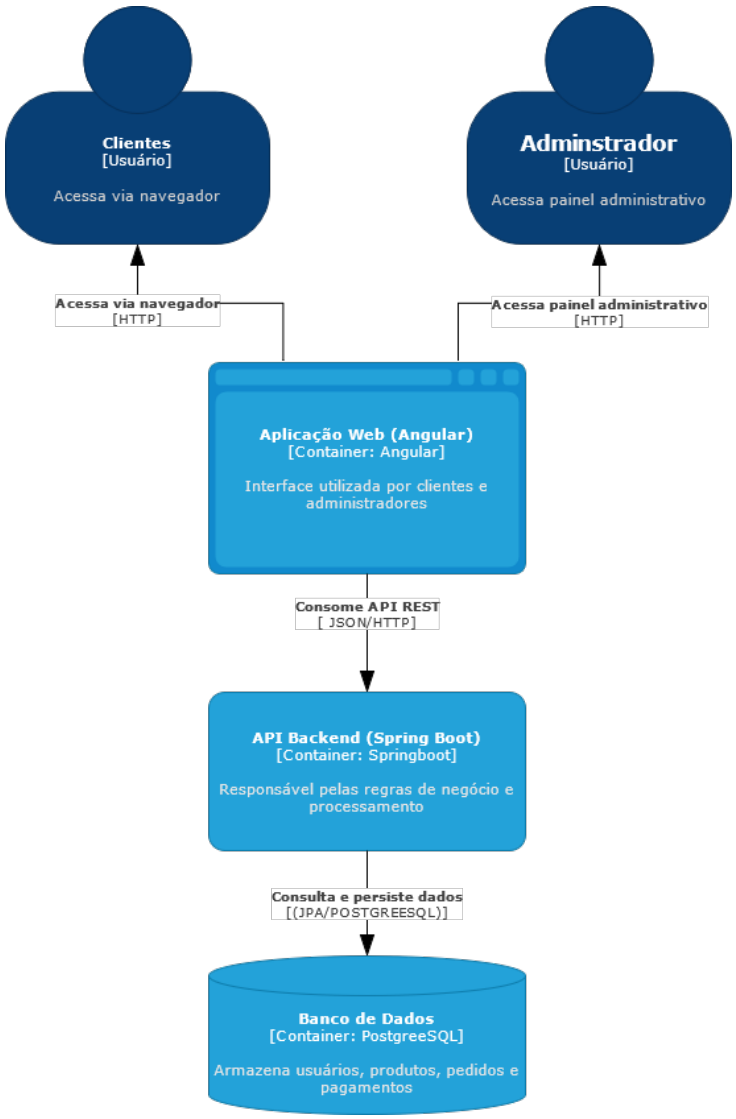
Autoria própria

3.6.2 Modelo C2

O diagrama de *containers* (C4 nível 2) representado através da Figura 6 detalha a estrutura interna do sistema, evidenciando os principais componentes tecnológicos responsáveis pelo seu funcionamento. A aplicação é composta por três *containers* principais: a aplicação *web*, o *backend* e o banco de dados. A aplicação *web*, desenvolvida com *Angular*, é responsável pela interface com o usuário, permitindo a interação dos clientes e administradores por meio de um navegador.

O *backend* desenvolvido com *Spring Boot* concentra as regras de negócio e o processamento das requisições, sendo responsável por receber as chamadas provenientes da aplicação *web* por meio de uma *API REST*, utilizando o protocolo *HTTP* e o formato de dados *JSON*. Por fim, o sistema utiliza um banco de dados *PostgreSQL* responsável pelo armazenamento das informações da aplicação, como usuários, pedidos, pagamentos e produtos, A comunicação entre o *backend* e o banco de dados é realizada por meio do *JPA*, que abstrai as operações de persistência e manipulação de dados.

Figura 6 – Modelo C2

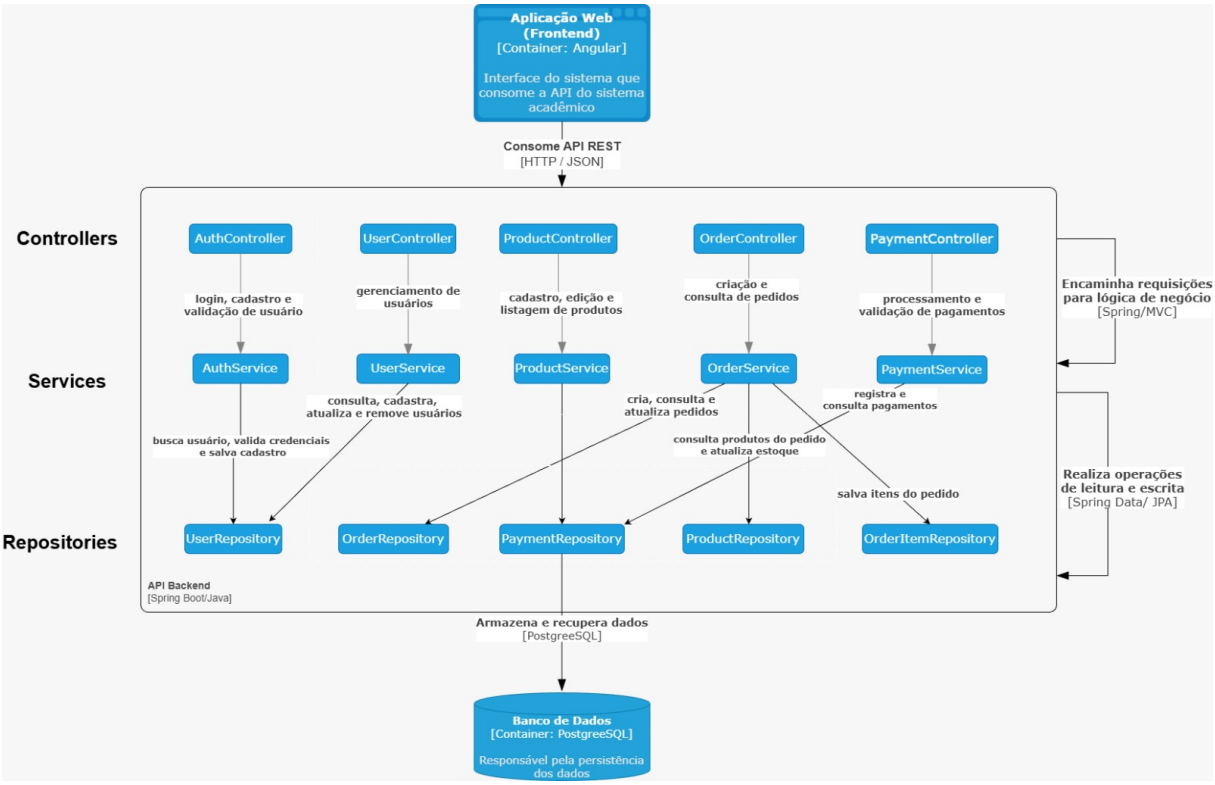


Autoria própria

3.6.3 Modelo C3

O diagrama de componentes(C4 nível 3) apresenta uma visão detalhada da organização interna do *backend* do sistema, destacando a divisão em camadas e a responsabilidade de seus respectivos componentes. A Arquitetura do *backend* segue o padrão em camadas, composta por *services*, *controllers* e *repositories*. Os *controllers* são responsáveis por receber as requisições *HTTP* e encaminhá-las para as camadas internas do sistema. A camada de *services* concentra as regras de negócios, processando dados, realizando validações e coordenando operações necessárias para atender as requisições. Já os *repositories* são responsáveis pela persistência de dados, realizando leitura e escrita no banco de dados por meio de utilização *JPA (Java Persistence API)*. Essa organização em camadas promove a separação de responsabilidades, facilitando a evolução do sistema, escalabilidade e manutenção. A Figura 7 representa a figura desse nível.

Figura 7 – Modelo C3



Autoria própria

3.7 Diagrama de Classes UML

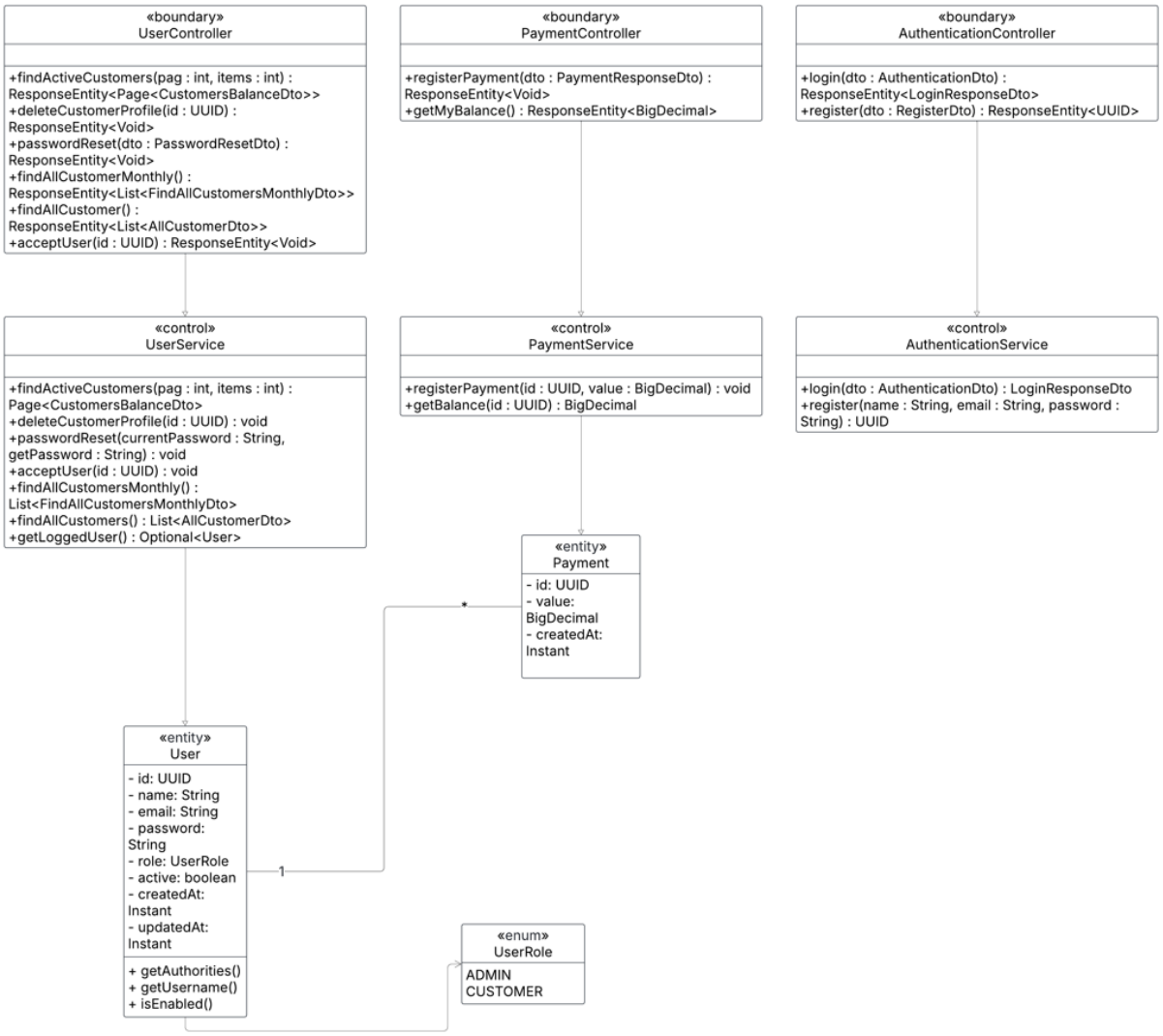
As Figuras 8 e 9 apresentam o Diagrama de Classes UML do sistema de Gestão de Vendas com Controle de Débitos e Pagamentos em Microcomércios. Para facilitar a visualização e a organização das informações, o diagrama foi dividido em duas partes, contemplando as principais classes da aplicação, seus atributos, métodos e relacionamentos.

A Figura 8 apresenta as classes relacionadas ao gerenciamento de usuários, autenticação e pagamentos. Nessa parte do sistema, destacam-se as classes *User*, *Payment*, *UserController*, *UserService*, *PaymentController*, *PaymentService*, *AuthenticationController* e *AuthenticationService*. A classe *User* representa os usuários da aplicação, armazenando informações como identificador, nome, e-mail, senha, papel de acesso, status de atividade e datas de criação e atualização. O papel do usuário é definido pelo *enum* *UserRole*, que diferencia os perfis *ADMIN* e *CUSTOMER*.

Observa-se também que o sistema possui classes específicas para autenticação e controle de acesso. O *AuthenticationController* atua como ponto de entrada para as requisições de *login* e cadastro de usuários, enquanto o *AuthenticationService* concentra a lógica responsável por validar credenciais, registrar novos usuários e apoiar o processo de autenticação. Já as classes *UserController* e *UserService* são responsáveis pelas funcionalidades relacionadas aos usuários, como consulta de clientes, alteração de senha, aprovação de cadastro e exclusão lógica. A classe *Payment*, por sua vez, representa os pagamen-

tos realizados pelos clientes, armazenando o valor pago e a data de criação do registro. Cada pagamento está associado a um usuário, permitindo o controle dos valores pagos e o cálculo do saldo devedor.

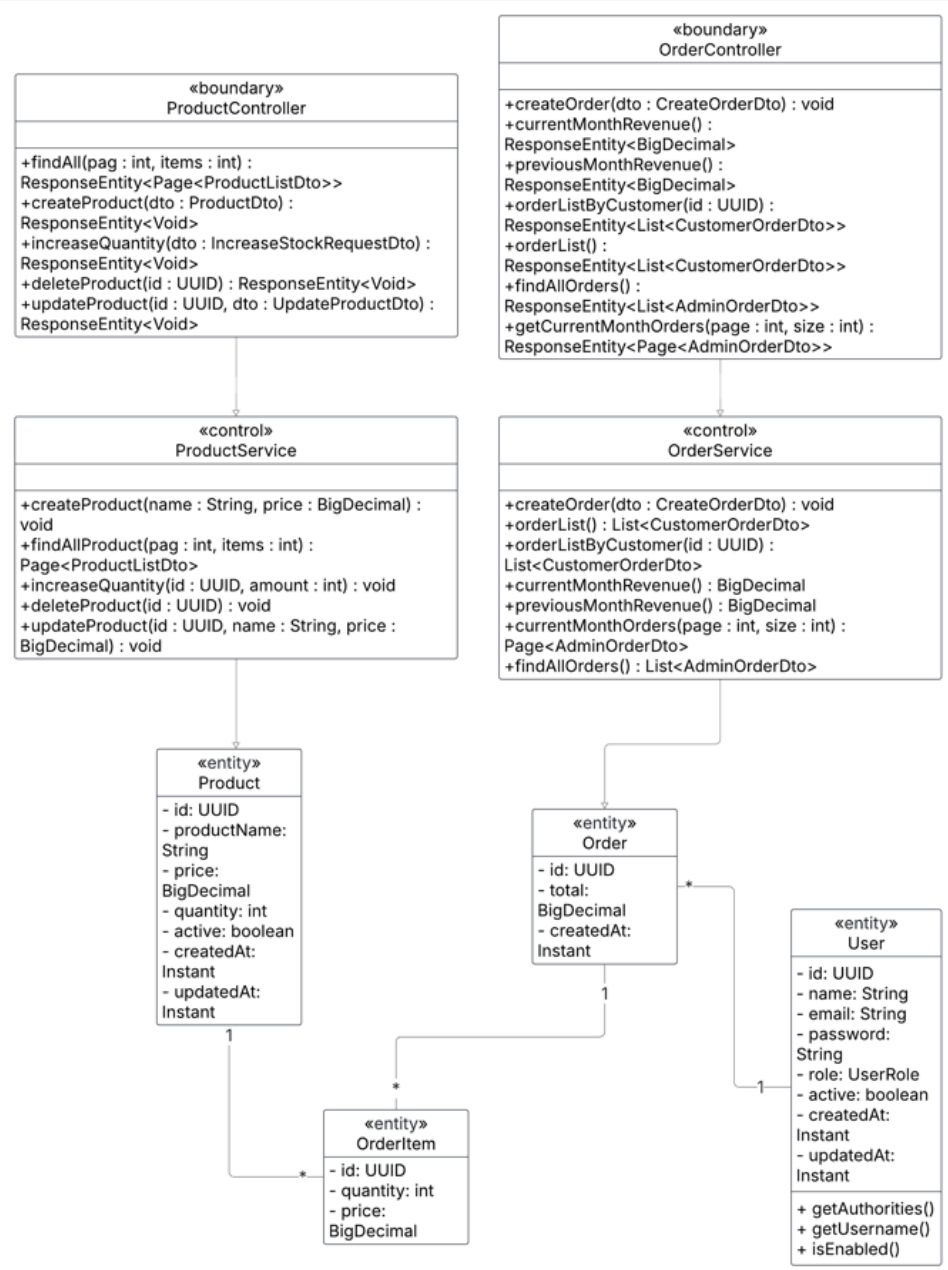
Figura 8 – Diagrama de Classes UML – Usuários, autenticação e pagamentos



Autoria própria

A Figura 9 apresenta as classes relacionadas ao gerenciamento de produtos, pedidos e itens de pedido. Nessa parte do diagrama, destacam-se as classes *Product*, *Order*, *OrderItem*, *ProductController*, *ProductService*, *OrderController* e *OrderService*. A classe *Product* representa os produtos cadastrados no sistema, contendo informações como nome, preço, quantidade em estoque, status de atividade e datas de criação e atualização. As classes *ProductController* e *ProductService* são responsáveis pelas operações de cadastro, consulta, atualização, controle de estoque e exclusão lógica dos produtos.

Figura 9 – Diagrama de Classes UML – Produtos, pedidos e itens de pedido



Autoria própria

No fluxo de compras, a classe *Order* representa o pedido realizado por um cliente, armazenando informações como identificador, valor total e data de criação. Cada pedido está associado a um usuário e pode conter um ou mais itens, representados pela classe *OrderItem*. Essa classe registra os produtos incluídos na compra, a quantidade adquirida e o preço correspondente. Dessa forma, o relacionamento entre *Order* e *OrderItem* permite estruturar os pedidos de maneira detalhada, enquanto a associação com *Product* indica quais produtos foram comprados.

As classes controladoras, identificadas pelo estereótipo *boundary*, são responsáveis por receber as requisições realizadas pelos usuários por meio da API. As classes de serviço,

identificadas pelo estereótipo *control*, concentram as regras de negócio da aplicação, como validações, cálculos, consultas e operações de atualização. Já as classes marcadas como *entity* representam os dados persistidos no banco de dados, compondo a estrutura principal do domínio do sistema.

Dessa forma, o Diagrama de Classes UML evidencia a separação de responsabilidades entre as camadas da aplicação, facilitando a compreensão da estrutura do sistema e contribuindo para sua manutenção e evolução futura.

3.8 Desenvolvimento

Esta seção apresenta o processo de desenvolvimento do *software*, descrevendo as principais etapas realizadas desde a definição inicial da solução até a implementação das funcionalidades do sistema. São abordadas as decisões técnicas que orientaram a construção da aplicação, incluindo a organização da arquitetura, a estruturação das camadas, a definição dos recursos da *API REST*, a integração com o banco de dados e a implementação dos mecanismos de autenticação e segurança.

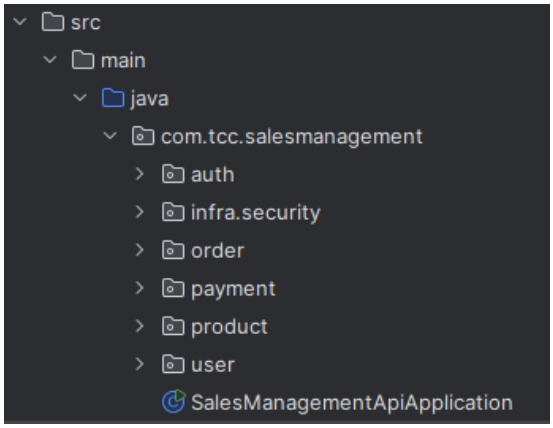
O desenvolvimento foi conduzido de forma incremental, permitindo que as funcionalidades fossem implementadas, testadas e ajustadas gradualmente. Inicialmente, foram definidos os requisitos do sistema e as principais entidades envolvidas no domínio da aplicação, como usuários, produtos, pedidos, itens de pedido e pagamentos. Em seguida, foram estruturadas as camadas responsáveis pelo recebimento das requisições, aplicação das regras de negócio e persistência dos dados.

Dessa forma, esta seção detalha como as tecnologias escolhidas foram aplicadas na prática para atender aos objetivos do projeto, demonstrando a organização interna da aplicação e o funcionamento dos principais recursos implementados.

3.9 Arquitetura em Camadas

O sistema foi desenvolvido utilizando uma arquitetura em camadas, separando responsabilidades entre *Controller, Entidade, Service, Repository e DTOs.* Além disso, foi adotado o padrão *Package by Feature* para organização dos pacotes, agrupando as classes de acordo com suas funcionalidades, o que contribui para uma melhor organização e manutenção do código. Dessa forma, o código se torna mais fácil de ser entendido e mantê-lo, já que tudo relacionado a uma funcionalidade está dentro de um só pacote. As funcionalidades foram separadas em: *Auth, Infra, Order, Payment, Product* e *User*, conforme é possível visualizar na Figura 10.

Figura 10 – Arquitetura do Projeto



Autoria própria

Auth: estão as classes responsáveis pela autenticação da aplicação. O *AuthenticationController* expõe os *endpoints* de *login* e registro, enquanto o *Service* contém a lógica de validação das credenciais e criação de usuários.

Infra: ficam as configurações de segurança e infraestrutura. Ele inclui o *SecurityConfiguration*, que define as regras de acesso, o *SecurityFilter*, que intercepta as requisições, o *TokenService*, responsável pela geração e validação de *tokens (JWT)*, e o *AuthorizationService*, que auxilia na identificação do usuário autenticado.

User: estão as classes responsáveis pelo gerenciamento dos usuários. A classe *User* representa a entidade, o *UserController* expõe os *endpoints* da *API*, o *UserService* contém as regras de negócio e o *UserRepository* realiza o acesso ao banco de dados. Além disso, o *UserRole* define os perfis de acesso dos usuários.

Order: estão as classes responsáveis pelo gerenciamento de pedidos. A classe *Order* representa o pedido, enquanto *OrderItem* representa os itens associados a ele. O *OrderController* expõe os *endpoints* da *API*, o *OrderService* contém as regras de negócio, e os repositórios *OrderRepository* e *OrderItemRepository* realizam o acesso ao banco de dados.

Product: estão as classes responsáveis pelo gerenciamento de produtos disponíveis na aplicação. A classe *Product* representa os dados do produto, como nome, quantidade e preço. O *ProductController* expõe os *endpoints* para operações como cadastro e consulta, o *ProductService* contém as regras de negócio relacionadas aos produtos, e o *ProductRepository* realiza a persistência no banco de dados.

Payment: estão as classes responsáveis pelo gerenciamento de pagamentos. O *PaymentController* expõe os *endpoints* para registro de pagamentos e saldo do cliente, o *PaymentService* contém a lógica de negócio, incluindo o cálculo da dívida do cliente com base nos pagamentos realizados, e o *PaymentRepository* realiza o acesso ao banco de dados.

3.10 *API REST*

A aplicação foi desenvolvida seguindo o estilo arquitetural *REST*, para construção da *API*. A escolha pelo *REST* se deu pela sua simplicidade, ampla adoção no mercado e compatibilidade nativa com o *Angular*.

Nesse modelo, os recursos do sistema são disponibilizados através de *endpoints* acessíveis por meio do protocolo *HTTP*, utilizando os métodos como *GET, POST, PUT, PATCH* e *DELETE* para realizar operações sobre os dados. As respostas da *API* são retornadas no formato *JSON*, facilitando a comunicação entre o *backend* e o *frontend*.

Figura 11 – Classe com anotação *@RestController*

Autoria própria

Na implementação, a anotação @*RestController* do *Spring Boot* é utilizada para expor os *endpoints* e garantir automaticamente a serialização dos objetos *Java* para o formato *JSON* nas respostas *HTTP*. Além disso, a anotação @*RequestMapping* é utilizada para definir o caminho base das requisições da *API*, organizando os *endpoints* de acordo com cada recurso. A Figura 11 representam *UserController* que é referente a entidade *User*.

3.11 Autenticação e Segurança

Com o objetivo de garantir a segurança do sistema, foi utilizado o *framework Spring Security*, com autenticação baseada em *JSON Web Token (JWT)*. Após o *login*, o sistema gera um *token* contendo informações do usuário, como *e-mail*, papel de acesso (*role*) e nome, que é enviado ao cliente.

Esse *token* deve ser incluído nas requisições para acesso aos *endpoints* protegidos da *API*, sendo validado a cada requisição por meio de filtros de segurança. O controle de acesso é baseado em *roles*, permitindo restringir funcionalidades de acordo com o perfil do usuário. A Figura 12 apresenta o método de geração do *token JWT*.

Figura 12 – Método que gera o *token JWT*

```
public String generateToken(User user){  1 usage  Antonio Vitor
    try{
        Algorithm algorithm = Algorithm.HMAC256(secret);
        String token = JWT.create()
                .withIssuer("salesmanagement-api")
                .withSubject(user.getEmail())
                .withClaim( name: "role", user.getRole().name())
                .withClaim( name: "nome", user.getName())
                .withExpiresAt(genExperiationDate())
                .sign(algorithm);
        return token;
    }catch(JWTCreationException exception){
        throw new RuntimeException("Error while generating token", exception);
    }
}
```

Autoria própria

No *token*, o *e-mail* é definido como *subject*, identificando o usuário autenticado, enquanto o nome e o papel de acesso são adicionados como *claims*, possibilitando sua recuperação durante o processamento das requisições. Além disso, o *token* possui um emissor definido ("*salesmanagement-api*") e um tempo de expiração, aumentando a segurança ao evitar reutilização indevida.

3.12 Banco de Dados

A persistência dos dados é realizada por meio do *framework Spring Data JPA*, que acaba facilitando a comunicação entre a aplicação e o banco de dados relacional *PostgreSQL*. As entidades da aplicação representam as tabelas do banco de dados, sendo mapeadas por meio de anotações do *JPA*, como *@Entity*, *@Table*, *@Id* e *@GeneratedValue*.

Figura 13 – Classe de entidade representando a tabela de usuários

```
@NoArgsConstructor  Antonio Vitor
@Getter
@Setter
@Entity
@Table(name = "tb_users")
public class User implements UserDetails {

    @Id
    @GeneratedValue
    private UUID id;
```

Autoria própria

Conforme ilustra a Figura 13, a anotação @*Entity* indica que a classe será mapeada para uma tabela no banco de dados, enquanto @*Id* e @*GeneratedValue* definem a chave primária e sua geração automática. Na aplicação, os identificadores são do tipo *UUID*, garantindo maior unicidade e segurança na identificação dos registros. A anotação @*Table* é utilizada para definir o nome da tabela no banco de dados à qual a entidade será mapeada.

O acesso aos dados é realizado através de interfaces de repositório que estendem *JpaRepository*, permitindo a execução de operações de *CRUD* simplificada, sem a necessidade de implementação manual de consultas *SQL*. A configuração da conexão com o banco de dados é realizada no arquivo *application properties*.

3.13 *Soft Delete*

Para evitar a remoção definitiva de registros no banco de dados, foi utilizada a estratégia de *Soft Delete*. Dessa forma, ao invés de excluir fisicamente um registro, seja um usuário ou um produto, o sistema apenas altera um atributo booleano (*active*) que indica se o registro está ativo ou inativo.

Com isso, os dados permanecem armazenados no banco de dados, possibilitando rastreabilidade, auditoria e eventual recuperação das informações. Nas operações de consulta da aplicação, são considerados apenas os registros marcados como ativos, garantindo que os dados logicamente excluídos não sejam exibidos ao usuário final.

Figura 14 – Método de inativação do usuário

```
public void deleteCustomerProfile(UUID id) {  1 usage   Antonio Vitor *
    User user = userRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("Usuário não encontrado"));

    user.setActive(false);
    userRepository.save(user);
}
```

Autoria própria

Durante o processo de exclusão, o cliente (administrador do sistema) informa o identificador do usuário para a *API*. Em seguida, o sistema recupera o registro do usuário e atualiza o atributo *active* para *false*, caracterizando a exclusão lógica, conforme ilustrado no código apresentado na Figura 14.

3.14 *Endpoints* da *API*

Nesta seção são apresentados alguns dos principais *endpoints* da *API*, responsáveis pelas funcionalidades centrais do sistema. Os *endpoints* representam os pontos de acesso da API, permitindo a comunicação entre cliente e servidor por meio de requisições *HTTP*.

O cadastro de usuários é feito através do *endpoint* ***POST /auth/register*** que chama o método *register*, onde o usuário informa nome, *e-mail* e senha. No início do método, é realizada uma verificação para identificar se já existe algum usuário cadastrado com o mesmo *e-mail* informado. Caso o *e-mail* já esteja em uso, o sistema lança uma exceção informando que aquele endereço de *e-mail* já foi cadastrado, impedindo a criação de uma nova conta.

Se o *e-mail* não existir na base de dados, o processo continua normalmente. A senha informada é criptografada utilizando o *PasswordEncoder*, garantindo mais segurança no armazenamento das credenciais. Depois disso, é criado um novo usuário com os dados recebidos e com o perfil padrão definido como *CUSTOMER*.

Por fim, o usuário é salvo no banco de dados através do userRepository, e o método retorna o identificador (*UUID*) do usuário cadastrado, conforme apresentado na Figura 15.

Figura 15 – Método de Cadastro

```
public UUID register(String name, String email, String password) {
    if(this.userRepository.findByEmail(email).isPresent()){
        throw new RuntimeException("Email already exists");
    }

    String pass = passwordEncoder.encode(password);
    User user = new User(name, email, pass, UserRole.CUSTOMER);
    userRepository.save(user);
    return user.getId();
}
```

Autoria própria

A autenticação é realizada por meio do *endpoint* ***POST /auth/login***, no qual o usuário informa suas credenciais (*e-mail* e senha). Esse *endpoint* aciona o método *login*, responsável por processar a autenticação. Inicialmente, as credenciais são encapsuladas em um objeto de autenticação e validadas pelo *AuthenticationManager* do *Spring Security*. Caso a autenticação seja bem-sucedida, o usuário autenticado é recuperado pelo sistema. Em seguida, é realizada uma validação adicional para verificar se o usuário foi aprovado por um administrador.

Após essas etapas, é gerado um *token JWT* por meio do *TokenService*, que é retornado ao cliente e utilizado para autenticar as requisições subsequentes da *API*. A Figura 16 demonstra esse *endpoint*.

Figura 16 – Método de *Login*

```
public LoginResponseDto login(AuthenticationDto dto){  1 usage   Antonio Vitor
    var userNamePassword = new UsernamePasswordAuthenticationToken(dto.email(), dto.password());
    var auth = this.authenticationManager.authenticate(userNamePassword);
    var user = (User) auth.getPrincipal();

    if(!user.isAccepted()){
        throw new RuntimeException("Usuário não aceito pelo admin.");
    }

    var token = tokenService.generateToken((User) auth.getPrincipal());
    return new LoginResponseDto(token);

}
```

Autoria própria

O *endpoint* ***POST /order/create-order*** é responsável pela criação de pedidos no sistema e aciona o método *createOrder*, que implementa essa funcionalidade. Ao ser chamado, o sistema recupera o usuário autenticado e inicia a criação de um novo pedido.

Durante o processamento, os itens informados são validados, associados ao pedido e utilizados para o cálculo do valor total. Também é realizada a verificação de disponibilidade em estoque, com a devida atualização da quantidade dos produtos. O método é anotado com @*Transactional*, garantindo que todas as operações envolvidas na criação do pedido sejam executadas de forma atômica, ou seja, em caso de erro, nenhuma alteração é persistida no banco de dados.

Por fim, o pedido é validado e persistido no banco de dados. A Figura 17 apresenta o método responsável por esse processo.

Figura 17 – Método para criar pedido

```
@Transactional  1 usage   Antonio Vitor
public void createOrder(CreateOrderDto dto){
    User user = userService.getLoggedUser().orElseThrow();
    Order order = new Order(user);
    BigDecimal total = processOrderItems(dto, order);
    validateOrder(order);
    order.setTotal(total);

    orderRepository.save(order);
}
```

Autoria própria

O *endpoint* ***POST /payment/register*** é responsável pelo registro de pagamentos no sistema. Ele chama o método *registerPayment*, esse método recebe o identificador do usuário e o valor do pagamento. Em seguida, o sistema recupera o usuário no banco de dados e cria um novo registro de pagamento associado a ele, realizando sua persistência.

Além disso, o sistema possui o *endpoint* ***GET /payment/balance/id***, que aciona o método *getBalance*, responsável pelo cálculo do saldo devedor do cliente. Esse cálculo é realizado a partir da soma do valor total dos pedidos, subtraindo-se o total de pagamentos já efetuados. Caso não existam valores registrados, são considerados valores zero, evitando inconsistências. Na Figura 18 são apresentados os métodos *registerPayment*, responsável pelo *endpoint /payment/register, e getBalance*, responsável pelo *endpoint /payment/balance/id*.

Figura 18 – Métodos de pagamento e saldo do cliente

```
public void registerPayment(UUID id, BigDecimal value){  1 usage  Antonio Vitor
    var user = userRepository.findById(id).orElseThrow();
    Payment payment = new Payment(user, value);
    paymentRepository.save(payment);
}

public BigDecimal getBalance(UUID id){  1 usage  Antonio Vitor
    var sumOrder = orderRepository.sumTotalByCustomerId(id);
    var sumPayment = paymentRepository.sumValueByUserId(id);

    sumOrder = sumOrder != null ? sumOrder : BigDecimal.ZERO;
    sumPayment = sumPayment != null ? sumPayment : BigDecimal.ZERO;

    return sumOrder.subtract(sumPayment);
}
```

Autoria própria

O *endpoint* ***GET /order/current-revenue*** é responsável por retornar o faturamento do mês atual. Ao ser acionado, ele chama o método *currentMonthRevenue*, que realiza o cálculo do total de vendas com base nos pedidos registrados no mês e ano corrente.

Para isso, o método utiliza a consulta *sumByMonthAndYear*, presente no *OrderRepository*, responsável por somar os valores dos pedidos (*SUM*) considerando apenas os registros do período atual. Caso não existam vendas no período, é retornado o valor zero.

Esse *endpoint* é utilizado na interface administrativa para exibição de indicadores, como o faturamento mensal do sistema, conforme a Figura 19.

Figura 19 – Método de faturamento mensal

```
public BigDecimal currentMonthRevenue(){  1 usage  Antonio Vitor
    ZoneId zone = ZoneId.of( zoneId: "America/Sao_Paulo");
    YearMonth current = YearMonth.now(zone);
    return orderRepository.sumByMonthAndYear(current.getMonthValue(),
            current.getYear());
}
```

Autoria própria

O *endpoint* ***GET /order/current-month-orders*** permite a listagem dos pedidos realizados no mês atual. Ao ser acionado, ele chama o método *currentMonthOrders*, responsável por buscar os pedidos com base no mês e ano corrente, como mostra a Figura 20.

Figura 20 – Método de pedidos do mês atual

```
public Page<AdminOrderDto> currentMonthOrders(int page, int size){
    ZoneId zone = ZoneId.of( zoneId: "America/Sao_Paulo");
    YearMonth current = YearMonth.now(zone);

    Pageable pageable = PageRequest.of(page, size);

    return orderItemRepository.findOrdersByMonth(
            current.getMonthValue(),
            current.getYear(), pageable
    );
}
```

Autoria própria

O *endpoint* ***POST /product*** é responsável pelo cadastro de produtos no sistema. Ao ser acionado, ele chama o método *createProduct*, que recebe o nome e o preço do produto. Em seguida, o produto é criado e salvo no banco de dados. É possível visualizar o método na Figura 21.

Figura 21 – Método para cadastrar produto

```
public void createProduct(String name, BigDecimal price){
    Product product = new Product(name, price);
    productRepository.save(product);
}
```

Autoria própria

O *endpoint* ***PATCH /product/id*** é responsável pela atualização parcial de produtos e aciona o método *updateProduct*. Ao ser chamado, o sistema recupera o produto pelo identificador informado e realiza a atualização apenas dos campos enviados na requisição.

Nesse processo, é possível alterar o nome e/ou o preço do produto. Caso algum dos campos não seja informado, o valor atual é mantido. Após as alterações, o produto é persistido novamente no banco de dados, conforme mostra a Figura 22.

Figura 22 – Método para atualizar produto

```
public void updateProduct(UUID id, String name, BigDecimal price){
    var product = productRepository.findById(id).orElseThrow();
    if(name != null){
        product.setProductName(name);
    }
    if(price != null){
        product.setPrice(price);
    }

    productRepository.save(product);

}
```

Autoria própria

3.15 Interface do Sistema

A interface do sistema foi desenvolvida utilizando o *framework Angular*, com o objetivo de proporcionar uma experiência de uso simples, organizada e intuitiva. Por meio da interface, os usuários conseguem acessar as funcionalidades disponibilizadas pela aplicação, interagindo com os recursos do sistema de acordo com seu perfil de acesso.

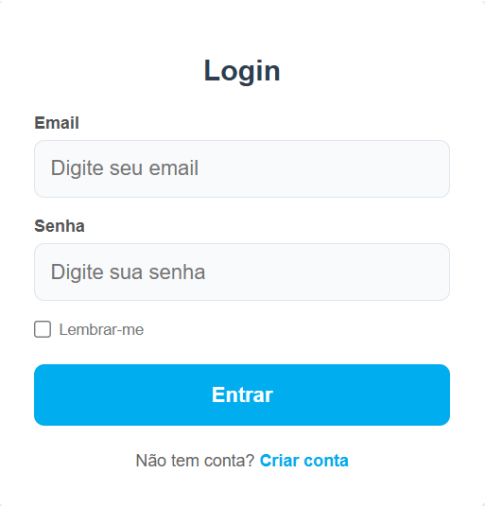
Foram criadas telas específicas para atender às necessidades dos dois principais tipos de usuários: cliente e administrador. O cliente possui acesso a funcionalidades como cadastro, *login*, visualização de produtos, realização de compras, consulta ao histórico de compras, acompanhamento de débitos e alteração de senha. Já o administrador possui acesso a recursos de gerenciamento, como aprovação de clientes, cadastro e atualização de produtos, controle de estoque, registro de pagamentos, consulta de pedidos, visualização de relatórios e acompanhamento do faturamento.

A comunicação entre a interface e o *backend* ocorre por meio do consumo dos *endpoints* da *API REST*, permitindo que as ações realizadas pelo usuário sejam processadas pelo servidor e persistidas no banco de dados. Dessa forma, a interface atua como a camada de interação entre o usuário final e as regras de negócio implementadas na aplicação, facilitando o uso do sistema e tornando o controle de vendas, débitos e pagamentos mais acessível.

3.15.1 *Login*

A tela de *login* é a tela inicial do sistema. Nessa tela, o usuário vai decidir se ele deseja realizar o *login* ou se cadastrar caso não tenha cadastro, conforme ilustra a Figura 23.

Figura 23 – Tela de *login*

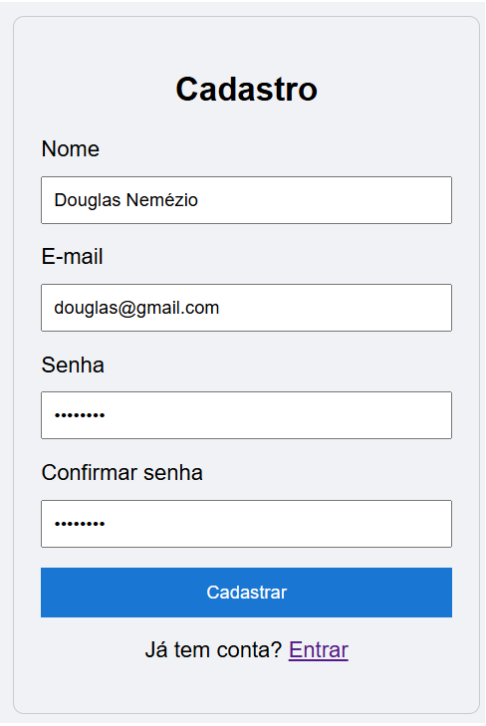


Autoria própria

3.15.2 Cadastro

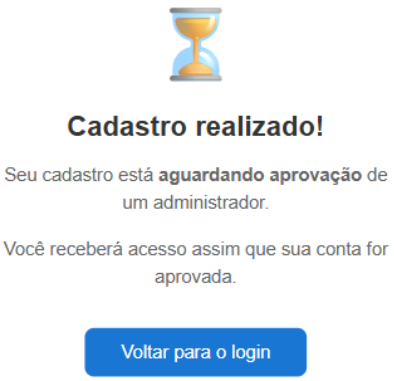
Essa tela contém as informações básicas para cadastro do usuário, como nome, *e-mail* e senha. Após o cadastro, o usuário é redirecionado para uma tela que informa que o cadastro foi realizado e está aguardando aprovação de um administrador. As Figuras 24 e 25 ilustram os detalhes.

Figura 24 – Formulário de cadastro



Autoria própria

Figura 25 – Mensagem



Autoria própria

3.15.3 Tela do Administrador

A tela do administrador centraliza as principais funcionalidades de gestão do sistema, sendo composta por: **Aceitar Clientes**, responsável por exibir os clientes cadastrados que aguardam aprovação do administrador para acessar o sistema; **Produtos**, onde é possível visualizar os produtos cadastrados e adicionar novos; **Clientes**, que lista todos os clientes já aprovados no sistema e o saldo de cada um, além de atualizar o saldo e excluir clientes; **Pedidos**, que exibe os pedidos realizados no mês atual; **Registrar Pagamentos**, para registrar os pagamentos feitos pelos clientes; e **Relatórios**, que apresenta o histórico completo de todos os pedidos registrados no sistema. Além disso, a tela exibe o faturamento do mês atual e do mês anterior, permitindo ao administrador acompanhar o desempenho financeiro da empresa de forma rápida e objetiva. É possível visualizar essas informações na Figura 26.

Figura 26 – Tela Administrador

Autoria própria

3.15.3.1 Lista de Produtos

Na tela de listagem de produtos, o administrador pode visualizar os produtos cadastrados e realizar ações de gerenciamento, como editar informações, atualizar o estoque e excluir registros. Essas funcionalidades permitem manter os dados dos produtos atualizados e contribuem para o controle adequado dos itens disponíveis para venda.

A funcionalidade de edição permite alterar informações do produto, como nome e valor. O controle de estoque possibilita atualizar a quantidade disponível, permitindo o acréscimo de unidades conforme a disponibilidade para comercialização. Já a ação de exclusão é realizada por meio de exclusão lógica, ou seja, o produto não é removido permanentemente do banco de dados. Nesse caso, o registro é apenas marcado como inativo, por meio da alteração do campo *active* para *false*. Essa abordagem permite preservar o histórico das informações e evitar inconsistências em pedidos já registrados. A Figura 27 ilustra a tela de listagem de produtos.

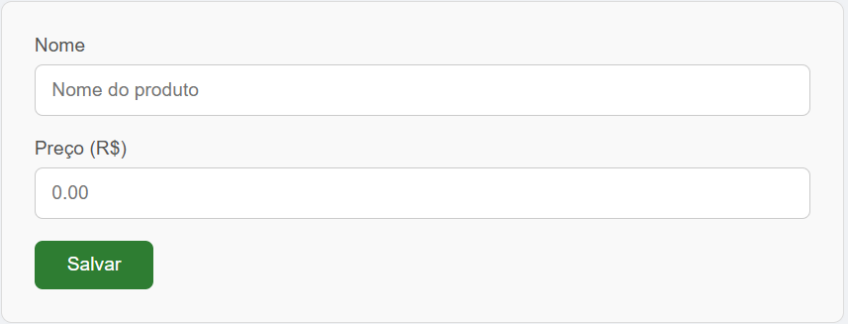
Figura 27 – Lista de produtos

Autoria própria

3.15.3.2 Cadastro de Produtos

Para cadastrar um novo produto, é necessário clicar no botão "Novo Produto"que se encontra na tela de listagem de produtos. Após isso, é aberta uma interface para cadastrar o produto com o nome do produto e o preço. Para finalizar o cadastro, basta clicar no botão de salvar conforme ilustra a Figura 28.

Figura 28 – Cadastrar Produto

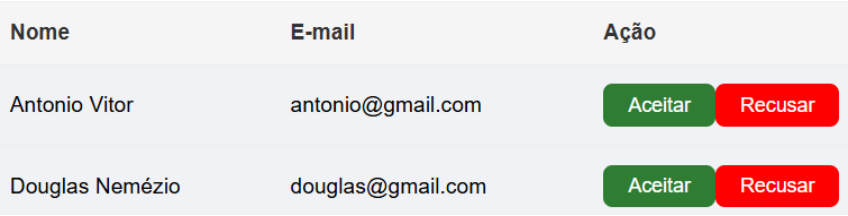


Autoria própria

3.15.3.3 Aceitar Usuário

Nessa tela de aceitar usuários, são mostrados todos os usuários pendentes de aprovação, o administrador tem acesso ao nome e *e-mail* do usuário, além de duas ações disponíveis que são a de aceitar e recusar usuário, conforme mostra a Figura 29. Caso o cliente venha a ser recusado, ele é deletado de forma definitiva do banco de dados.

Figura 29 – Aceitar Usuário



Autoria própria

3.15.3.4 Lista de Clientes

Na tela de listagem de usuários, ilustrada na Figura 30, que representam os clientes cadastrados no sistema, são exibidas informações como nome, *e-mail* e saldo devedor de cada cliente.

Figura 30 – Lista de Clientes

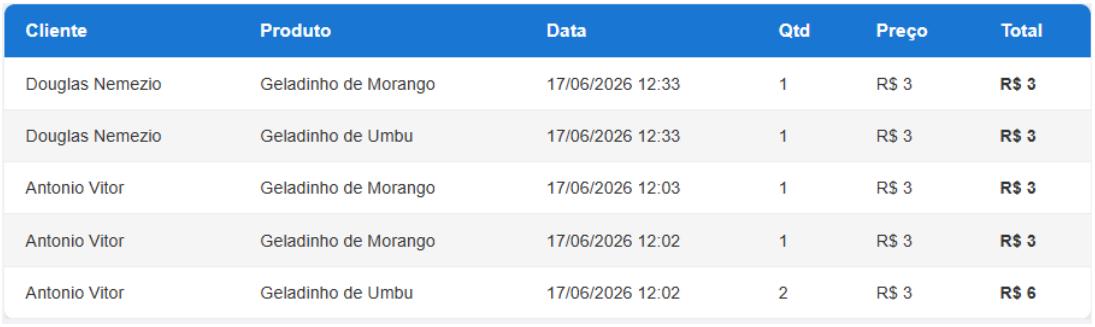
Autoria própria

Além disso, a interface disponibiliza duas ações: excluir e detalhes. A ação de excluir realiza a remoção lógica do usuário, mantendo o registro no banco de dados e alterando seu *status* para inativo. Já a ação de detalhes permite visualizar o histórico completo de compras do cliente selecionado, apresentando todas as transações realizadas por ele no sistema.

3.15.3.5 Pedidos

A tela de pedidos traz todos os pedidos do mês, desde o primeiro ao último pedido. Nessa tela, é exibida uma tabela contendo os dados do cliente, produto, data e hora da compra, quantidade, preço e o total daquela compra por produto, conforme ilustra a Figura 31.

Figura 31 – Histórico de Pedidos



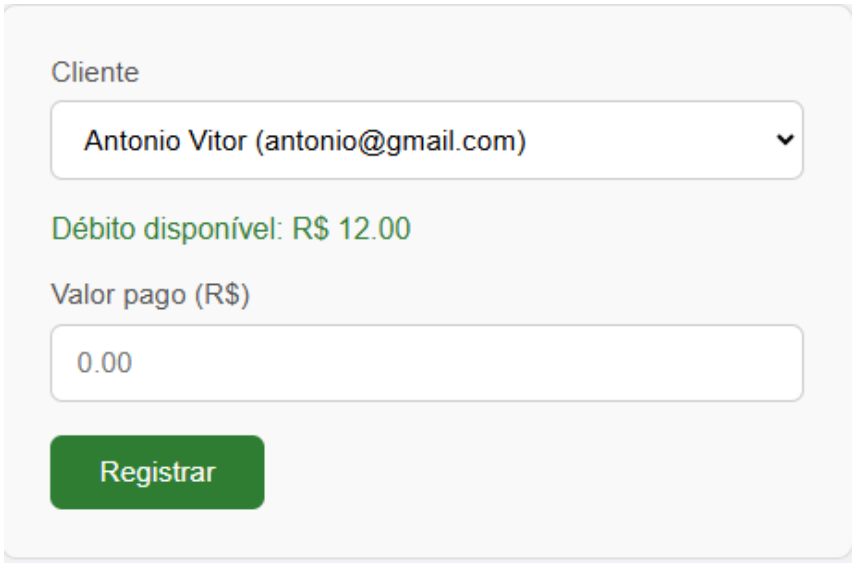
Autoria própria

3.15.4 Registrar Pagamento

Nessa tela, o administrador pode registrar os pagamentos realizados pelos clientes. A interface é composta por dois campos principais: o primeiro permite selecionar o cliente cadastrado no sistema, enquanto o segundo é destinado ao preenchimento do valor pago.

Após o registro, o pagamento é associado ao cliente selecionado e considerado no cálculo do seu saldo devedor. Dessa forma, a funcionalidade contribui para manter o controle dos débitos atualizado, permitindo acompanhar com mais precisão os valores pagos e os valores ainda pendentes. A Figura 32 apresenta a tela de registro de pagamento.

Figura 32 – Registro de pagamento

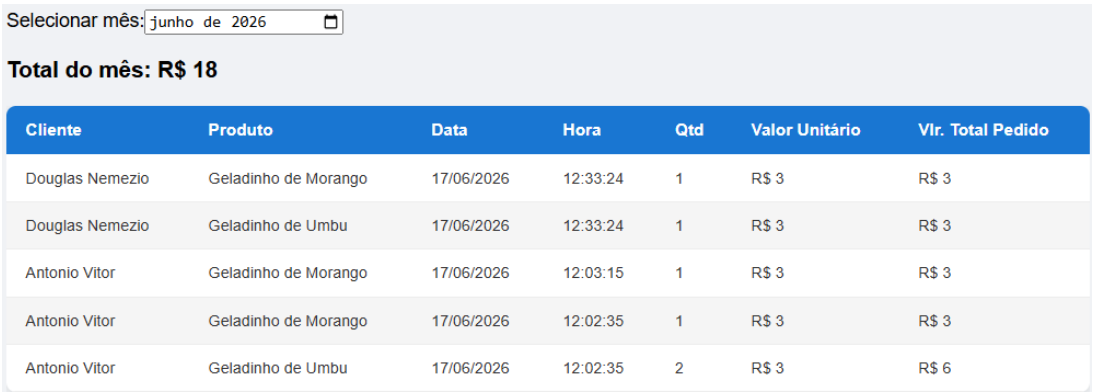


Autoria própria

3.15.5 Relatório

Na tela de relatórios, o administrador pode visualizar todos os pedidos realizados no sistema de acordo com o mês selecionado. Após a seleção do mês desejado, o sistema exibe o histórico completo das vendas realizadas no período correspondente, conforme ilustrado na Figura 33.

Figura 33 – Relatório



Cliente	Produto	Data	Hora	Qtd	Valor Unitário	Vlr. Total Pedido
Douglas Nemezio	Geladinho de Morango	17/06/2026	12:33:24	1	R$ 3	R$ 3
Douglas Nemezio	Geladinho de Umbu	17/06/2026	12:33:24	1	R$ 3	R$ 3
Antonio Vitor	Geladinho de Morango	17/06/2026	12:03:15	1	R$ 3	R$ 3
Antonio Vitor	Geladinho de Morango	17/06/2026	12:02:35	1	R$ 3	R$ 3
Antonio Vitor	Geladinho de Umbu	17/06/2026	12:02:35	2	R$ 3	R$ 6

Autoria própria

3.15.6 Tela Geral do Cliente

A tela inicial do cliente após o mesmo efetuar o *login* e ser autenticado é representada na Figura 34. Essa tela consta com um menu que disponibiliza **Ver Produtos** para visualizar os produtos disponíveis, o **Histórico de Compras** para mostrar todas as compras do cliente e **Trocar Senha** caso o usuário deseje trocar sua senha.

Figura 34 – Tela do Cliente

Autoria própria

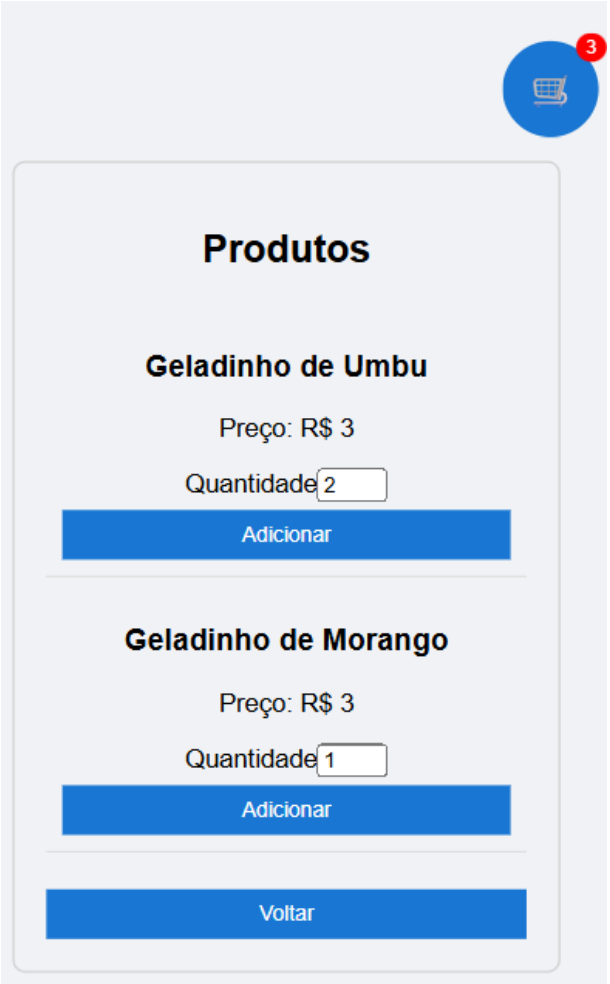
Além disso, a tela disponibiliza um componente de alerta visual, que indica a existência de débitos em aberto e o valor total devido. Caso não haja débito em aberto, o alerta irá mostrar que não há débito em aberto.

3.15.6.1 Ver Produtos

Nessa tela se encontram os produtos disponíveis para venda. O cliente adiciona no carrinho de compras quais ou qual produto ele deseja e a quantidade de cada um. O campo de quantidade permite que o cliente coloque apenas a quantidade disponível em estoque. A Figura 35 demonstra o que foi descrito.

Após adicionar os produtos no carrinho, o usuário decide se continua a compra ou se finaliza a compra.

Figura 35 – Produtos disponíveis



Autoria própria

3.15.6.2 Carrinho de compras

Após adicionar os produtos ao carrinho, o usuário é direcionado para a tela de resumo da compra. Nessa tela, são exibidos os itens selecionados, suas respectivas quantidades e os valores correspondentes, permitindo que o usuário revise as informações antes de finalizar o pedido. A partir dessa etapa, é possível continuar comprando, retornar à tela inicial ou confirmar a compra. A Figura 36 apresenta a tela do carrinho de compras.

Figura 36 – Carrinho de compras

Autoria própria

3.15.6.3 Histórico de compras

A tela de histórico de compras permite que o cliente acompanhe todas as compras realizadas no sistema. Nessa tela, são exibidas informações como o produto adquirido, a data e o horário da compra, a quantidade comprada, o valor unitário e o valor total do pedido. Essa funcionalidade contribui para que o cliente tenha maior controle sobre suas compras e possa consultar, sempre que necessário, os registros das transações realizadas. A Figura 37 apresenta a tela de histórico de compras do cliente.

Figura 37 – Histórico de compras

Produto	Data	Hora	Qtd	Valor Unitário	Vlr. Total Pedido
Geladinho de Morango	17/06/2026	12:02:35	1	R$ 3	R$ 3
Geladinho de Umbu	17/06/2026	12:02:35	2	R$ 3	R$ 6
Geladinho de Morango	17/06/2026	12:03:15	1	R$ 3	R$ 3

Autoria própria

3.15.6.4 Alteração de senha

Na tela de alteração de senha, o usuário deve informar sua senha atual, a nova senha e a confirmação da nova senha. Após o preenchimento dos campos, a alteração é solicitada por meio do botão salvar. Para que a senha seja atualizada, o sistema verifica se a senha atual informada está correta e se os campos referentes à nova senha são iguais. Caso alguma dessas validações não seja atendida, a alteração não é realizada, garantindo maior segurança no processo. A Figura 38 apresenta a tela de troca de senha.

Figura 38 – Troca de senha

Autoria própria

4 Conclusão e Trabalhos Futuros

A partir do problema identificado no controle manual de vendas, débitos de clientes e faturamento, este trabalho possibilitou o desenvolvimento de uma aplicação *web* voltada à organização dessas informações em um microcomércio. A solução desenvolvida atende ao objetivo proposto, pois permite registrar vendas, controlar os débitos dos clientes, gerenciar produtos, acompanhar pagamentos e visualizar informações relacionadas ao faturamento.

A utilização do sistema contribui para melhorar a organização dos dados financeiros, reduzir falhas associadas aos registros manuais e facilitar o acompanhamento das movimentações realizadas no negócio. Dessa forma, o processo de controle financeiro torna-se mais ágil, centralizado e acessível, favorecendo a consulta de informações e o acompanhamento das pendências dos clientes.

Além de atender à necessidade identificada no cenário de estudo, o desenvolvimento da aplicação demonstrou como a tecnologia pode apoiar pequenos negócios na gestão de suas informações. A centralização dos dados em um sistema permite consultas mais rápidas, melhor controle sobre vendas e pagamentos e maior apoio à tomada de decisões relacionadas ao funcionamento do microcomércio.

Como trabalhos futuros, o sistema pode ser aprimorado com a implementação da verificação de *e-mail* no momento do cadastro do usuário, permitindo confirmar a validade do endereço informado. Também poderá ser integrada uma funcionalidade de pagamento via Pix por *QR Code*, oferecendo ao cliente a possibilidade de realizar o pagamento imediato após a compra ou optar pela geração de débito para pagamento posterior. Outra melhoria possível é o envio automático de notificações de cobrança para clientes com débitos pendentes, por meio de *e-mail* ou *WhatsApp*, facilitando a comunicação e o acompanhamento das pendências financeiras.

Referências

ALMEIDA, R. M. G. de. Boas práticas em desenvolvimento de apis para integração de sistemas. *Lumen et Virtus*, v. 16, n. 45, p. 1636–1655, 2025.

ARAÚJO, S. d. S. Um estudo comparativo entre as apis para aplicações web: Restful, graphql e grpc. 2025.

FILHO, N. S. Modelagem de apis com rest e graphql: Estudo de caso em projetos. net. *Leaders Tec*, Leaders Tec, v. 2, n. 19, 2025.

GONÇALVES, V. T. Introdução à criação de apis: conceitos fundamentais e implementação em linguagem python. 2025.

JUNIOR, J. R. M. *Métodos de inteligência artificial em testes de software*. [S.l.]: Editora Senac São Paulo, 2025.

LEITE, M. D.; JÚNIOR, F. S. de S.; FONSECA, I. E. Análise de vulnerabilidades em aplicações que usam api rest.

MONTEIRO, B. F. A evolução do comércio eletrônico no brasil: impactos, desafios e oportunidades. 130, 2025.

MORAIS, E. C. et al. E-commerce invertido: Valorizando microempreendedores. 2024.

NAKAGAWA, W. T. Segurança em aplicação web: comparativo entre api mysql e api pdo. 004, 2017.

NUNES, C. E. S. Sistema para otimização de custos logísticos em e-commerce: Simulação de fretes por variação dimensional. *LUMEN ET VIRTUS*, v. 17, n. 58, p. e12531–e12531, 2026.

OLIVEIRA, G. C. P. d. Gennai api: desenvolvimento de uma api com a temática de digimon utilizando graphql. 2022.

OLIVEIRA, G. S. d.; RODRIGUES, L. F. Análise de desempenho de apis restful para e-commerce: comparação entre actix, express. js e gin. Universidade Estadual Paulista (Unesp), 2025.

PINHEIRO, F. F. N. Uma análise comparativa de performance entre requisições em apis rest e apis graphql utilizando javascript. 2022.

PONTES, H. R. P. Consumindo dados da indústria de petróleo e gás: uma implementação open-source de api restfull para comunicação com um witsml™ server. Universidade Federal do Rio Grande do Norte, 2022.

RAPÔSO, C. F. L. et al. Divergências e convergências entre arquitetura de software e arquitetura cloud: Uma análise crítica. *Revista Tópicos*, v. 3, n. 19, p. 1–17, 2025.

REZENDE, G. B. S. d. C. et al. Renovar: Desenvolvimento de uma application programming interface (api) rest com kotlin e spring boot para gestão de funcionários, ferramentas e equipamentos de proteção individual (epis) em obras. Instituto Federal Goiano, 2025.

RIBEIRO, L. Graphql: Uma alternativa aos webservices graphql: An alternative to webservices. *Brazilian Journal of Development*, v. 7, n. 12, p. 119574–119618, 2021.

TAQUIWI, G. A. S.; DIAS, L. S.; BARBOSA, G. V. B. Integração de api com banco de dados e seus desafios. *CADERNO DE RESUMOS*, p. 80, 2024.

TWARDOWSKI, A. R.; RIDER, A. P. et al. A gestão e o controle de estoques no sistema de e-commerce: Estudo de caso nas empresas artys e closet rider. Florianópolis, SC., 2023.

VALE, P. T. P. d. Comparação entre arquiteturas de software monolítica e de microsserviços. 2025.

VAROTO, A. C. *Visões em arquitetura de software*. Tese (Doutorado) — Universidade de São Paulo, 2002.