



UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO, DE CIÊNCIAS EXATAS E EDUCAÇÃO
DEPARTAMENTO DE ENG. DE CONTROLE, AUTOMAÇÃO E COMPUTAÇÃO
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Klaus Dieter Kupper

Estudo para Monitoramento do Nivel de Vias Fluviais

Itajai
2025

Klaus Dieter Kupper

Estudo para Monitoramento do Nivel de Vias Fluviais

Trabalho de Conclusão de Curso de Graduação em Engenharia de Controle e Automação do Centro Tecnológico, de Ciências Exatas e Educação da Universidade Federal de Santa Catarina como requisito para a obtenção do título de Engenheiro de Controle e Automação.
Orientador: Prof. Dr. Jordan Sausen

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Kupper, Klaus Dieter

Estudo e Implementação de um Sistema Web para Vendas e
Pagamentos / Klaus Dieter Kupper ; orientador, Carlos
Roberto Moratelli, 2023.

52 p.

Trabalho de Conclusão de Curso (graduação) -
Universidade Federal de Santa Catarina, Campus Blumenau,
Graduação em Engenharia de Controle e Automação, Blumenau,
2023.

Inclui referências.

1. Engenharia de Controle e Automação. 2. Internet of
Things. 3. Automação de Pedidos. 4. Desenvolvimento Web. 5.
Sockets. I. Moratelli, Carlos Roberto. II. Universidade
Federal de Santa Catarina. Graduação em Engenharia de
Controle e Automação. III. Título.

Klaus Dieter Kupper

Estudo para Monitoramento do Nivel de Vias Fluviais

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de “Engenheiro de Controle e Automação” e aprovado em sua forma final pelo Curso de Graduação em Engenharia de Controle e Automação.

Itajai, 10 de Julho de 2025.

Banca Examinadora:

Prof. Dr. Carlos Roberto Moratelli
Universidade Federal de Santa Catarina

Prof. Dr. Maiquel de Brito
Universidade Federal de Santa Catarina

Prof. Dr. Ciro André Pitz
Universidade Federal de Santa Catarina

Este trabalho é dedicado aos meus colegas de classe, a
minha namorada e aos meus queridos pais.

AGRADECIMENTOS

Gostaria de expressar minha profunda gratidão ao meu professor orientador, pela sua orientação acadêmica e apoio ao longo deste trabalho. Suas orientações sábias e paciência foram fundamentais para o meu desenvolvimento.

A todos os professores que tive ao longo da minha jornada acadêmica, sou imensamente grato. Cada aula, conselho e palavra contribuíram para o meu crescimento pessoal e profissional. Em especial, agradeço por terem despertado em mim o interesse pela engenharia e tecnologia, que hoje são a minha paixão e a área em que atuo.

Agradeço também aos meus pais, cujo amor, apoio incondicional e crença em mim foram essenciais para que eu chegasse até aqui. Vocês foram minha inspiração e força motriz em todos os momentos desafiadores.

Aos meus colegas, expresso minha gratidão pelo apoio, compreensão e companheirismo ao longo dessa jornada. Nossos momentos de estudo, discussões e desafios compartilhados foram fundamentais para o meu crescimento e formação.

Por fim, gostaria de estender meu agradecimento a todos que, de alguma forma, contribuíram para a minha formação. Cada gesto e palavra de apoio foram importantes para o meu crescimento como pessoa e profissional.

"O computador é a bicicleta da mente." (Steve Jobs, 1985)

RESUMO

Este trabalho apresenta o desenvolvimento de um sistema para automação de pedidos e compras em pontos de venda, integrando tecnologias modernas de web e IoT. O objetivo é otimizar o processo de compras, tornando-o mais eficiente e seguro, além de demonstrar a viabilidade da integração de tecnologias web e IoT em aplicações comerciais. A aplicação utiliza WebSocket, ReactJS, tRPC, NodeJS e NextJs, juntamente com a API de pagamentos do Mercado Pago. Os clientes acessam os produtos através de um link exclusivo de código QR, realizando pagamentos de forma segura e rápida. Um componente crucial é a integração com dispositivos para a entrega dos pedidos, através de notificações via WebSocket. Com base nos resultados, o sistema desenvolvido mostrou-se eficiente e viável, permitindo a automação de processos e gestão de pedidos.

Palavras-chave: Automação de Pedidos; Internet das Coisas; NodeMCU ESP8266; Aplicação Web Full-Stack; API de Pagamentos do Mercado Pago.

ABSTRACT

This work presents the development of a system for automating orders and purchases at points of sale, integrating modern web and IoT technologies. The objective is to optimize the purchasing process, making it more efficient and secure, in addition to demonstrating the viability of integrating web and IoT technologies in commercial applications. The application uses WebSocket, ReactJS, tRPC, NodeJS, and NextJs, along with Mercado Pago's payment API. Customers access the products through an exclusive QR code link, making payments quickly and securely. A crucial component is the integration with devices for order delivery, through notifications via WebSocket. Based on the results, the developed system proved to be efficient and viable, allowing for process automation and order management.

Keywords: Order Automation; Internet of Things; NodeMCU ESP8266; Full-Stack Web Application; Mercado Pago Payment API.

LISTA DE FIGURAS

Figura 1 – Buscando dados da Web.	17
Figura 2 – Camadas da <i>web</i>	17
Figura 3 – API Rest.	19
Figura 4 – QR Code com link para loja.	21
Figura 5 – Internet of Things.	22
Figura 6 – ESP8266 NodeMcu.	27
Figura 7 – Arquitetura do Sistema.	28
Figura 8 – ERD - Projeto.	32
Figura 9 – Fluxograma de compra.	35
Figura 10 – Página da loja.	36
Figura 11 – Fluxograma administrativo.	36
Figura 12 – Fluxograma do controle de um ponto de vendas.	37
Figura 13 – Tela de controle de PDVs.	37
Figura 14 – Tela para adicionar Item.	38
Figura 15 – Diagrama da conexão NodeMCU e servidor.	41
Figura 16 – Tela de pedidos.	42
Figura 17 – QR code para a loja de testes.	43
Figura 18 – Loja de testes.	44
Figura 19 – Mensagem ao finalizar compra.	44
Figura 20 – Tela de pagamento.	45
Figura 21 – NodeMCU ao identificar ID esperado no socket.	46

LISTA DE TABELAS

Tabela 2 – Preços dos planos de hospedagem em nuvem.	47
--------------------------------------------------------------	----

LISTA DE ABREVIATURAS E SIGLAS

API	<i>Application Programming Interface</i>
CORS	<i>Cross-origin Resource Sharing</i>
ERD	<i>Entity-Relationship Diagram</i>
HTML	<i>HyperText Markup Language</i>
HTTP	<i>Hypertext Transfer Protocol</i>
IA	Inteligência Artificial
IoT	<i>Internet of Things</i>
JSON	<i>JavaScript Object Notation</i>
JWT	<i>JSON Web Token</i>
ORM	<i>Object-Relational Mapping</i>
QR	<i>Quick Response</i>
REST	<i>Representational State Transfer</i>
SQL	<i>Structured Query Language</i>
tRPC	<i>Typescript Remote Procedure Call</i>
UUID	<i>Universal Unique Identifier</i>
WWW	<i>World Wide Web</i>

SUMÁRIO

1	INTRODUÇÃO	14
1.1	OBJETIVO GERAL	14
1.2	OBJETIVOS ESPECÍFICOS	15
1.3	ESTRUTURA DO DOCUMENTO	15
2	FUNDAMENTAÇÃO TEÓRICA	16
2.1	AUTOMAÇÃO DE SERVIÇOS	16
2.2	COMUNICAÇÃO HTTP E WEB	16
2.3	API	18
2.3.1	REST API	18
2.3.2	APIs de Pagamento	19
2.4	SOCKETS E WEBSOCKETS	20
2.5	TOKENS E JWT	20
2.6	QR CODE	20
2.7	INTERNET DAS COISAS (IOT)	21
2.8	UUID	22
2.9	TECNOLOGIAS DA APLICAÇÃO WEB	22
2.9.1	TypeScript	22
2.9.2	SQL e Prisma	23
2.9.3	Zod	24
2.9.4	tRPC	24
2.9.5	Fastify	25
2.9.6	Next.js	25
2.10	NODEMCU ESP8266	26
2.10.1	C++ e Bibliotecas	27
3	ARQUITETURA DO SISTEMA	28
3.1	APLICAÇÃO WEB FULLSTACK	28
3.2	SERVIDOR DE SOCKETS	29
3.3	AGENTE DE DISTRIBUIÇÃO	30
3.4	MERCADO PAGO	30
3.5	BASE DE DADOS	30
4	IMPLEMENTAÇÃO	31
4.1	BANCO DE DADOS	31
4.2	APLICAÇÃO WEB FULL STACK	34
4.2.1	Fluxo de Compras	34
4.2.2	Fluxo de Administração do Sistema	34
4.2.3	Fluxo de Administração de Um Ponto de Vendas	35
4.2.4	Frontend	35

4.2.5	<i>Backend</i>	38
4.3	SERVIDOR DE <i>SOCKETS</i>	39
4.3.1	Configurações iniciais e Criação do servidor <i>websocket</i>	40
4.3.2	Gerenciamento e Atualização de conexões <i>websocket</i>	40
4.3.3	Roteamento HTTP e Socket	40
4.4	AGENTE DE DISTRIBUIÇÃO	41
5	RESULTADOS	43
5.1	RESULTADOS E TESTES DA IMPLANTAÇÃO EM AMBIENTE DE PRODUÇÃO	43
5.2	HOSPEDAGEM E CUSTOS	45
5.3	CONSIDERAÇÕES FINAIS	46
6	CONCLUSÃO	48
	REFERÊNCIAS	49

1 INTRODUÇÃO

A ascensão do comércio eletrônico e dos pagamentos online, que se estabeleceu no início dos anos 2000, revolucionou a maneira como as transações são realizadas. A necessidade de eficiência e redução de custos nas operações de venda levou ao surgimento de sistemas automatizados de pedidos e compras. Esses sistemas, aliados à expansão da IoT (*Internet of Things*), abriram novas possibilidades para aprimorar a experiência do cliente e a eficiência operacional dos negócios.

Este projeto tem como visão simplificar o processo de integrar plataformas de pagamento a projetos de menor escala ou que tenham recursos e tempo limitados. Para isso, deve disponibilizar conexões do tipo *socket* para acompanhamento dos pagamentos em tempo real. Além disso, deve fornecer uma plataforma completa, pronta para o registro de itens e para a efetuação de pagamentos.

Nesse contexto, o presente trabalho propõe o desenvolvimento de uma aplicação *web* que integra tecnologias como *websocket*, ReactJS, tRPC (*Typescript Remote Procedure Call*), NodeJS e TypeScript com a API (*Application Programming Interface*) de pagamentos do Mercado Pago. A aplicação proposta será um sistema de automação de pedidos e compras para pontos de venda. Os clientes poderão acessar os produtos disponíveis em cada estabelecimento por meio de um link com código QR (*Quick Response*) exclusivo. Através da API de pagamentos do Mercado Pago, os usuários poderão realizar pagamentos de maneira segura e rápida.

Um componente crucial deste sistema é a integração com agentes de distribuição, que são outros dispositivos ou sistemas que possam estar conectados via *sockets* neste projeto. Para o presente trabalho foi escolhido um dispositivo baseado na plataforma NodeMCU ESP8266. Este dispositivo será programado em C++ para se comunicar com a aplicação e demonstrar a versatilidade dessa plataforma para a entrega dos pedidos. Isso inclui receber notificações via *websocket* quando um pagamento é realizado, permitindo que o dispositivo IoT entregue um produto ou serviço de acordo com a necessidade de cada projeto. Além disso, a aplicação também incluirá uma interface para funcionários das empresas que utilizarem o sistema. Esta interface permitirá a confirmação e entrega de pedidos, cadastro de itens, bem como o monitoramento do status dos pedidos. O acesso a essa interface será controlado por *login* e senha.

Este trabalho contribuirá para a compreensão e aplicação das tecnologias mencionadas, além de demonstrar a viabilidade e utilidade de uma solução integrada de automação de pedidos e compras em pontos de venda.

1.1 OBJETIVO GERAL

O objetivo geral deste projeto é desenvolver uma solução de automação de pedidos e compras em pontos de venda utilizando tecnologias *web* modernas, integradas com um

dispositivo IoT. A solução visa otimizar o processo de compras em pontos de venda, tornando-o mais eficiente e seguro para os usuários e estabelecimentos. Além disso, o projeto tem como objetivo demonstrar a viabilidade e utilidade da integração de tecnologias *web* e IoT em aplicações comerciais, contribuindo para a disseminação e adoção dessas tecnologias no mercado.

1.2 OBJETIVOS ESPECÍFICOS

1. Desenvolver uma aplicação *web full-stack*, ou seja, uma aplicação que integra a interface (*frontend*) e o servidor (*backend*).
2. Desenvolver uma aplicação capaz de se comunicar com múltiplos dispositivos via *sockets*.
3. Integrar o sistema com APIs de pagamento.
4. Garantir segurança nos pagamentos online.
5. Permitir pagamento via dispositivos móveis.
6. Implementar interface de gerenciamento.
7. Demonstrar a comunicação do sistema com um agente de distribuição.
8. Determinar as tecnologias mais adequadas para implementação da solução.

1.3 ESTRUTURA DO DOCUMENTO

Este trabalho está dividido em 6 capítulos. O Capítulo 1 apresenta a introdução e objetivos. O Capítulo 2 trata da fundamentação teórica do trabalho. O Capítulo 3 apresenta a arquitetura proposta e define seus elementos. O Capítulo 4 aborda a implementação e as tecnologias aplicadas no desenvolvimento. O Capítulo 5 apresenta os resultados obtidos e estimativa de custos. O Capítulo 6 traz a conclusão.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção é dedicada à exploração e discussão dos conceitos e teorias que embasam este trabalho. Neste capítulo, serão apresentados os principais temas e tecnologias envolvidos, como HTTP, NodeJS, *websockets*, e APIs. A discussão será organizada de maneira progressiva, começando pelos conceitos mais gerais e avançando para os aspectos mais específicos e técnicos, como as bibliotecas e *frameworks* utilizados.

2.1 AUTOMAÇÃO DE SERVIÇOS

Automação de processos, se refere ao uso de tecnologia para realizar tarefas rotineiras e repetitivas de maneira eficiente e sem erros, e é um aspecto fundamental da automação de serviços.

A automação de serviços é uma tendência em crescimento nos últimos anos, especialmente na área de comércio eletrônico. Ela busca tornar processos mais eficientes, reduzir custos e oferecer melhores experiências aos clientes. Através da automatização, é possível reduzir erros e aumentar a precisão e rapidez no atendimento ao cliente. Com isso, os estabelecimentos podem oferecer um serviço de qualidade, com maior agilidade e menor tempo de espera.

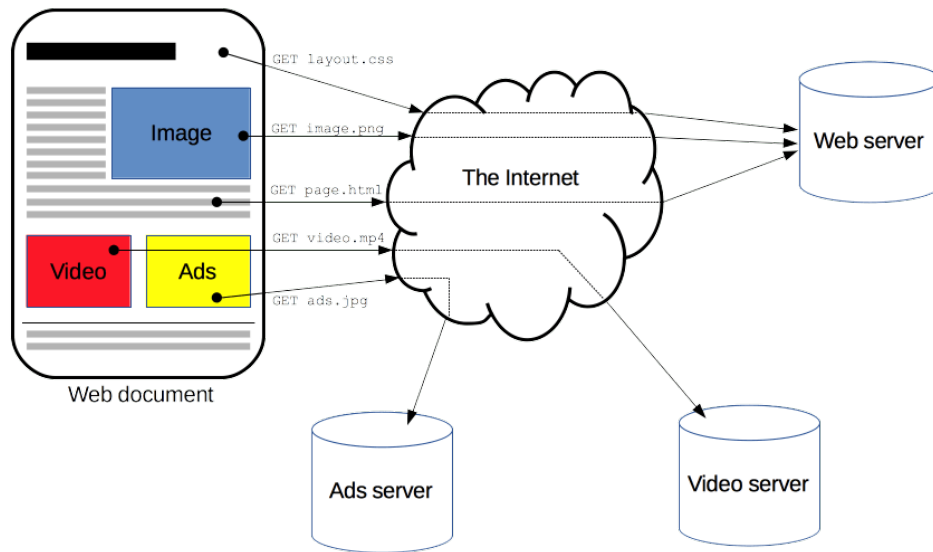
A pandemia de COVID-19 acelerou a necessidade de digitalização em muitos setores. Com as restrições de movimento e o fechamento de lojas físicas, muitos negócios tiveram que se adaptar rapidamente para oferecer seus serviços online. Isso levou a um aumento na demanda por automação de serviços, à medida que as empresas procuravam maneiras de continuar operando de forma eficiente em um ambiente digital. De acordo com um estudo de Sousa, Silva e Araújo (2008), a automação não é apenas uma parte do processo; é um ecossistema em desenvolvimento em que a IA (Inteligência Artificial) e a IoT trabalham juntas para criar uma força de trabalho mais inteligente, eficiente e produtiva (Gourley; Totty, 2002).

2.2 COMUNICAÇÃO HTTP E WEB

O protocolo HTTP (*Hypertext Transfer Protocol*) é um dos principais métodos para transferência de dados via *web*. Trata-se de um protocolo de cliente-servidor, o que significa que as solicitações são iniciadas pelo cliente, geralmente o navegador da *web*, e um documento completo é reconstruído a partir dos diferentes recursos buscados, como texto, descrição de *layout*, imagens, vídeos, *scripts* e mais (HTTP..., s.d.), a Figura 1 ilustra esse processo.

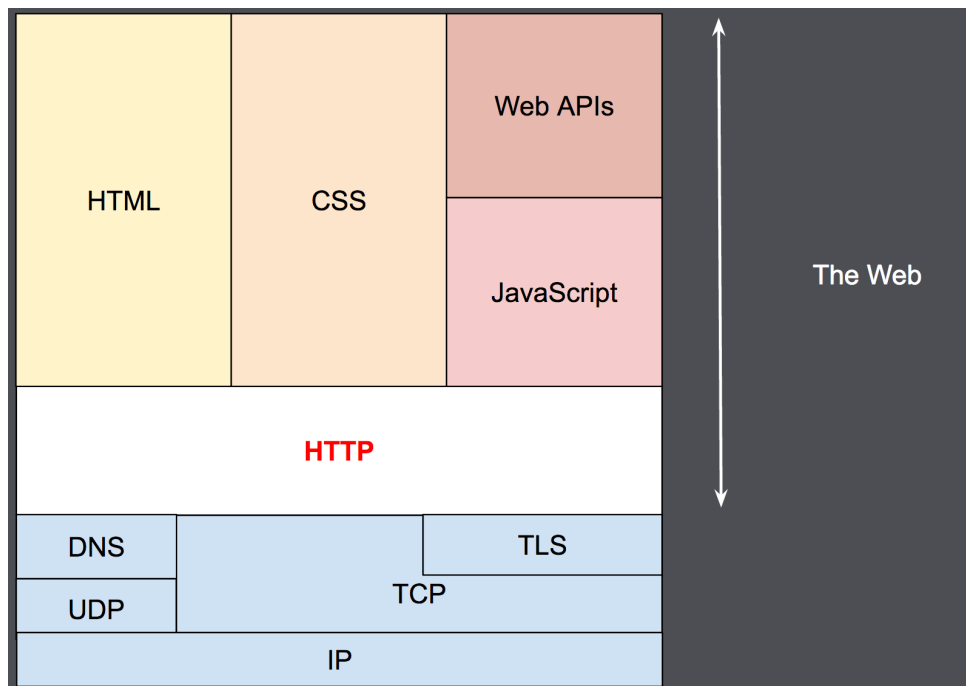
O HTTP é um protocolo sem estado, o que significa que cada requisição é independente das outras. Isso permite que a *web* seja altamente escalável, pois os servidores não precisam manter informações sobre cada usuário.

Figura 1 – Buscando dados da Web.



Fonte: (HTTP..., s.d.).

A WWW foi criada em 1989 por Tim Berners-Lee, um cientista da computação britânico que trabalhava no CERN, o laboratório de física de partículas na Suíça. O objetivo era criar uma maneira fácil de compartilhar informações entre os cientistas que trabalhavam em diferentes universidades e institutos ao redor do mundo. O HTML foi a linguagem de marcação que Berners-Lee desenvolveu para criar páginas *web*. A Figura 2 demonstra a conexão entre estas tecnologias e como elas se encaixam para produzir a *web*.

Figura 2 – Camadas da *web*.

Fonte: (HTTP..., s.d.).

Com o tempo, o HTTP evoluiu e novas versões foram lançadas. A versão mais recente é o HTTP/3, que oferece melhor desempenho e segurança em comparação com as versões anteriores. No entanto, o HTTP/1.1 e HTTP/2 ainda são amplamente utilizados na *web* (Elhadeh; Grira, 2016).

Um conceito importante para a comunicação HTTP são os métodos, os principais métodos HTTP são:

- **GET:** Solicita um recurso do servidor. Este é o método mais comum e é usado para solicitar a visualização de páginas da web.
- **POST:** Envia dados para o servidor para criar um novo recurso. Os dados são incluídos no corpo da solicitação.
- **PUT:** Envia dados para o servidor para atualizar um recurso existente. Os dados são incluídos no corpo da solicitação.
- **DELETE:** Solicita a exclusão de um recurso no servidor.
- **OPTIONS:** Solicita informações sobre os métodos de comunicação disponíveis para um recurso ou para o servidor em geral.
- **PATCH:** Aplica modificações parciais a um recurso.
- **CONNECT:** É usado para abrir uma conexão de rede bidirecional com o recurso solicitado. Geralmente é usado para acesso SSL (HTTPS).

Esses métodos são definidos no protocolo HTTP e são usados para indicar a ação desejada a ser realizada no recurso especificado. Eles formam a base da interação entre o cliente e o servidor na web (Elhadeh; Grira, 2016).

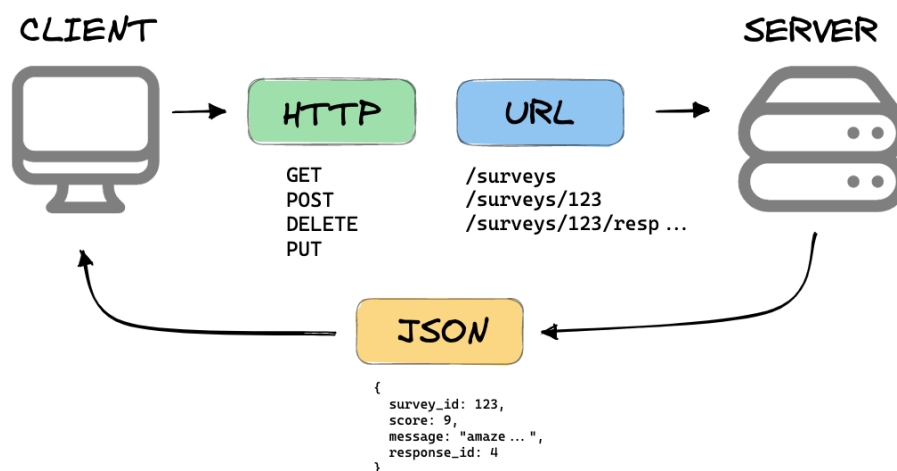
2.3 API

Uma API (*Application Programming Interface*) é um conjunto de rotinas, protocolos e ferramentas para construção de software e aplicações. Ela define como os componentes de software devem interagir entre si, permitindo que diferentes aplicações possam se comunicar e compartilhar informações (Mulloy, 2012).

2.3.1 REST API

REST (*Representational State Transfer*) é um estilo arquitetural para sistemas distribuídos, baseado no protocolo HTTP. Ele define um conjunto de restrições que devem ser seguidas para que as aplicações possam se comunicar de forma eficiente e escalável. Uma das principais características do REST é a separação entre cliente e servidor, onde o cliente faz requisições ao servidor para acessar recursos, e o servidor responde com uma representação do estado atual do recurso solicitado. A Figura 3 mostra a arquitetura de uma REST API.

Figura 3 – API Rest.



Fonte: (restAPI).

O REST é um elemento fundamental na construção de APIs modernas, devido à sua simplicidade e eficiência. Ele permite que os desenvolvedores criem APIs que podem ser facilmente consumidas por diferentes clientes, incluindo navegadores web, aplicativos móveis e outros servidores. Além disso, o REST é independente de linguagem, o que significa que pode ser usado com qualquer linguagem de programação que suporte HTTP.

O REST foi proposto por Roy Fielding em sua tese de doutorado em 2000. Fielding é um dos principais contribuidores para o desenvolvimento do protocolo HTTP e co-fundador da Apache HTTP Server Project. Em sua tese, Fielding descreveu o REST como um conjunto de princípios arquiteturais que podem ser usados para projetar sistemas distribuídos que são escaláveis, eficientes e fáceis de modificar e manter.

Os métodos HTTP citados na Seção 2.2 são incorporados ao REST. Eles formam um estilo arquitetural que utiliza métodos com a mesma semântica para permitir a construção de APIs, com rotas do tipo POST, GET, DELETE etc.

Desde então, o REST se tornou o estilo arquitetural mais popular para a construção de APIs na web. Ele é usado por muitas grandes empresas, incluindo Google, Facebook e Twitter, para fornecer acesso programático aos seus serviços (Richardson; Ruby, 2007).

2.3.2 APIs de Pagamento

As APIs de pagamento surgiram como uma necessidade para facilitar as transações online. No início, as APIs de pagamento eram principalmente usadas para processar pagamentos com cartão de crédito. Empresas como a PayPal foram pioneiras nesse campo, fornecendo APIs que permitiam aos comerciantes aceitar pagamentos com cartão de crédito em seus sites.

Com o tempo, as APIs de pagamento evoluíram para suportar uma variedade de métodos de pagamento. Isso inclui não apenas cartões de crédito, mas também débito

direto, pagamentos móveis e até mesmo cripto-moedas. Além disso, as APIs de pagamento também começaram a oferecer funcionalidades adicionais, como suporte para pagamentos recorrentes, reembolsos, e a capacidade de gerenciar várias moedas.

No Brasil, intermediários de pagamento como o Mercado Pago e o PagSeguro oferecem APIs que permitem aos comerciantes aceitar uma variedade de métodos de pagamento, incluindo boleto bancário e transferências bancárias. Recentemente, com a introdução do PIX, um sistema de pagamentos instantâneos operado pelo Banco Central do Brasil, esses intermediários também começaram a oferecer APIs que suportam PIX, permitindo transações quase instantâneas (Aué et al., 2018; ADYEN..., s.d.).

2.4 SOCKETS E WEBSOCKETS

Os *sockets* e *webSockets* são tecnologias fundamentais para a comunicação em tempo real na internet. *Sockets* são um mecanismo de comunicação bidirecional entre dois nós em uma rede, permitindo a troca de dados em tempo real. Eles são amplamente utilizados em sistemas distribuídos e aplicações de rede para estabelecer conexões entre servidores e clientes (Attoui, 2000).

Os *websockets*, por outro lado, são uma extensão dos *sockets*, projetados especificamente para comunicação em tempo real na web. Eles superam as limitações das soluções tradicionais de comunicação em tempo real na web, como *polling* e *long-polling*, fornecendo um mecanismo eficaz para comunicação bidirecional sustentada entre o cliente e o servidor. A tecnologia *websockets* permite uma comunicação mais eficiente, reduzindo o tráfego de rede e a latência, tornando-a ideal para aplicações que exigem interações em tempo real (Liu; Sun, 2012).

2.5 TOKENS E JWT

A autenticação e a autorização são componentes críticos de qualquer aplicação segura. *Tokens*, particularmente JWT (*JSON Web Token*), são usados para transmitir informações de forma segura entre partes. JWT é um padrão aberto (RFC 7519) que define uma maneira compacta e independente de transmitir informações entre partes como um JSON. Essas informações podem ser verificadas e confiáveis porque são assinadas digitalmente. JWT pode ser assinado usando um segredo (com o algoritmo HMAC) ou um par de chaves pública/privada usando RSA ou ECDSA (INTRODUCTION..., 2023).

2.6 QR CODE

O código QR (*Quick Response*) é um código de barras bidimensional que pode armazenar informações em um formato legível por máquina. Ele foi desenvolvido para

permitir a leitura rápida e eficiente de informações por dispositivos eletrônicos, como *smartphones* e *tablets*.

Amplamente utilizado em várias aplicações, o QR Code se tornou muito comum no rastreamento de produtos, gerenciamento de inventário e marketing (QR. . . , 2023). A Figura 4 demonstra um QR que foi gerado com um *link*.

Figura 4 – QR Code com link para loja.



Fonte: Criado pelo autor.

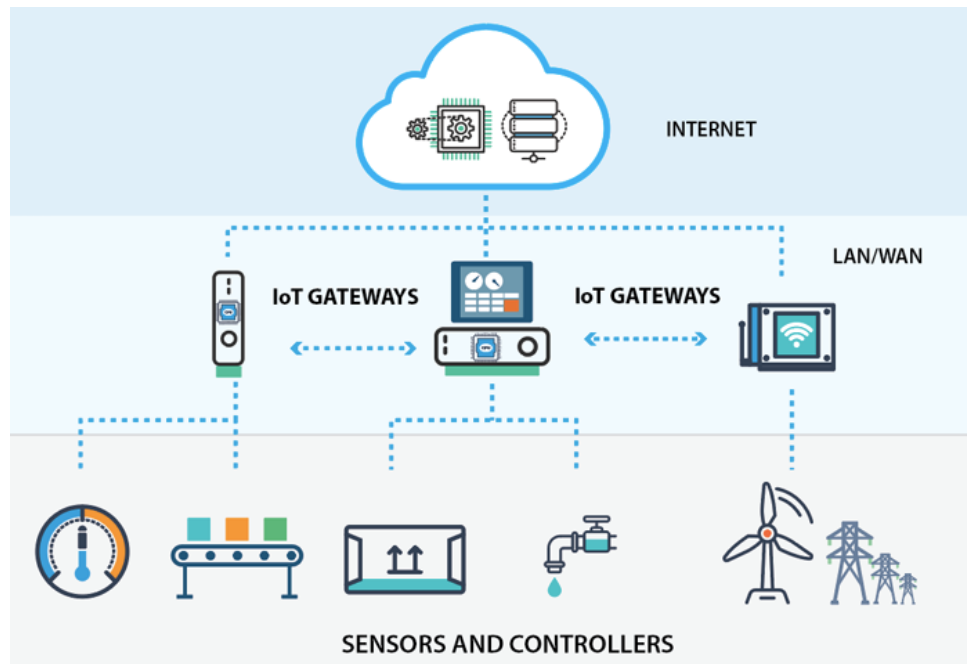
Esta codificação é gerada através de um processo que transforma a informação desejada (como um *link* para um site) em um padrão de pontos pretos e brancos. Este padrão é lido por um *scanner* (geralmente uma câmera de *smartphones* com um aplicativo de leitura de QR Code), que decodifica a informação e realiza a ação correspondente (como abrir um *link* em um navegador).

2.7 INTERNET DAS COISAS (IOT)

IoT é discutido como uma tecnologia emergente que transforma objetos do mundo real em objetos virtuais inteligentes. IoT visa unificar tudo em nosso mundo sob uma infraestrutura comum, não apenas nos dando controle sobre as coisas ao nosso redor, mas também nos mantendo informados sobre o estado dessas coisas (Madakam; Ramaswamy; Tripathi, 2015).

Isso torna possível a criação de dispositivos conectados a internet, capazes de monitorar e controlar processos e equipamentos remotamente. IoT tem um papel crucial na transformação digital e na criação de cidades inteligentes, permitindo a coleta de dados em tempo real e a tomada de decisões baseada em dados. A Figura 5 representa como diversos dispositivos se conectam a internet através da IoT.

Figura 5 – Internet of Things.



Fonte: (IOT..., 2023).

2.8 UUID

Um UUID (*Universal Unique Identifier*) é um identificador único de 128 bits que é usado para identificar informações de forma única em um sistema de computação distribuído (Leach; Mealling; Salz, 2005). Ele é gerado de forma aleatória, tornando-o altamente improvável de ser duplicado. O formato de UUID mais comum é o UUID versão 4, que utiliza a geração aleatória de números para criar uma identificação única. O UUID é amplamente utilizado em sistemas distribuídos, como bancos de dados, sistemas de mensagens e sistemas de arquivos distribuídos (Leach; Mealling; Salz, 2005).

Essa ferramenta permite que cada elemento seja rastreado e gerenciado de forma única, evitando conflitos ou duplicatas no sistema. Além disso, aumenta a segurança, visto que dificulta tentativas, por parte de usuários mal intencionados, de manipular algum elemento do sistema.

2.9 TECNOLOGIAS DA APLICAÇÃO WEB

Neste trabalho, serão utilizadas diversas tecnologias e bibliotecas. As mais importantes são explicadas nas seções a seguir.

2.9.1 TypeScript

TypeScript é uma linguagem de programação que estende o JavaScript, adicionando tipos estáticos. Sua principal motivação é permitir o desenvolvimento de aplicações

JavaScript em larga escala de maneira mais eficiente. O TypeScript introduz um sistema de módulos, classes e interfaces, além de um sistema de tipos gradual e robusto. Essas características permitem que os desenvolvedores escrevam código JavaScript de maneira mais clara e estruturada, facilitando a compreensão e a manutenção do código (Bierman; Abadi; Torgersen, s.d., p. 257).

O TypeScript oferece ferramentas que auxiliam na construção do código, como a capacidade de listar os campos presentes em um objeto ou todos os métodos de uma classe. Isso facilita a navegação e a manipulação do código, especialmente em projetos de grande escala. Além disso, o TypeScript, por meio de seu sistema de tipos estáticos, é capaz de fornecer garantias sobre o comportamento do código, assegurando que os tipos de dados sejam consistentes ao longo do código, o que pode resultar em software mais confiável e de maior qualidade. Essas características tornam o TypeScript uma escolha popular para o desenvolvimento de aplicações web de grande escala, tanto para front-end quanto para back-end.

2.9.2 SQL e Prisma

SQL (*Structured Query Language*) é uma linguagem de programação utilizada para gerenciar e manipular bancos de dados. Ela permite aos usuários criar, ler, atualizar e deletar dados em um banco de dados. SQL é uma linguagem padrão para bancos de dados relacionais e é usada em muitos sistemas de gerenciamento de bancos de dados, como MySQL, Oracle, PostgreSQL e SQL Server. SQL é uma linguagem essencial para desenvolvedores de software, analistas de dados e administradores de banco de dados, pois permite a interação eficiente com os dados armazenados em um banco de dados (SQL..., 2023).

No entanto, trabalhar diretamente com SQL pode ser complexo e propenso a erros. Para resolver isso, os desenvolvedores usam ferramentas chamadas mapeadores objeto-relacional (ORMs). Um ORM permite que os desenvolvedores interajam com o banco de dados usando o paradigma de programação orientada a objetos, o que é mais intuitivo e seguro para muitos desenvolvedores.

O Prisma é um ORM moderno e poderoso que permite o acesso a bancos de dados através de uma interface simples e intuitiva. Ele oferece diversas funcionalidades, como migrações de banco de dados, controle de versão de esquema e consultas otimizadas, permitindo um acesso rápido e eficiente aos dados. O Prisma pode ser usado para acessar bancos de dados de forma eficiente e segura, com suporte para várias funcionalidades, como migrações automáticas, cliente de banco de dados com tipagem segura e navegador de banco de dados visual (PRISMA..., 2023).

2.9.3 Zod

Zod é uma biblioteca para TypeScript que ajuda a garantir que os dados em uma aplicação estejam corretos e seguros. Ele faz isso através do uso de "esquemas", que são como modelos que descrevem como os dados devem ser estruturados, incluindo o tipo de dados e regras. Por exemplo, um esquema pode especificar que um item em uma loja online deve ter um nome (caracteres), uma descrição (caracteres) e um preço (número). O trecho de Código 1 mostra como esse elemento pode ser representado através dessa biblioteca, criando um tipo de dado chamado "itemSchema".

Código Fonte 1 – Exemplo de uso da biblioteca Zod.

```
1 import z from 'zod';
2
3 export const itemSchema = z.object({
4   id: z.string().uuid(),
5   name: z.string(),
6   description: z.string().min(4),
7   price: z.number(),
8 });
```

Fonte: Criado pelo autor.

Com Zod, é possível definir esses esquemas e usar a biblioteca para verificar se os dados de entrada e saída da aplicação correspondem a esses esquemas. Isso é especialmente útil em aplicações web, onde os dados de entrada geralmente vêm de usuários e podem ser imprevisíveis.

Ao usar Zod para validar esses dados, é possível prevenir muitos erros e vulnerabilidades de segurança, tornando a aplicação mais robusta e confiável (ZOD..., 2023).

2.9.4 tRPC

O tRPC (*Typescript Remote Procedure Call*) é uma biblioteca leve que permite a criação de APIs totalmente seguras em termos de tipo (TRPC..., s.d.). Ele se apresenta como uma alternativa a outras tecnologias de chamada de procedimento remoto, como REST, GraphQL e gRPC.

Ele permite que os desenvolvedores criem APIs, sem a necessidade de manter manualmente as definições do formato dos dados entre o cliente e o servidor. Isso é conseguido através da geração automática de definições de tipo com base no código do servidor. Em outras palavras, o tRPC entende quais dados devem ser enviados na requisição, com base no código do servidor, eliminando a necessidade de manutenção manual dessas definições (TRPC..., s.d.). Complementando o que foi exibido anteriormente com o Zod, o Código 2 demonstra a implementação de uma rota backend que utiliza tRPC e Zod.

O Código 3, corresponde a uma implementação do tRPC no frontend, que utiliza esta rota para enviar informações.

Código Fonte 2 – Exemplo de uso da biblioteca tRPC no backend.

```
1
2 export const itemsRouter = createTRPCRouter({
3   create: publicProcedure
4     .input(
5       z.object({
6         name: z.string(),
7         description: z.string().min(4),
8         price: z.number(),
9       })
10    )
11   .mutation(({ ctx, input }) => {
12     return ctx.prisma.items.create({
13       data: {
14         description: input.description,
15         price: input.price,
16         name: input.name,
17       },
18     });
19   }),
20 });
```

Fonte: Criado pelo autor.

No contexto de desenvolvimento full stack, o tRPC se destaca como uma biblioteca fundamental para a construção tanto do servidor quanto do cliente. De acordo com o trabalho de (Nivasalo, 2022), o tRPC se mostrou uma excelente biblioteca para se basear, pois reduziu significativamente o tempo de desenvolvimento ao tornar extremamente fácil a implementação de chamadas de endpoint da API para o frontend.

Por fim, vale ressaltar que o tRPC é um projeto de código aberto e gratuito para uso, o que o torna uma excelente opção para desenvolvedores e empresas que buscam uma solução eficiente e econômica para a criação de APIs seguras em termos de tipo.

2.9.5 Fastify

Fastify é um *framework* web eficiente para Node.js, projetado para ser o mais rápido possível, tanto em termos de tempo de execução quanto de velocidade de desenvolvimento (FASTIFY: . . . , s.d.). Ele fornece um conjunto robusto de recursos para construir aplicações web e é totalmente extensível com seu sistema de *plugins*. Fastify também oferece um modelo de roteamento fácil de usar e suporte para manipulação de solicitações e respostas HTTP, tornando-o uma escolha popular para muitos desenvolvedores de Node.js.

2.9.6 Next.js

Next.js é um *framework* JavaScript para sistemas baseados em React, que permite a criação de páginas estáticas e dinâmicas, bem como a geração de conteúdo sob demanda, proporcionando uma experiência de carregamento mais rápida e eficiente para o usuário.

Código Fonte 3 – Exemplo de uso da biblioteca tRPC no frontend.

```
1 export const itemsRouter = createTRPCRouter({
2   // importa a rota da api trpc
3   const createItem = api.items.create.useMutation({
4     onSuccess: (createdItem) => {
5       toast.success("Item criado com sucesso", {
6         position: "top-right",
7         autoClose: 3000,
8         theme: "colored",
9       });
10    },
11    onError: (err) => {
12      toast.error("Erro ao criar item", {
13        position: "top-right",
14        autoClose: 5000,
15        theme: "colored",
16      });
17    }
18  });
19
20  // utilizacao dela para enviar a requisicao
21  createItem.mutate({
22    name: values.name,
23    description: values.description,
24    price: values.price,
25  });
```

Fonte: Criado pelo autor.

Este *framework* pode ser usado para criar aplicações web eficientes e escaláveis, com suporte para várias funcionalidades, como roteamento por arquivo, *streaming* de HTML dinâmico e suporte a CSS (NEXT... , 2023).

Amplamente utilizado e reconhecido na indústria de desenvolvimento web, o NextJs é o 14º maior projeto no GitHub e é considerado o *framework* ReactJS número 1, com mais de 100.000 estrelas no GitHub. Grandes empresas como Notion e Twitch utilizam o Next.js em suas aplicações, o que demonstra a confiabilidade e a maturidade do *framework* (Vercel, 2023).

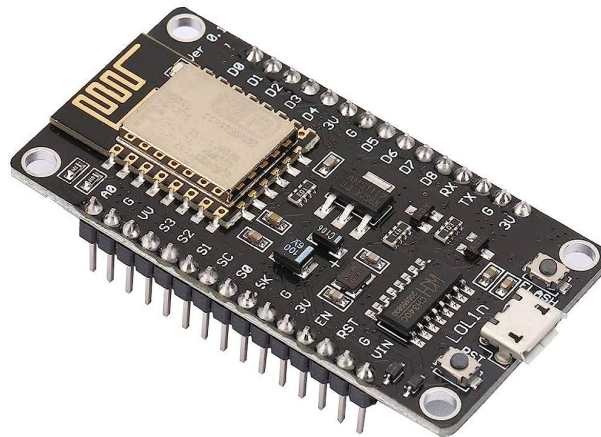
Além disso, o Next.js é recomendado na documentação oficial do React para projetos que se beneficiariam de suas características específicas, como renderização do lado do servidor e otimizações de desempenho. Isso indica a importância e a relevância do Next.js no ecossistema de desenvolvimento web atual. A adoção do Next.js neste trabalho é uma escolha estratégica que visa aproveitar essas vantagens e recursos poderosos para criar uma aplicação web robusta e eficiente.

2.10 NODEMCU ESP8266

O NodeMCU ESP8266 é um microcontrolador baseado na plataforma ESP8266, que possui conectividade Wi-Fi integrada. Ele é bastante utilizado em projetos de IoT

devido à sua facilidade de programação, baixo custo e recursos avançados. O ESP8266, demonstrado na Figura 6, é especialmente adequado para tais aplicações devido à sua capacidade de operar em baixa potência, o que é crucial para dispositivos alimentados por bateria, e sua capacidade de se conectar a redes Wi-Fi, permitindo a comunicação com a internet e outros dispositivos na rede (Kolban, 2016).

Figura 6 – ESP8266 NodeMcu.



Fonte: (NODE. . . , 2023).

2.10.1 C++ e Bibliotecas

O microcontrolador NodeMCU, pode ser programado em C++ e as bibliotecas relevantes para este projeto estão listadas a seguir.

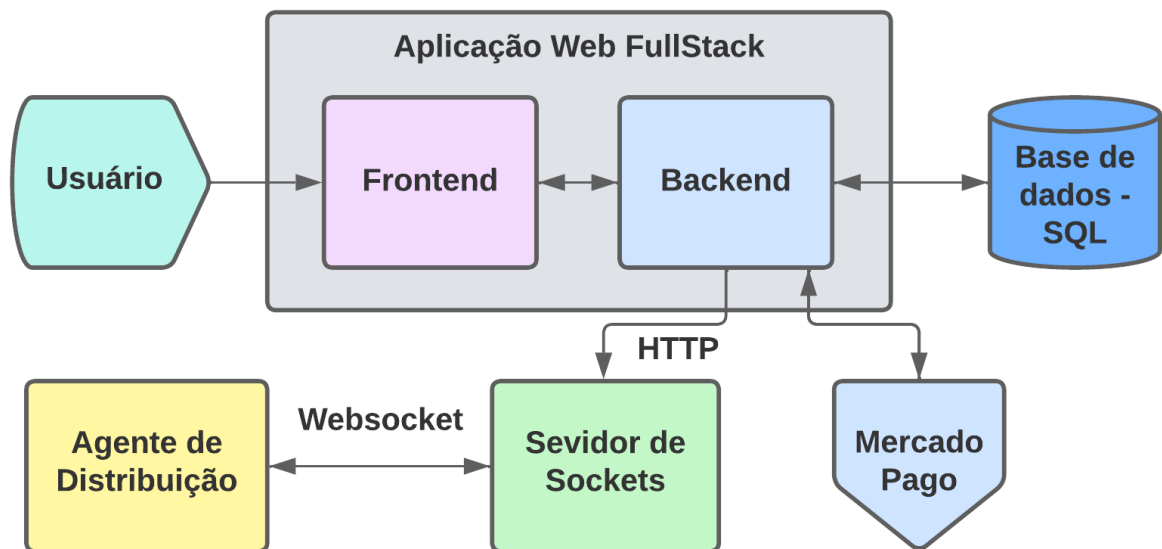
- **ESP8266WiFi.h:** Esta biblioteca permite que o NodeMCU se conecte a redes Wi-Fi. Ela fornece funções para conectar, desconectar e verificar o status da conexão Wi-Fi. No código fornecido, essa biblioteca é usada para conectar o NodeMCU à rede Wi-Fi especificada pelas constantes “ssid” e “password”.
- **WebSocketsClient.h:** Esta biblioteca permite que o NodeMCU se comunique com servidores *websocket*. *Websocket* é um protocolo que permite comunicação bidirecional em tempo real entre clientes e servidores. No código fornecido, essa biblioteca é usada para conectar o NodeMCU a um servidor *websocket* e enviar/receber mensagens.

3 ARQUITETURA DO SISTEMA

Este capítulo delinea a arquitetura do sistema proposto, fornecendo justificativas para as decisões arquitetônicas tomadas. A proposta em questão visa desenvolver uma alternativa inovadora para a automação de pagamentos, incorporando tecnologias contemporâneas e princípios da Web 3.0, como *tokens* e criptografia de dados. Elementos como a computação em nuvem, a Internet das Coisas (IoT) e o uso de *tokens* únicos, como JWT e UUID, são integrados com o objetivo de estabelecer uma infraestrutura eficaz e segura.

O sistema é composto por três componentes principais: uma aplicação *web fullstack*, um servidor de *sockets* e um agente de distribuição. Esses componentes interagem entre si para fornecer a funcionalidade desejada. A Figura 7 ilustra a arquitetura do sistema.

Figura 7 – Arquitetura do Sistema.



Fonte: Criado pelo autor.

Cada componente tem um papel específico no sistema e trabalha em conjunto com os outros para fornecer a funcionalidade desejada. A arquitetura foi projetada para ser escalável e eficiente, e alcança através de múltiplos meios. A implementação de *sockets* em um servidor separado, evita a necessidade de *pooling*, reduzindo o número de requisições ao servidor e a separação de componentes, como o banco de dados e servidor de *sockets*, permite que sejam escalados separadamente para atender as demandas.

As seções 3.1 a 3.5 detalham cada um dos elementos da arquitetura.

3.1 APLICAÇÃO WEB FULLSTACK

Este elemento é o centro do sistema e coordena todos os demais. Está dividido entre *frontend* e *backend* para melhor representar suas funcionalidades, porém é representado

de forma única pois é dessa forma que se dá sua implementação e desenvolvimento.

1. **Frontend:** Esta é a interface do sistema, que é executada no dispositivo do usuário, onde o mesmo pode interagir com as lojas online cadastradas no sistema. Além disso, permite a criação, edição, visualização e exclusão de contas de administradores, lojas, itens e pedidos. É através desta interface que todas as interações do usuário com o sistema ocorrem. Estas ações serão efetuadas através de requisições para o *backend* que é executado no servidor.
2. **Backend:** Esta é a API que processa as solicitações vindas do *frontend*. É neste elemento que a conexão com o banco de dados é estabelecida para recuperar e armazenar informações. Além disso, o *backend* é responsável por se comunicar com a API do Mercado Pago para registrar pedidos e receber os links que permitem aos usuários efetuar pagamentos. Este componente também se conecta com a API do sistema responsável pelo gerenciamento dos *websockets*, garantindo que as notificações sejam enviadas sempre que o status de um pedido for atualizado.

3.2 SERVIDOR DE SOCKETS

A decisão de utilizar *sockets* foi tomada para permitir que os Agentes de Distribuição recebam informações de maneira contínua. A adoção dessa tecnologia resulta em uma redução significativa no número de requisições ao servidor, especialmente quando comparada à comunicação via múltiplas requisições HTTP feitas por diversos dispositivos em intervalos curtos de tempo.

A separação deste componente da aplicação *fullstack* é necessária devido às características das conexões *socket*. Este tipo de conexão mantém uma comunicação contínua, o que gera uma complexidade maior tanto no desenvolvimento quanto na implantação, se comparada a uma API do tipo REST. Isso ocorre pois, para implementar *sockets*, o servidor deve manter as conexões ativas indeterminadamente, algo que não é garantido por todos os provedores de hospedagem em nuvem e também não é indicado para uso na API da aplicação *fullstack*, com NextJS. Este tipo de API não foi projetada para manter conexões ativas por longos períodos de tempo, e por mais que possam estabelecer conexões *sockets*, isto não é uma prática recomendada em sua documentação e pelos criadores. Por esses motivos, optou-se por criar este elemento separadamente.

Este componente da arquitetura é um servidor independente, que hospeda uma API capaz de receber comunicações HTTP e *websocket*. Esta API estabelece conexões *socket* com os microcontroladores e as mantém ativas enquanto houver algum dispositivo conectado. Para garantir a segurança, as conexões devem ser autenticadas com uma chave UUID incluída no cabeçalho de cada requisição. Além disso, cada *socket* possui um ID único, permitindo a distinção entre diferentes pontos de venda.

3.3 AGENTE DE DISTRIBUIÇÃO

Os agentes de distribuição são dispositivos que estabelecem conexão com o servidor de *sockets* para receber informações e responder aos pedidos. No contexto deste trabalho, a ênfase será dada à conexão e ao recebimento de dados provenientes da API.

A implementação destes agentes de distribuição, capazes de entregar um produto ou realizar alguma tarefa paga através do sistema pode variar significativamente e não será o foco deste trabalho. Estes dispositivos incluem mas não se limitam a: aplicativos, microcontroladores, dispensers, entregadores ou uma tela que apresente os itens pedidos a uma equipe de funcionários. Portanto, a principal preocupação aqui é garantir uma comunicação eficaz e segura entre um agente de distribuição e o servidor de sockets.

3.4 MERCADO PAGO

Esta é a aplicação externa utilizada para efetivar os pagamentos, tem *frontend* e *backend*. O *frontend* permite ao usuário fornecer os dados de pagamento e o *backend* recebe os pedidos e envia notificações. Poderia ser substituída por outra API similar, porém seriam necessárias alterações significativas na estrutura do projeto, visto que APIs diferentes utilizam métodos de autenticação e estruturas de dados diferentes.

Quando um pedido é gerado no *frontend* da aplicação *web*, uma requisição é feita para a API do Mercado Pago contendo informações deste pedido e um token do sistema. Ao receber estas informações do pedido, a API retorna um link contendo o endereço de uma página onde o usuário pode realizar o pagamento. Assim que o pagamento é confirmado internamente pelo Mercado Pago, a API envia uma notificação para o *backend* da aplicação *web* deste sistema, informando que houve uma atualização no pedido, e a aplicação *web* faz uma nova requisição para a API do Mercado Pago, para obter o status do pedido atualizado.

3.5 BASE DE DADOS

É o banco de dados da aplicação, deve ser capaz de armazenar itens, pedidos, usuários, pontos de vendas e quaisquer outros elementos necessários ao sistema.

4 IMPLEMENTAÇÃO

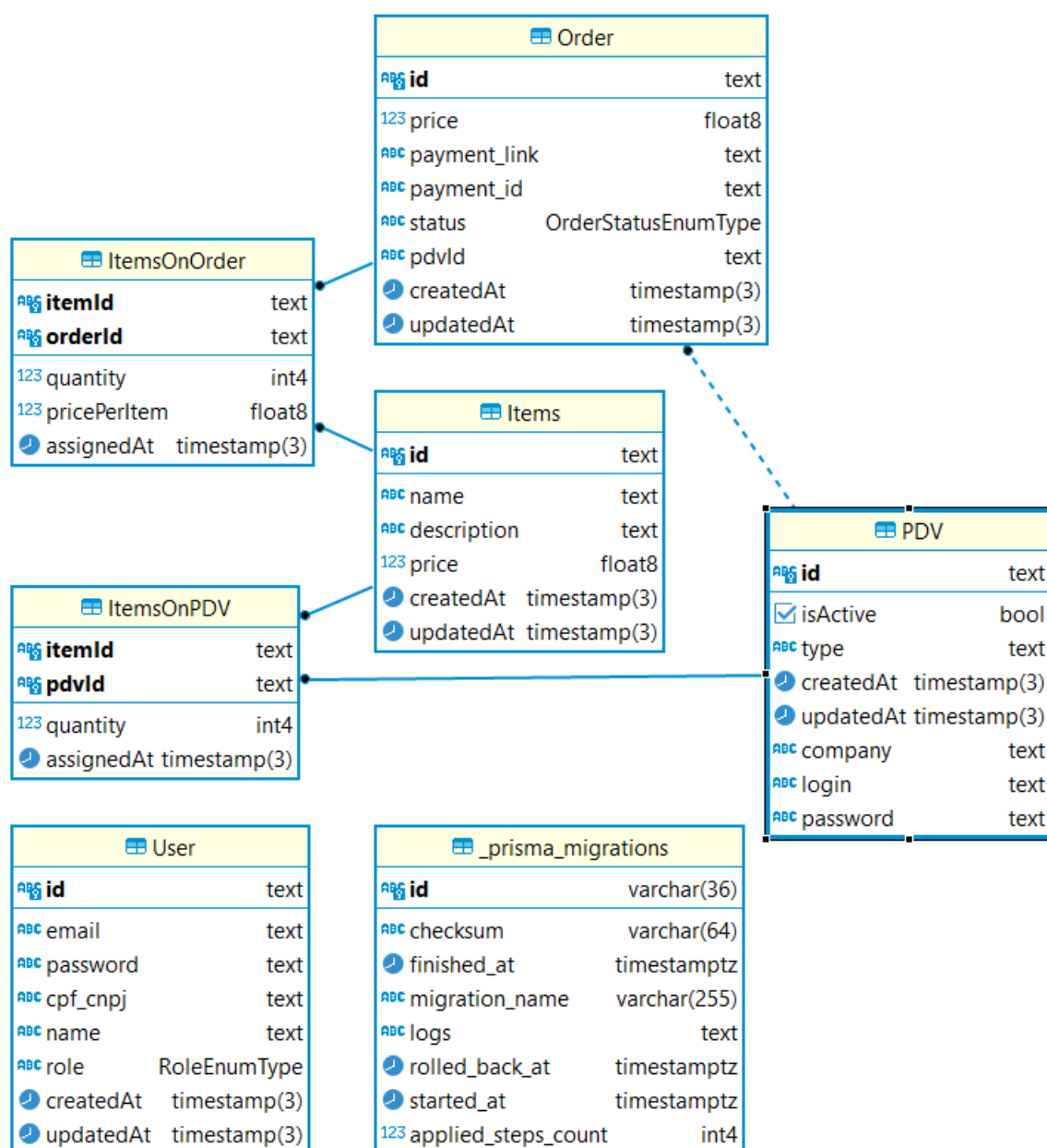
Este capítulo descreve o desenvolvimento do projeto, que envolveu a aplicação dos conceitos apresentados na fase de fundamentação teórica e a aplicação da arquitetura definida. O capítulo é dividido em quatro seções: a Seção 4.1 fala sobre o banco de dados, a Seção 4.2 apresenta a aplicação *web full-stack*, a Seção 4.3 traz a definição o servidor de *sockets* e a Seção 4.4 aborda o agente de distribuição.

4.1 BANCO DE DADOS

A primeira etapa do processo de desenvolvimento foi a modelagem do banco de dados relacional. O modelo definido é composto por quatro entidades fundamentais: usuários, pontos de venda (PDVs), pedidos e itens. Cada entidade possui seus atributos específicos e suas relações com as outras entidades. O ERD (*Entity-Relationship Diagram*) do banco de dados, obtido através da ferramenta DBeaver está representado pela Figura 8. A seguir são descritas cada uma das entidades presentes na Figura 8:

- *User* (Usuário): Este modelo representa um usuário do sistema. Os usuários têm uma identificação única (id), e-mail, senha, cpf/cnpj, nome, cargo (role), além dos campos de controle *createdAt* e *updatedAt*.
- *RoleEnumType*: Este é um tipo de enumeração que define os possíveis papéis que um usuário pode ter: *user* (usuário) ou *admin* (administrador).
- *PDV* (Ponto de Venda): Representa um ponto de venda dentro do sistema. Cada PDV tem uma identificação única, estado (ativo ou não), tipo, empresa (company), login, senha e duas listas de itens e pedidos relacionados a ele.
- *Order* (Pedido): Representa um pedido feito no sistema. Cada pedido tem uma identificação única, uma lista de itens (*ItemsOnOrder*), preço, link de pagamento, identificação de pagamento, status, identificação do ponto de venda onde foi feito e os campos de controle *createdAt* e *updatedAt*.
- *OrderStatusEnumType*: Este é um tipo de enumeração que define os possíveis estados que um pedido pode ter: *pending* (pendente), *approved* (aprovado), *accredited* (creditado), *delivered* (entregue), *canceled* (cancelado).
- *ItemsOnOrder* (Itens no Pedido): Representa a relação entre pedidos e itens. Ele inclui a quantidade de cada item no pedido, o preço por item, a identificação do item, a identificação do pedido e a data em que o item foi atribuído ao pedido.
- *ItemsOnPDV* (Itens no Ponto de Venda): Representa a relação entre os pontos de venda e os itens. Ele inclui a quantidade de cada item no ponto de venda, a identificação do item, a identificação do ponto de venda e a data em que o item foi atribuído ao ponto de venda.

Figura 8 – ERD - Projeto.



Fonte: Criado pelo autor.

- Items (Itens): Representa um item que pode ser vendido em um ponto de venda e incluído em um pedido. Cada item tem uma identificação única, nome, descrição, preço e listas de pedidos e pontos de venda aos quais está associado.
- Prisma Migrations: Este modelo é utilizado pelo ORM Prisma para controlar as alterações na estrutura do banco, controlando quais migrations foram aplicadas no banco de dados, permitindo um versionamento deste.

Em termos de relacionamentos, os usuários (User) não estão diretamente associados a outras entidades. Os pedidos (Order) estão associados a um ponto de venda (PDV) numa relação muitos pra um, através da coluna pdvId na tabela Order, e de muitos para muitos com itens (Items), através da tabela ItemsOnOrder. Já os pontos de venda (PDV) tem uma associação de um para muitos com itens através da entidade ItemsOnPDV.

Optou-se pela utilização do PostgreSQL como banco de dados, e foram incluídas as entidades citadas anteriormente, seguindo a sintaxe do Prisma de forma a criar e se comunicar com o banco de dados. O Código 4 exibe a declaração da tabela User, usando a sintaxe do Prisma.

Código Fonte 4 – Exemplo de um schema no Prisma.

```
1 generator client {
2   provider = "prisma-client-js"
3 }
4
5 datasource db {
6   provider = "postgresql"
7   url      = env("DATABASE_URL")
8 }
9
10 model User {
11   id          String          @id @unique @default(uuid())
12   email       String
13   password    String
14   cpf_cnpj    String          @unique
15   name        String
16   role        RoleEnumType?   @default(user)
17   createdAt   DateTime        @default(now())
18   updatedAt   DateTime        @updatedAt
19 }
20
21 enum RoleEnumType {
22   user
23   admin
24 }
```

Fonte: Criado pelo autor.

4.2 APLICAÇÃO WEB FULL STACK

A aplicação web foi desenvolvida utilizando o *framework* Next.js, que permite a criação de aplicações fullstack com React. A IDE escolhida foi o Visual Studio Code (VS Code). Como o Next.js é um *framework full-stack*, tanto o *front-end* quanto o *back-end* foram desenvolvidos dentro de um projeto NextJS. Pode-se destacar algumas bibliotecas como o tRPC para criação das rotas da API no *backend* e acesso a elas no *frontend*, material UI que é uma biblioteca de componentes React de código aberto, Zod para verificação de dados e Prisma como ORM para gerenciar o banco de dados.

A aplicação possui três pontos de entrada para os usuários: a página para compras, a página de gestão de um ponto de vendas, e a página de gestão do sistema. A página de compras é pública, já as páginas de gestão de um ponto de vendas e gestão do sistema contam com autenticação, onde o usuário deve ter um login e senha autorizados. Cada ponto de vendas tem um acesso, conforme estabelecido no momento em que foi cadastrado, já para o acesso de gestão do sistema, múltiplos acessos podem ser cadastrados.

4.2.1 Fluxo de Compras

A Figura 9 mostra o fluxograma de compra para um cliente. Ao acessar a página de compras, o sistema segue um layout tradicional, listando todos os produtos em uma lista e exibindo para o usuário o preço total, como exibido na Figura 10. Quando o usuário finaliza a compra, é redirecionado para o pagamento através do Mercado Pago. A API do Mercado Pago foi escolhida por permitir o pagamento via PIX e por disponibilizar uma interface muito completa e confiável para o usuário, sendo uma marca conhecida no Brasil e que traz diversas garantias e verificações para o processo de pagamento.

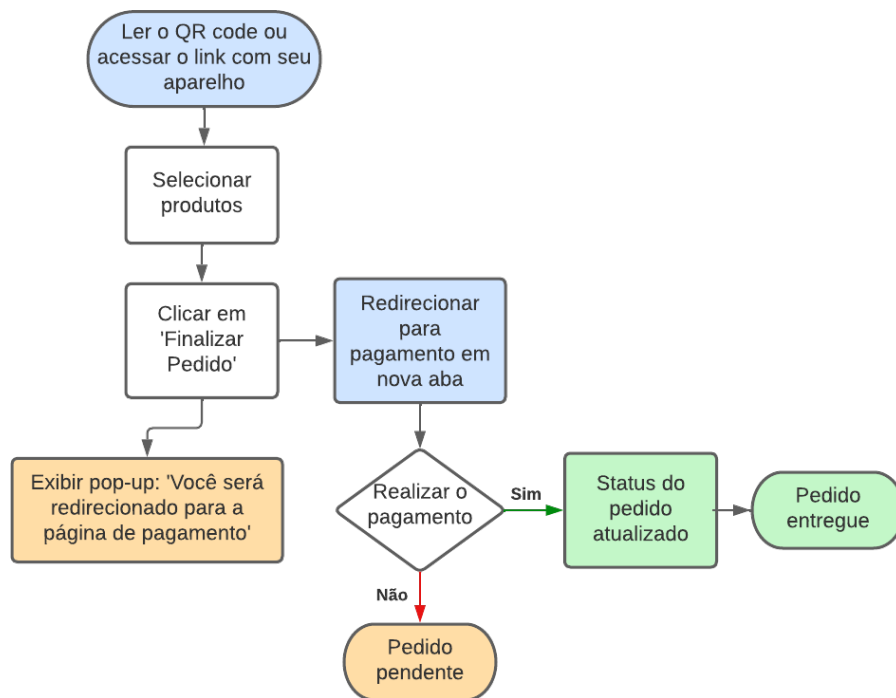
Assim que o pagamento é efetuado, uma notificação é enviada para a API da aplicação Web, e a aplicação realiza uma requisição ao servidor do Mercado Pago para obter confirmação do pagamento.

Ao ser confirmado o pagamento, a aplicação Web envia uma requisição HTTP Post para a API que controla os Websockets, de forma a postar uma mensagem na conexão do ponto de vendas em que ocorreu a compra.

4.2.2 Fluxo de Administração do Sistema

A administração do sistema, apenas pode ser feita por usuários autorizados através da página de *login*. Um administrador pode visualizar, criar, editar e excluir pontos de vendas e usuários administradores. Além disso, contas de administrador têm acesso à página “log de pagamentos”, que exibe as últimas notificações recebidas da API do Mercado Pago. A Figura 11 representa o diagrama de administração do sistema.

Figura 9 – Fluxograma de compra.



Fonte: Criado pelo autor.

4.2.3 Fluxo de Administração de Um Ponto de Vendas

O controle de um ponto de vendas apenas pode ser acessado através da página de *login* para controle de loja. A Figura 12 mostra um fluxograma com as opções para controle de um ponto de vendas. Através desse acesso o usuário pode visualizar, criar, editar e excluir itens de um ponto de venda, gerenciar e visualizar pedidos, e também pode acessar a página da loja ligada ao ponto de vendas, para conferir como está a interface onde os clientes realizam os pedidos.

4.2.4 *Frontend*

O *frontend* foi desenvolvido utilizando React, uma biblioteca JavaScript para construção de interfaces de usuário. A interface foi projetada para ser intuitiva e simples. Ela utiliza no *layout* componentes da biblioteca Material UI, uma biblioteca de componentes que implementa o Material Design do Google, e componentes criados com HTML e CSS. Foram criadas páginas para a listagem, criação, edição e exclusão de usuários e pontos de venda. A Figura 13 mostra a tela do painel do administrador que permite o controle dos pontos de vendas, e representa o layout que foi seguido ao longo de todo o frontend para listar dados.

Figura 10 – Página da loja.

Loja1

Cafe

teste

Preço: R\$ 1.00

Quantidade:

ADICIONAR AO PEDIDO

Chocolate

Doce

Preço: R\$ 2.00

Quantidade:

ADICIONAR AO PEDIDO

Pedido

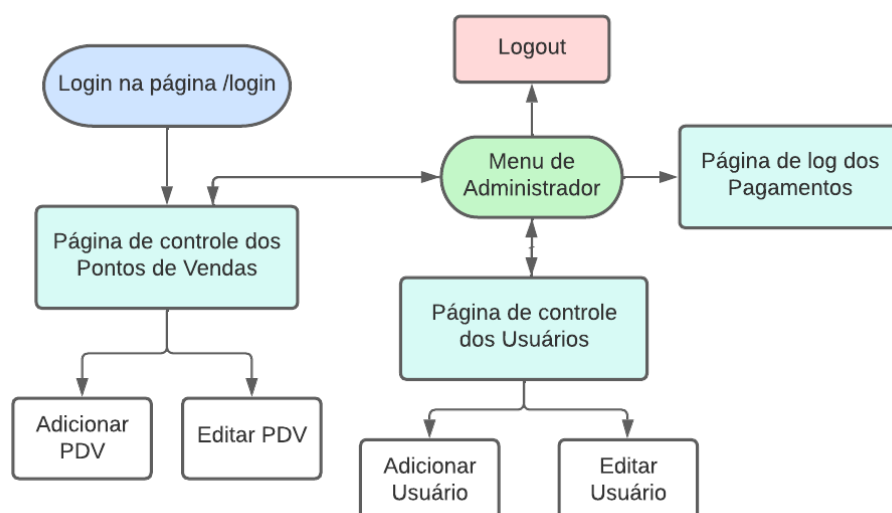
Items: Cafe (x1), Chocolate (x2)

Preço Total: \$5.00

FINALIZAR COMPRA

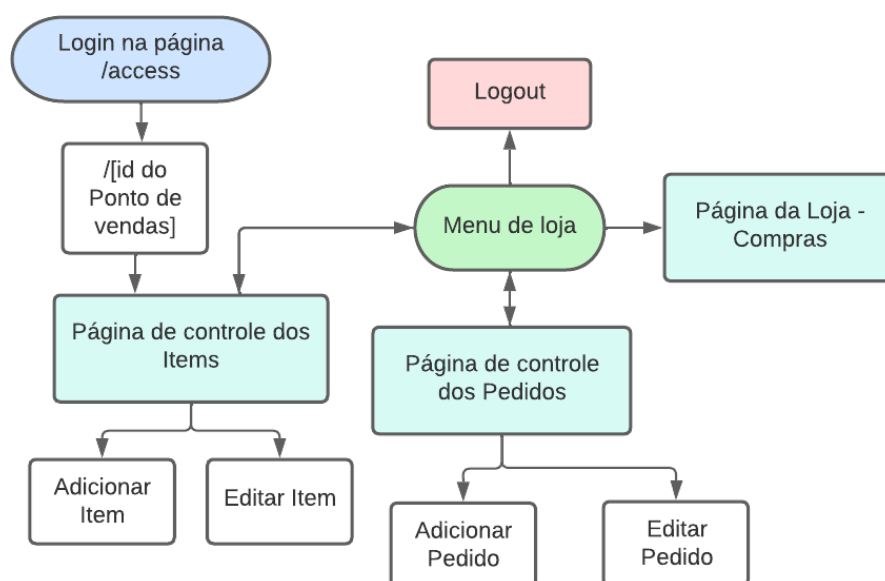
Fonte: Criado pelo autor.

Figura 11 – Fluxograma administrativo.



Fonte: Criado pelo autor.

Figura 12 – Fluxograma do controle de um ponto de vendas.



Fonte: Criado pelo autor.

Figura 13 – Tela de controle de PDVs.

PDVS USUÁRIOS LOG DE PAGAMENTOS				
Pontos de Vendas				
ADICIONAR				
Editar	Companhia	Login	Tipo	Ativo
	Klaus Dev.	klausdk1999@gmail.com	manual	Sim
	User1	user1@email.com	manual	Sim
	Loja1	admin@email.com	automated	Sim
Linhas por página 5 1-3 of 3 < >				

Fonte: Criado pelo autor.

Figura 14 – Tela para adicionar Item.



The screenshot shows a web application interface with a dark green header bar. On the left of the header is a logo consisting of three white triangles. In the center of the header are the navigation links 'PEDIDOS', 'ITEMS', and 'LOJA'. On the right is a power icon. Below the header, the main content area has a light green background. At the top of this area is the title 'Adicionar item'. Below the title is a white form box. Inside the form box, there are four input fields: 'Nome do item' with the value 'Café', 'Descrição' with the value 'Des' and a red border and error message 'Digite pelo menos 4 caracteres', 'Preço' with the value '2', and 'Quantidade' with the value '4'. At the bottom right of the form box are two buttons: 'VOLTAR' (black with white text) and 'SALVAR' (green with white text).

Fonte: Criado pelo autor.

Além do *layout* anterior, usado para listagem, outro *layout* que aparece em vários locais do sistema destina-se a coleta de informações, representado na Figura 14, usado para adicionar e editar informações.

4.2.5 Backend

A API *backend* dentro da aplicação *full-stack* foi dividida em grupos de rotas, acordo com as entidades do banco de dados envolvidas. Existem grupos de rotas para autenticação, itens, pedidos, pontos de vendas e usuários. Um exemplo de rota pode ser visto no Código 4. Trata-se da rota de *login* que exige como entrada um objeto com os campos “email” e “password”. Se esses parâmetros forem fornecidos corretamente, o processo descrito como *mutation* é iniciado. Uma *mutation* se refere a qualquer procedimento que altera o estado dos dados. Nesse caso, a ação desejada é realizada e a senha criptografada no banco é comparada com a senha recebida na rota. O Código 5 mostra como a rota de login foi implementada e parte do processo de autenticação.

O tRPC serve como um gerenciador destas rotas. Diferente de uma API mais comum, onde o desenvolvedor especifica o caminho de cada rota, e precisa criar verificações para os tipos e respostas, o tRPC, em conjunto com o Zod, fornecem esta estrutura, cabendo ao desenvolver apenas definir quais dados devem ser recebidos, sem a necessidade de criar os meios de verificação.

As rotas implementadas, além de serem utilizadas pelas páginas do frontend para enviar e receber dados, também fazem as conexões externas, como confirmação de pagamentos com a API do Mercado Pago e envio de mensagens para a API de Websockets.

Código Fonte 5 – Exemplo de uma rota tRPC.

```
1 export const authRouter = createTRPCRouter({
2   login: publicProcedure
3     .input(
4       z.object({
5         email: z.string(),
6         password: z.string().min(4),
7       })
8     )
9     .mutation(async ({ input, ctx }) => {
10      const { res } = ctx;
11      const { email, password } = input;
12
13      const databaseUser = await ctx.prisma.user.findFirst({
14        where: { email: email },
15      });
16      if (!databaseUser) {
17        throw new TRPCError({
18          code: "NOT_FOUND",
19          message: "Usuario nao encontrado",
20        });
21      }
22
23      const isValidPassword = bcrypt.compareSync(
24        password,
25        databaseUser.password
26      );
```

Fonte: Criado pelo autor.

O desenvolvimento da aplicação web resultou em um sistema funcional que atende aos requisitos estabelecidos na seção de objetivos deste trabalho. Foram criadas telas para o cadastro de usuários, pontos de venda e pedidos, além de telas para a visualização e edição dessas informações. O sistema conta com autenticação de usuários e utilização de *tokens* JWT para garantir a segurança das informações.

A interface do sistema foi desenvolvida seguindo boas práticas de usabilidade e design, resultando em uma interface intuitiva e fácil de usar para os usuários finais.

4.3 SERVIDOR DE *SOCKETS*

Este elemento do sistema é um servidor *websocket* construído com a biblioteca Fastify para Node.js, com autenticação por chave de acesso e suporte a conexões CORS (*Cross-origin Resource Sharing*)¹. Ele é encarregado da comunicação entre o servidor e os agentes de distribuição, permitindo que as atualizações sejam enviadas assim que ocorrem, sem a necessidade de solicitações constantes. Este método de comunicação foi escolhido

¹ CORS é um mecanismo que permite que muitos recursos (por exemplo, fontes, JavaScript, etc.) em uma página da web sejam solicitados de outro domínio fora do domínio da qual a origem da solicitação foi feita.

por permitir uma melhor escalabilidade do sistema, reduzindo o número de requisições que precisam ser processadas pelo servidor.

Ao final do desenvolvimento, é possível criar conexões do tipo *socket* privadas, e enviar mensagens para os *sockets* via requisições HTTP, atendendo as necessidades de comunicação do sistema entre a API e os agentes de distribuição. A seguir será detalhado o funcionamento do servidor.

4.3.1 Configurações iniciais e Criação do servidor *websocket*

As variáveis de ambiente são carregadas utilizando a biblioteca `dotenv` e instância do Fastify é criada, e configurada para permitir requisições de qualquer origem e para os métodos GET e POST.

Uma instância do servidor *websocket* é criada usando a biblioteca `ws`. O servidor *websocket* é configurado para não criar um servidor HTTP próprio, pois o servidor HTTP será fornecido pelo Fastify.

4.3.2 Gerenciamento e Atualização de conexões *websocket*

Uma lista para armazenar as conexões *websocket* é criada. Cada conexão é identificada por uma *string*, que é obtida do URL da requisição que abriu a conexão. O servidor *websocket* é configurado para lidar com eventos de conexão, mensagem e fechamento. Quando uma nova conexão é estabelecida, o servidor registra a conexão em uma lista de conexões. Quando uma mensagem é recebida, o servidor a envia para a conexão a qual se destina. Quando uma conexão é fechada, o servidor remove a conexão do mapa.

O servidor Fastify é configurado para lidar com eventos de upgrade de conexões HTTP para *websocket*. Quando uma requisição de upgrade é recebida, o servidor verifica se a chave de acesso fornecida nos cabeçalhos da requisição é válida. Se a chave de acesso for válida, a requisição de upgrade é passada para o servidor WebSocket. Caso contrário, a conexão é encerrada.

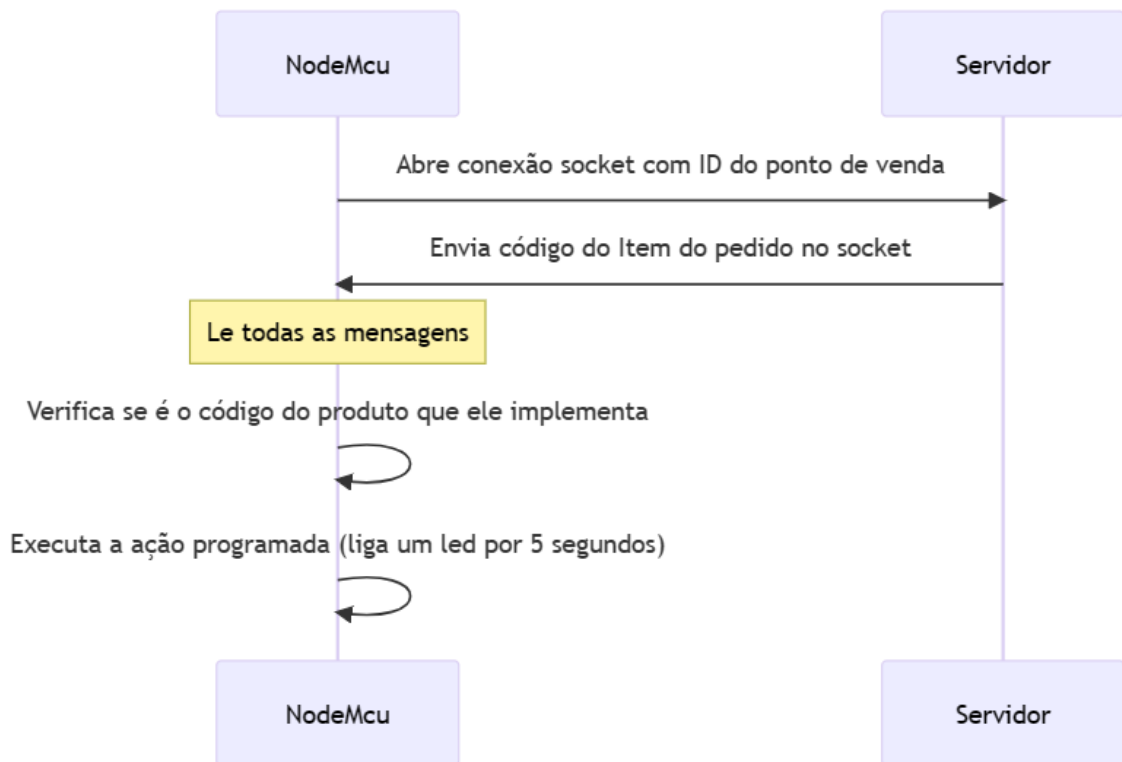
4.3.3 Roteamento HTTP e Socket

Duas rotas HTTP são definidas:

- **GET `"/status"`** - retorna uma mensagem indicando que o servidor está online.
- **POST `"/post"`** - é usada para enviar mensagens através das conexões WebSocket. Antes de tratar a requisição, a rota verifica se a chave de acesso fornecida nos cabeçalhos da requisição é válida. Se a chave de acesso for válida, a requisição é processada. A requisição deve incluir no corpo um ID de conexão e uma mensagem. A mensagem é enviada através da conexão socket correspondente ao ID fornecido.

Para conexão via WebSocket:

Figura 15 – Diagrama da conexão NodeMCU e servidor.



Fonte: Criado pelo autor.

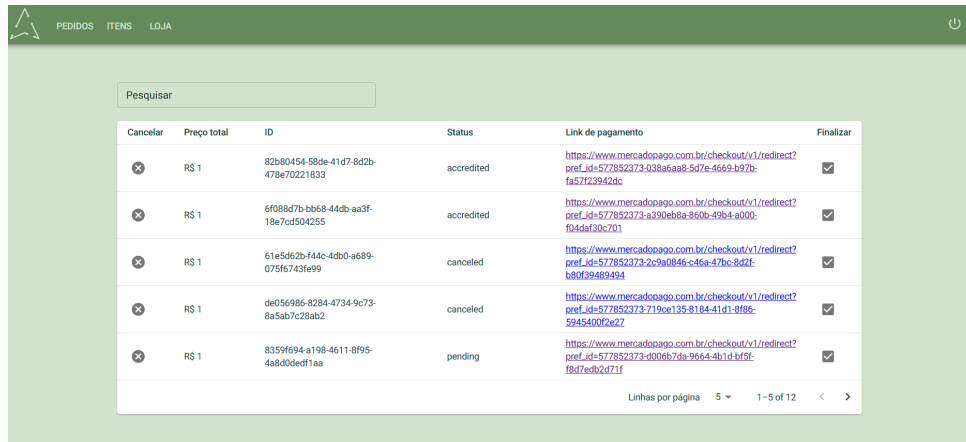
- **ws://{endereço do servidor}/{ID do socket}** - conecta no socket com o ID passado como parâmetro. Similar a rota HTTP /post, antes de tratar a conexão, a rota verifica se a chave de acesso fornecida nos cabeçalhos da requisição é válida.

4.4 AGENTE DE DISTRIBUIÇÃO

O desenvolvimento de um agente de distribuição se deu através de um microcontrolador NodeMCU ESP8266 e da plataforma Arduino IDE. O código do controlador desenvolvido em C++, inclui as bibliotecas ESP8266WiFi.h, que permite que o dispositivo se conecte com qualquer rede wifi, e WebSocketsClient.h que permite a conexão com *websocket*. O microcontrolador se conecta no *websocket*, com o UUID do ponto de vendas, e verifica cada mensagem recebida. Neste projeto a ênfase será na conexão e recebimento de dados do servidos de *socket*, e o microcontrolador acende um led para demonstrar que recebeu a notificação. A Figura 15 apresenta o funcionamento do microcontrolador.

Para implementar um agente de distribuição, este deverá verificar pelos códigos de itens presentes no ponto de venda, e implementar o que for necessário para fazer a entrega de cada item.

Figura 16 – Tela de pedidos.



Cancelar	Preço total	ID	Status	Link de pagamento	Finalizar
	R\$ 1	82b80454-58de-41d7-8d2b-478e70221833	accredited	https://www.mercadopago.com.br/checkout/v1/redirect?pref_id=577852373-038a6aa8-5d7e-4669-b97b-fa5723942dc	<input checked="" type="checkbox"/>
	R\$ 1	6f088d7b-bb68-44db-aa3f-18e7cd504255	accredited	https://www.mercadopago.com.br/checkout/v1/redirect?pref_id=577852373-a390eb8a-860b-49b4-a000-f04daf30c7d1	<input checked="" type="checkbox"/>
	R\$ 1	61e5d62b-f44c-4db0-a689-075f6743fe99	canceled	https://www.mercadopago.com.br/checkout/v1/redirect?pref_id=577852373-2c9a0846-c45a-47bc-8d2f-b80739489494	<input checked="" type="checkbox"/>
	R\$ 1	de056986-8284-4734-9c73-8a5ab7c28ab2	canceled	https://www.mercadopago.com.br/checkout/v1/redirect?pref_id=577852373-719ce135-8184-41d1-8f86-5945400f2e27	<input checked="" type="checkbox"/>
	R\$ 1	8359f694-a198-4611-8f95-4a8d0dedf1aa	pending	https://www.mercadopago.com.br/checkout/v1/redirect?pref_id=577852373-d006b7da-9664-4b1d-bf5f-f8d7edbd2d71f	<input checked="" type="checkbox"/>

Linhas por página 5 1-5 of 12

Fonte: Criado pelo autor.

O desenvolvimento do agente de distribuição no NodeMCU resultou em um dispositivo funcional capaz de se comunicar com a aplicação web e receber dados sobre o status dos pedidos. Foram realizados testes que provaram a integração do NodeMCU com a aplicação web e validaram o recebimento de dados sobre o status dos pedidos. Eles serão detalhados na Seção 5.2.

Uma alternativa para este agente de distribuição automático, é a utilização da interface para controle de pedidos do ponto de vendas, exibida na Figura 16. Este método é chamado de “manual”, e pode ser selecionado no momento de criação do ponto de vendas no sistema. Pontos de vendas manuais não precisam de um agente de distribuição e os pedidos devem ser atualizados manualmente. Eles não utilizam conexões no servidor de *socket* e devem ser controlador inteiramente pelo usuário através da interface. O método apresentado anteriormente, com um agente de distribuição conectado via *socket*, é definido como “automated” no momento da criação do ponto de vendas e é considerado o padrão.

5 RESULTADOS

Este capítulo apresenta os resultados alcançados com a execução e implementação do projeto. O capítulo é dividido em 3 seções: a Seção 5.1 trata dos testes de funcionamento, a Seção 5.2 trata da hospedagem e a Seção 5.3 trata das considerações finais.

5.1 RESULTADOS E TESTES DA IMPLANTAÇÃO EM AMBIENTE DE PRODUÇÃO

Para testar o sistema e demonstrar na prática os resultados obtidos, será simulado o processo de compra de um café em um ponto de vendas programado para o teste. Os passos simulados são os seguintes:

- Para fazer uma compra, o cliente deve acessar a página loja do ponto de venda em que deseja fazer a compra. A Figura 17 mostra um QR code que redireciona para a loja de testes do sistema. O mesmo poderia estar presente em um ponto de vendas físico, ou o link¹ poderia ser fornecido ao cliente diretamente para vendas via chat virtual.
- Na loja *online*, o cliente seleciona o produto que deseja e clica em “finalizar compra”. A Figura 18 mostra a loja de testes. Ao clicar em finalizar compra uma mensagem é exibida, indicando ao usuário a página de pagamentos, Figura 19.
- Caso o dispositivo do usuário permita, ele será redirecionado a página de pagamento. Caso contrário, ele pode pagar usando o link presente na mensagem. A página de pagamento é demonstrada na Figura 20, nela o usuário pode selecio-

¹ <https://quickpay.vercel.app/store/00f138a0-e8ac-4497-8eb9-057fcf2dec0d>

Figura 17 – QR code para a loja de testes.



Fonte: Criado pelo autor.

Figura 18 – Loja de testes.

Loja de Testes

Cafe

O café espresso tradicional, sem leite

Preço: R\$ 1.00

Quantidade:

Chocolate

Doce

Preço: R\$ 2.00

Quantidade:

Pedido

Items: Cafe (x1)

Preço Total: \$1.00

FINALIZAR COMPRA

Fonte: Criado pelo autor.

Figura 19 – Mensagem ao finalizar compra.

Loja de Testes

Cafe

O café espresso tradicional, sem leite

Preço: R\$ 1.00

Quantidade:

Chocolate

Doce

Preço: R\$ 2.00

Quantidade:

Pedido

Items:

Preço Total: \$0.00

Compra finalizada com sucesso!

Atenção

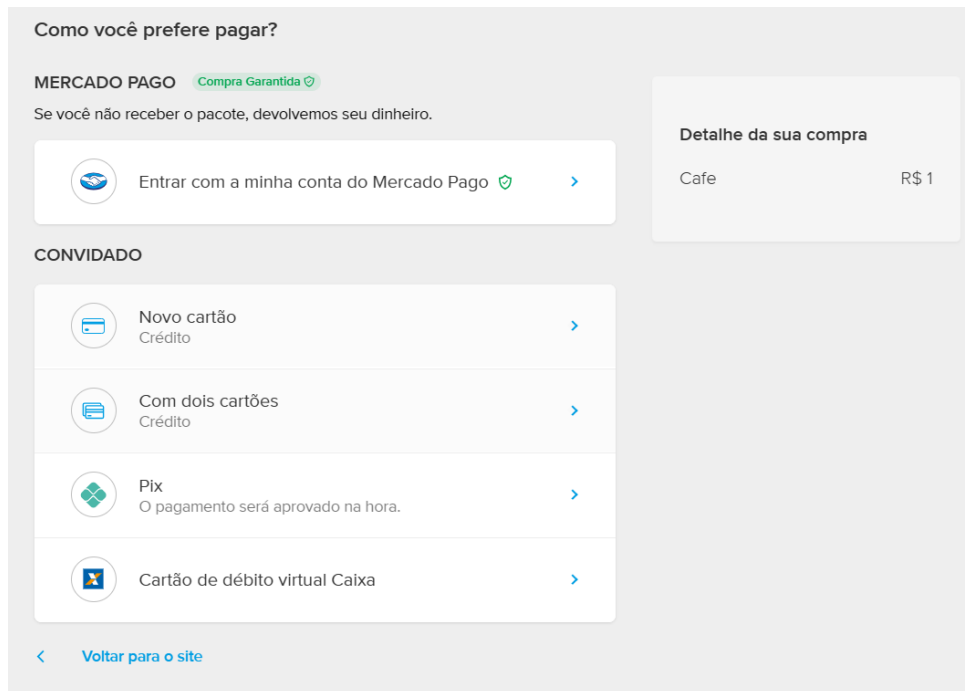
Você será redirecionado para a página de pagamento.

Link

https://www.mercadopago.com.br/checkout/v1/redirect?pref_id=577852373-134342ee-7128-49a6-a48b-5704a42a394a

Fonte: Criado pelo autor.

Figura 20 – Tela de pagamento.



Fonte: Criado pelo autor.

nar o método de pagamento de sua preferência. O processo é garantido através da integração com o Mercado Pago.

- Quando o pagamento é recebido e confirmado pela API, o status do pedido é atualizado no sistema, ficando visível na interface. Caso seja um ponto de vendas do tipo automático, uma mensagem com o id do item adquirido é enviada no *socket* do ponto de vendas.
- Caso o ponto de vendas tenha um agente de distribuição conectado, o pedido será entregue por este de forma automática. Caso o ponto de vendas seja do tipo manual, um responsável pelo ponto de vendas deve acompanhar os pedidos na interface do sistema, e realizar as entregas. Neste exemplo, o agente de distribuição é um NodeMCU que está programado para acender um led, de forma a demonstrar o recebimento do pedido, como exibido na Figura 21.

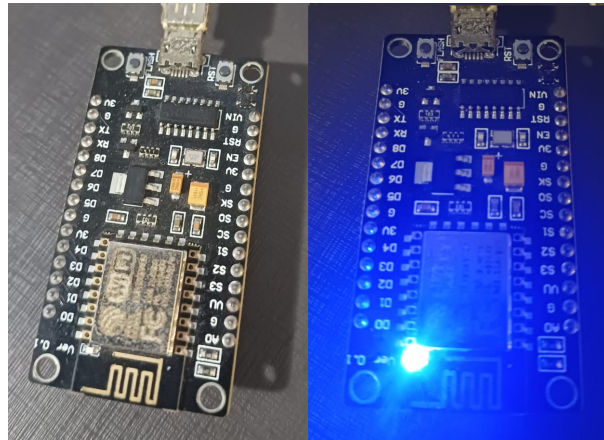
5.2 HOSPEDAGEM E CUSTOS

O sistema foi implantado em um ambiente de produção utilizando serviços de hospedagem em nuvem. A aplicação NextJS foi hospedada na plataforma Vercel². O servidor de sockets foi hospedado na plataforma Railway³. O banco de dados foi hospedado

² <https://vercel.com/>

³ <https://railway.app/>

Figura 21 – NodeMCU ao identificar ID esperado no socket.



Fonte: Criado pelo autor.

na plataforma ElephantSQL⁴. A implantação foi realizada sem maiores problemas e o sistema está disponível e funcionando como o esperado. A Tabela 2 exibe algumas faixas de precificação para os serviços utilizados. No momento o sistema opera com os planos gratuitos.

A hospedagem em nuvem permite a escalabilidade do sistema, o que significa que é possível aumentar a capacidade do sistema de acordo com a demanda, sem a necessidade de investimentos em infraestrutura própria.

5.3 CONSIDERAÇÕES FINAIS

Os resultados obtidos a partir da implementação do sistema são satisfatórios e atendem aos objetivos estabelecidos neste trabalho. O sistema desenvolvido é funcional, seguro e escalável, atendendo às necessidades. As tecnologias utilizadas mostraram-se adequadas para o desenvolvimento do sistema para automação de pagamentos e para a integração de um agente de distribuição exemplificado pelo NodeMCU com a aplicação web.

⁴ <https://www.elephantsql.com/>

Tabela 2 – Preços dos planos de hospedagem em nuvem.

Plataforma	Plano	Preço	Limites
Vercel	Hobby	Gratuito	Projetos pessoais ou não comerciais. Deploy a partir do CLI ou integrações git. HTTPS/SSL automático.
Vercel	Pro	\$20 por membro da equipe por mês.	1TB de banda. 1000 GB/horas de execução.
Vercel	Enterprise	Personalizado.	Infraestrutura de build isolada, em hardware melhor, sem filas.
Railway	Starter	Gratuito	\$5.00 em créditos de recursos todos os meses com tempo de execução de 500 horas.
Railway	Hobby	\$10 em créditos por mês	8 GB de RAM / 8 vCPU por serviço
Railway	Pro	\$20 por mês	Acesso compartilhado ao espaço de trabalho para equipes.
Railway	Enterprise	Personalizado	Personalizado
ElephantSQL	Tiny	Gratuito	20 MB, 5 conexões concorrentes
ElephantSQL	Simple	\$5 por mês	500 MB, 10 conexões concorrentes
ElephantSQL	Enormous	\$199 por mês	250 GB, centenas de conexões concorrentes

Fonte: Sites das respectivas plataformas.

6 CONCLUSÃO

A proposta deste projeto foi desenvolver um sistema para automação de pedido e compras em pontos de venda, que permita integração com agentes de distribuição, de forma a automatizar todo o processo.

Com base nos resultados apresentados, pode-se concluir que o desenvolvimento do sistema obteve sucesso. A aplicação web desenvolvida permite que os usuários possam fazer seus pedidos, enquanto os funcionários dos pontos de venda têm acesso a informações precisas sobre os pedidos realizados, podendo acompanhar o status da entrega em tempo real. Além disso, o servidor de *sockets* permite que os pontos de venda possam notificar os agentes de distribuição de qualquer atualização no status dos pedidos, de forma rápida e eficiente.

Por fim, a utilização de serviços de hospedagem em nuvem garantiu a disponibilidade e a escalabilidade do sistema, permitindo que ele possa ser usado por um grande número de usuários ao mesmo tempo.

Dessa forma, pode-se afirmar que o sistema de gerenciamento de pedidos para pontos de venda desenvolvido neste trabalho é uma solução eficiente e viável para empresas que desejam aprimorar seus processos de gerenciamento de pedidos e melhorar a experiência dos seus clientes, usando automação e tecnologias web modernas.

Para trabalhos futuros deseja-se incluir melhorias na interface, principalmente para exibir mais informações sobre os pedidos, e de forma geral torna-la mais amigável. Além disso, outro ponto que pode ser trabalhado seria o desenvolvimento de agentes de distribuição, conectando o NodeMCU com uma máquina de café, por exemplo, e distribuindo realmente os produtos.

REFERÊNCIAS

ADYEN Payment Methods. [S.l.: s.n.]. <https://www.adyen.com/payment-methods>. Accessed: 30-05-2023.

ATTOUI, Ilham. **Real-Time and Multi-Agent Systems**. [S.l.]: Springer, 2000.

AUÉ, Joop; ANICHE, Maurício; LOBBEZOO, Maikel; DEURSEN, Arie van. An exploratory study on faults in web API integration in a large-scale payment company. In: ACM. PROCEEDINGS of the 40th International Conference on Software Engineering. [S.l.: s.n.], 2018. p. 372–383.

BIERMAN, Gavin; ABADI, Martín; TORGERSEN, Mads. Understanding TypeScript, 2014. In: ECOOP. EUROPEAN Conference on Object-Oriented Programming. [S.l.: s.n.]. p. 257–281.

ELHADEF, M.; GRIRA, S. Performance Evaluation of a Game Theory-Based Fault Diagnosis Algorithm Under Partial Comparison Syndrome. In: 2016 IEEE International Conference on Computer and Information Technology (CIT). [S.l.: s.n.], 2016. p. 1–6.

FASTIFY: Fast and low overhead web framework, for Node.js. [S.l.: s.n.]. <https://www.fastify.io/>. Acessado em: 20 de junho de 2023.

GOURLEY, David; TOTTY, Brian. **HTTP: The Definitive Guide**. [S.l.]: O'Reilly Media, 2002.

HTTP An overview of HTTP | MDN. [S.l.: s.n.]. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>. Acessado em: 20 de junho de 2023.

INTRODUCTION to JSON Web Tokens. [S.l.]: JWT.io, 2023. Disponível em: <https://jwt.io/introduction/>.

IOT Architecture. [S.l.: s.n.], 2023. <https://www.zibtek.com/blog/iot-architecture>. Acessado em: 26 de junho de 2023.

KOLBAN, N. **Kolban's book on ESP8266**. [S.l.]: Leanpub, 2016.

LEACH, P.; MEALLING, M.; SALZ, R. **A Universally Unique IDentifier (UUID) URN Namespace**. [S.l.: s.n.], 2005. Disponível em: <https://www.rfc-editor.org/info/rfc4122>.

LIU, Qian; SUN, Xiaojun. Research of Web Real-Time Communication Based on Web Socket. **International Journal of Communications, Network and System Sciences**, v. 5, n. 12, p. 841–846, 2012.

MADAKAM, Somayya; RAMASWAMY, R.; TRIPATHI, Siddharth. Internet of Things (IoT): A Literature Review. **Journal of Computer and Communications**, Scientific Research Publishing, v. 3, n. 5, p. 164–173, 2015.

MULLOY, Brian. **Web API Design: The Missing Link**. [S.l.: s.n.], 2012. Disponível em: <https://pages.apigee.com/rs/apigee/images/api-design-ebook-2012-03.pdf>.

NEXT.JS Documentation. [S.l.]: Vercel, 2023. Disponível em: <https://nextjs.org/docs>.

NIVASALO, Markus. **Full Stack TypeScript Development with tRPC and React Native**. 2022. Disponível em: <https://www.theseus.fi/handle/10024/793117>.

NODE MCU. [S.l.: s.n.], 2023. <https://www.amazon.com.br/ESP8266-CH340G-NodeMcu-desenvolvimento-Internet/dp/B08H26NY16>. Acessado em: 26 de junho de 2023.

PRISMA Documentation. [S.l.]: Prisma, 2023. Disponível em: <https://www.prisma.io/docs/>.

QR Code Tutorial. [S.l.]: W3Schools, 2023. Disponível em: https://www.w3schools.com/whatis/whatis_qrcode.asp.

RICHARDSON, Leonard; RUBY, Sam. **RESTful Web Services**. [S.l.]: O'Reilly Media, 2007.

SQL Tutorial. [S.l.]: W3Schools, 2023. Disponível em: <https://www.w3schools.com/sql/>.

TRPC Documentation - End-to-end typesafe APIs. Accessed: 30-05-2023. Disponível em: <https://trpc.io/>.

VERCEL. **AWS and Vercel: Accelerating innovation with serverless computing**. 2023. Disponível em: <https://vercel.com/blog/aws-and-vercel-accelerating-innovation-with-serverless-computing>.

ZOD Documentation. [S.l.]: NPM, 2023. Disponível em: <https://www.npmjs.com/package/zod>.