FATEC JAHU

DESENVOLVIMENTO DE SOFTWARE MULTIPLATAFORMA

THIAGO FRANCA DE FIGUEREDO

FOMEZAP: PLATAFORMA DE CRIAÇÃO DE CARDÁPIO DIGITAL PARA RESTAURANTES E DELIVERYS

JAÚ

2025

THIAGO FRANCA DE FIGUEREDO

FOMEZAP: PLATAFORMA DE CRIAÇÃO DE CARDÁPIO DIGITAL PARA RESTAURANTES E DELIVERYS

Projeto Interdisciplinar apresentado ao Curso de Desenvolvimento de Software Multiplataforma da Fatec Jahu.

JAÚ

2025

RESUMO

O crescimento do mercado de delivery no Brasil, intensificado pela pandemia de COVID-19, evidenciou a necessidade de soluções tecnológicas acessíveis para pequenos e médios restaurantes que buscam alternativas aos aplicativos de delivery tradicionais, cujos custos de comissão podem atingir até 30% por pedido. Este trabalho apresenta o desenvolvimento do FomeZap, um sistema SaaS (Software as a Service) multi-tenant completo para gerenciamento de pedidos para delivery de restaurantes, com arquitetura escalável e custos operacionais reduzidos. O sistema foi desenvolvido utilizando tecnologias modernas como React no frontend, Node.js com Express no backend, e MongoDB Atlas como banco de dados em nuvem. A arquitetura multi-tenant permite que múltiplos restaurantes (tenants) compartilhem a mesma infraestrutura mantendo total isolamento de dados, otimizando recursos e reduzindo custos. O deploy foi realizado utilizando plataformas cloud nos planos gratuitos (Vercel para frontend e Render.com para backend) com implementação de CI/CD (Continuous Integration/Continuous Deployment) através do GitHub, garantindo atualizações automáticas e confiáveis. O sistema oferece funcionalidades completas incluindo painel administrativo para gestão de produtos, categorias e pedidos; cardápio público responsivo para clientes; autenticação JWT com diferentes níveis de permissão (super admin e admin de restaurante); e isolamento de dados por tenant.

Os resultados demonstram que é possível desenvolver e manter um MVP com custo operacional de R$0/mês no plano gratuito ou a partir de U$7/mês no plano escalável, representando economia significativa comparada às soluções existentes no mercado. O sistema foi validado através de testes funcionais completos, incluindo criação de tenants, gerenciamento de produtos, realização de pedidos e visualização em tempo real. Como trabalhos futuros, sugere-se a implementação de aplicativo mobile com Flutter, integração com sistemas de pagamento, notificações push, e funcionalidades de analytics avançado. Este trabalho contribui para a democratização do acesso à tecnologia de delivery, oferecendo uma solução viável e escalável para restaurantes de pequeno e médio porte.

Palavras-chave: SaaS. Multi-tenancy. Delivery. Cloud Computing.

ABSTRACT

FomeZap: Development of a Multi-tenant platform System for Restaurant Delivery Management. 2025. Multiplatform Software Development - FATEC, Jaú, 2025.

The growth of the delivery market in Brazil, intensified by the COVID-19 pandemic, highlighted the need for accessible technological solutions for small and medium-sized restaurants seeking alternatives to traditional delivery apps, whose commission costs can reach up to 30% per order. This work presents the development of FomeZap, a complete multi-tenant SaaS (Software as a Service) system for restaurant delivery management, with scalable architecture and reduced operational costs. The system was developed using modern technologies such as React 19 on the frontend, Node.js 20 with Express 5 on the backend, and MongoDB Atlas as a cloud database. The multi-tenant architecture allows multiple restaurants (tenants) to share the same infrastructure while maintaining complete data isolation, optimizing resources and reducing costs. Deployment was performed using free cloud platforms (Vercel for frontend and Render.com for backend) with CI/CD (Continuous Integration/Continuous Deployment) implementation through GitHub, ensuring automatic and reliable updates. The system offers complete functionalities including administrative panel for product, category, and order management; responsive public menu for customers; JWT authentication with different permission levels (super admin and restaurant admin); and data isolation per tenant.

The results demonstrate that it is possible to develop and maintain a professional SaaS system with operational cost of $0/month on the free plan or $7/month on the scalable plan, representing significant savings compared to existing market solutions. The system was validated through complete functional tests, including tenant creation, product management, order placement, and real-time visualization. As future work, implementation of mobile application, integration with payment systems, push notifications, and advanced analytics functionalities are suggested. This work

contributes to democratizing access to delivery technology, offering a viable and scalable solution for small and medium-sized restaurants.

Keywords: SaaS. Multi-tenancy. Delivery. Cloud Computing. Web Application. React. Node.js. MongoDB.

1. CONTEXTUALIZAÇÃO

O mercado de food delivery tem experimentado crescimento exponencial nos últimos anos, especialmente após a pandemia de COVID-19, que acelerou a transformação digital no setor alimentício (SEBRAE, 2021). Segundo dados da Associação Brasileira de Bares e Restaurantes (ABRASEL, 2023), o segmento de delivery representa atualmente cerca de 30% do faturamento dos estabelecimentos brasileiros, movimentando aproximadamente R$ 20 bilhões por ano.

No entanto, a utilização de grandes plataformas de delivery como iFood, Rappi e Uber Eats apresenta desafios significativos para pequenos e médios restaurantes. As taxas de comissão cobradas por esses aplicativos variam entre 12% e 30% do valor de cada pedido (FOOD CONNECTION, 2023), impactando diretamente na margem de lucro dos estabelecimentos. Além disso, essas plataformas intermediam completamente a relação entre restaurante e cliente, dificultando a fidelização e o relacionamento direto.

Diante desse cenário, surge a demanda por soluções tecnológicas alternativas que permitam aos restaurantes gerenciarem seu próprio sistema de delivery, mantendo controle sobre seus dados, clientes e operações, com custos operacionais reduzidos. A arquitetura SaaS (Software as a Service) multi-tenant apresenta-se como uma solução viável, permitindo que múltiplos estabelecimentos compartilhem a mesma infraestrutura tecnológica mantendo total isolamento de dados e personalização.

O modelo SaaS revolucionou a forma como software é distribuído e consumido (CHONG; CARRARO, 2006), oferecendo vantagens como eliminação de instalação local, atualizações automáticas, acesso de qualquer lugar, e modelo de pagamento por uso. A arquitetura multi-tenant, por sua vez, otimiza recursos ao permitir que uma única instância da aplicação atenda múltiplos clientes (tenants), reduzindo custos de infraestrutura e manutenção (BEZEMER; ZAIDMAN, 2010).

1.2 PROBLEMATIZAÇÃO

Os restaurantes de pequeno e médio porte enfrentam diversos desafios ao optarem por soluções de delivery:

Altos custos de comissão: Plataformas tradicionais cobram entre 12% e 30% por pedido, comprometendo significativamente a rentabilidade do negócio. Um restaurante que fatura R$ 10.000 mensais em delivery pode pagar até R$ 3.000 em comissões.

Dependência tecnológica: Os estabelecimentos ficam reféns das políticas, funcionalidades e disponibilidade das plataformas terceirizadas, sem controle sobre sua própria operação digital.

Perda de relacionamento com cliente: A intermediação completa impede que o restaurante construa relacionamento direto com seus clientes, acesse dados de consumo, ou implemente estratégias próprias de fidelização.

Falta de personalização: As plataformas oferecem layouts padronizados, limitando a identidade visual e experiência de marca do restaurante.

Custos proibitivos de desenvolvimento próprio: Desenvolver e manter um sistema de delivery proprietário demanda investimento técnico significativo, geralmente inviável para pequenos estabelecimentos. Estima-se que o desenvolvimento de uma solução completa custe entre R$ 40.000 e R$ 100.000, além de custos mensais de manutenção.

Surge, portanto, a seguinte questão de pesquisa: É possível desenvolver um sistema SaaS multi-tenant escalável e economicamente viável que permita a restaurantes de pequeno e médio porte gerenciarem seu próprio delivery, mantendo custos operacionais reduzidos e controle completo sobre seus dados e operações?

1.3 OBJETIVOS

1.3.1 Objetivo Geral

Desenvolver um sistema SaaS multi-tenant completo para gerenciamento de delivery de restaurantes, com arquitetura escalável, custos operacionais reduzidos, e funcionalidades que atendam às necessidades de pequenos e médios estabelecimentos, oferecendo alternativa viável às plataformas de delivery tradicionais.

1.3.2 Objetivos Específicos

a) **Projetar arquitetura multi-tenant** que garanta isolamento de dados entre restaurantes (tenants) compartilhando a mesma infraestrutura, otimizando recursos e custos;

b) **Implementar sistema de autenticação e autorização** baseado em JWT (JSON Web Tokens) com diferentes níveis de permissão (super administrador e administrador de restaurante);

c) **Desenvolver painel administrativo** completo com funcionalidades de gestão de produtos, categorias, extras, configurações do estabelecimento e visualização de pedidos em tempo real;

d) **Criar cardápio público responsivo** acessível via web, permitindo que clientes visualizem produtos, adicionem ao carrinho e finalizem pedidos sem necessidade de instalação de aplicativo;

e) **Implementar sistema de pedidos** com rastreamento de status, cálculo automático de valores, e isolamento por tenant;

f) **Realizar deploy em plataformas cloud gratuitas** (Vercel, Render.com, MongoDB Atlas) demonstrando viabilidade econômica da solução;

g) **Implementar CI/CD** (Continuous Integration/Continuous Deployment) automatizado através do GitHub para garantir atualizações confiáveis e ágeis;

h) **Validar o sistema** através de testes funcionais completos simulando operação real de restaurante;

i) **Analisar custos operacionais** e comparar com soluções existentes no mercado, demonstrando viabilidade econômica;

j) **Documentar arquitetura e processo de desenvolvimento** de forma detalhada, permitindo replicação e evolução do sistema.

1.4 JUSTIFICATIVA

O desenvolvimento deste trabalho justifica-se por múltiplas perspectivas:

Perspectiva Econômica: Segundo dados do SEBRAE (2021), 60% dos restaurantes são de pequeno porte, com faturamento mensal inferior a R$ 50.000. Para esses estabelecimentos, pagar 20-30% de comissão por pedido representa impacto financeiro insustentável. Uma solução com custo fixo reduzido (ou gratuito inicialmente) pode viabilizar a permanência desses negócios no mercado digital.

Perspectiva Tecnológica: Este trabalho explora arquitetura SaaS multi-tenant, padrão consolidado em empresas como Salesforce, Slack e Shopify, aplicando-a ao contexto de delivery. A implementação demonstra conceitos modernos de desenvolvimento web (React, Node.js, MongoDB), cloud computing, e DevOps (CI/CD), contribuindo para o conhecimento técnico na área.

Perspectiva Social: Ao democratizar acesso à tecnologia de delivery, o sistema permite que pequenos empreendedores compitam digitalmente, mantendo empregos e

fomentando a economia local. Restaurantes familiares, food trucks, e estabelecimentos de bairro podem oferecer delivery sem depender de grandes corporações.

Perspectiva Acadêmica: O trabalho abrange múltiplas áreas do conhecimento em Tecnologia da Informação: engenharia de software, arquitetura de sistemas, banco de dados, segurança da informação, cloud computing, e DevOps. Serve como estudo de caso completo de desenvolvimento full-stack moderno.

Relevância Atual: Com o crescimento do comércio eletrônico pós-pandemia (aumento de 73,88% segundo ABComm, 2021), soluções que facilitem a transformação digital de pequenos negócios são estratégicas para recuperação econômica do setor.

1.5 DELIMITAÇÃO DO TRABALHO

Este trabalho delimita-se ao desenvolvimento de um sistema web para gerenciamento de delivery de restaurantes, com as seguintes características:

Escopo Funcional:

- Gestão de tenants (restaurantes) por super administrador
- Gestão de produtos, categorias e extras por administrador do restaurante
- Cardápio público web responsivo para clientes
- Sistema de pedidos com carrinho de compras
- Painel administrativo para visualização de pedidos
- Autenticação e autorização multi-nível

Escopo Tecnológico:

- Frontend: React com Vite, Tailwind CSS
- Backend: Node.js com Express
- Banco de dados: MongoDB Atlas (cloud)

- Deploy: Vercel (frontend), Render.com (backend)
- CloudFlare (Gerenciamento de zona DNS de domínio e SSL)
- Hostgator (Domínio ganho ‘fomezap.com’ e hospedagem que eu já tinha para hospedar site de venda do FomeZap)
- Gmail (Senha APP para sistema de recuperação de senha)
- Versionamento: Git/GitHub, Docker e Kind (Local)

Limitações:

- Sistema acessível apenas via web (não inclui app mobile nativo)
- Pagamentos não integrados (pedidos registrados mas pagamento offline)
- Notificações por e-mail/SMS não implementadas
- Sistema de delivery/tracking em tempo real não incluído
- Analytics avançado não implementado
- React tem dificuldade indexação de SEO, cardápio público deverá ser migrado para NEXT.js por ser server-side rendering melhorando indexação nos motores de busca.

Público-Alvo:

- Restaurantes de pequeno e médio porte (até 20 funcionários)
- Estabelecimentos que já possuem estrutura própria de entrega ou pretender implementar parceria com motoboys
- Negócios que buscam autonomia sobre seu delivery

1.6 ESTRUTURA DO TRABALHO

Este trabalho está organizado em seis capítulos:

Capítulo 1 – Introdução: Apresenta contextualização do tema, problematização, objetivos (geral e específicos), justificativa, delimitação do trabalho e estrutura do documento.

Capítulo 2 – Referencial Teórico: Fundamenta teoricamente o trabalho, abordando conceitos de SaaS, arquitetura multi-tenant, tecnologias web modernas (React, Node.js, MongoDB), cloud computing, e padrões de desenvolvimento.

Capítulo 3 – Metodologia: Descreve a abordagem metodológica adotada, ferramentas utilizadas, processo de desenvolvimento, e métodos de validação do sistema.

Capítulo 4 – Desenvolvimento do Sistema: Detalha a arquitetura projetada, decisões técnicas, implementação dos módulos (frontend, backend, banco de dados), integração entre componentes, e processo de deploy.

Capítulo 5 – Resultados e Discussão: Apresenta os resultados obtidos, testes realizados, análise de custos operacionais, comparação com soluções existentes, e discussão sobre limitações e desafios encontrados.

Capítulo 6 – Conclusão: Sintetiza as contribuições do trabalho, verifica o atingimento dos objetivos, e propõe trabalhos futuros e melhorias.

1.7 CRIAÇÃO DA MARCA

Processo de Criação do Logo

O desenvolvimento do logo do FOMEZAP seguiu uma abordagem centrada na simplicidade e identificação imediata com o público-alvo. Foram realizadas pesquisas sobre elementos visuais que remetem à alimentação e à comunicação digital, buscando criar uma identidade visual que fosse facilmente reconhecida e associada ao propósito do projeto. O resultado foi um logo que combina ícones relacionados à comida com elementos gráficos que remetem ao aplicativo WhatsApp, como o balão de conversa, reforçando a ideia de comunicação rápida e acessível.

Análise dos Elementos Visuais do Logo FOMEZAP

Cores Utilizadas

Os logos do FOMEZAP utilizam predominantemente tons de laranja e verde. O laranja, no contexto alimentar, estimula o apetite e está relacionado ao bem-estar, conforto e sociabilidade. Pesquisas indicam que ambientes e marcas que utilizam o laranja tendem a transmitir proximidade, alegria e dinamismo, favorecendo a interação e o engajamento dos usuários. Por essas razões, a presença do laranja no logo do FOMEZAP reforça a proposta de acolhimento e incentivo à alimentação, tornando a identidade visual mais convidativa e alinhada ao propósito deste projeto.

O verde faz referência direta ao WhatsApp, reforçando a associação com o aplicativo de comunicação.

Fontes

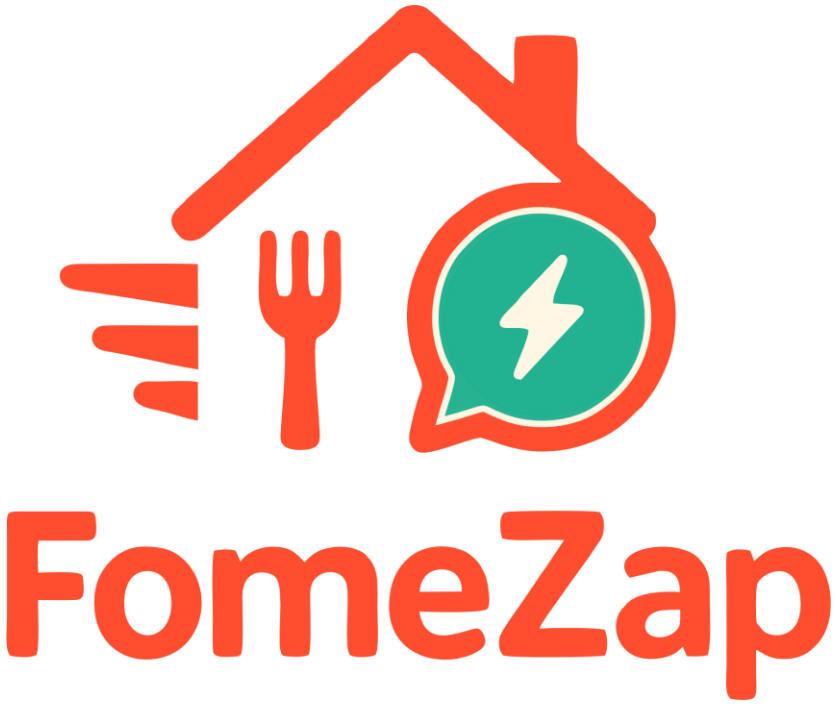
A tipografia escolhida para o logo é simples, sem serifa, com traços arredondados e modernos. Essa escolha visa garantir legibilidade e transmitir uma imagem contemporânea, acessível e próxima do público. O uso de letras maiúsculas em "FOMEZAP" reforça a força da marca e facilita a memorização.

Elementos Gráficos

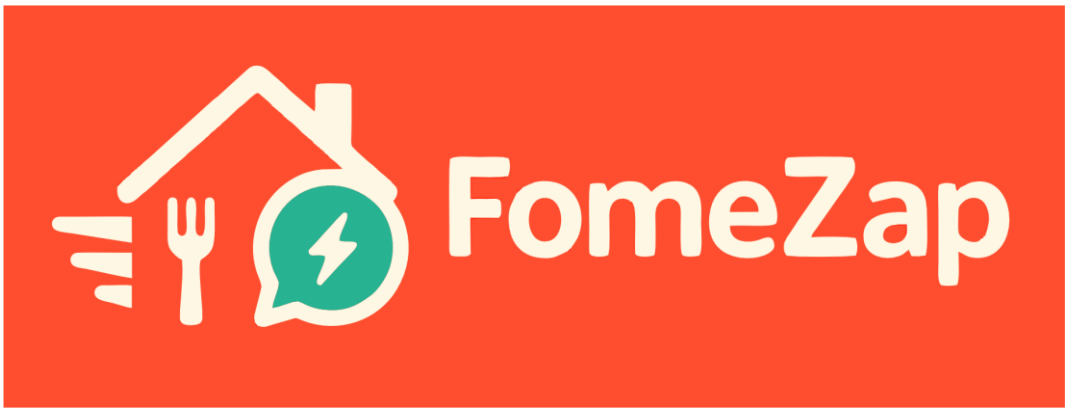
Os logos analisados apresentam ícones relacionados à comida, como talheres estilizados ou pratos, integrados ao balão de conversa típico do WhatsApp. Essa combinação reforça o propósito do projeto: unir alimentação e comunicação digital. O balão de conversa, elemento central, simboliza a troca de mensagens e o acesso rápido à uma forma de realizer seu pedido de forma rápida e segura.

Justificativa Visual

A integração dos elementos visuais foi pensada para criar uma identidade marcante e facilmente reconhecível pelo público brasileiro. O uso das cores, fontes e ícones familiares facilita a identificação e o engajamento dos usuários, tornando o FOMEZAP uma solução inovadora, inclusiva e alinhada à cultura local.



[Figura 01 – Logo formato quadrado]



[Figura 02 – Logo formato horizontal]

Justificação do Nome da Marca

O nome FOMEZAP foi escolhido por sua forte conexão com o contexto sociocultural brasileiro. "Fome" representa diretamente a condição do cliente que deseja pedir uma refeição do conforto do seu lar, que seja de forma rápida, segura e confiáel. Já "Zap" é uma forma popular de se referir ao aplicativo WhatsApp, amplamente utilizado pela população brasileira para comunicação cotidiana. A junção dos termos cria uma marca de fácil memorização, que transmite de maneira clara a proposta de conectar pessoas que buscam uma forma prática de pedir uma refeição por meio de delivery ou para retirada com restautantes que querem oferecer uma forma de atendimento automatizado e oferecer uma excelente experiência ao usuáio por meio de uma ferramenta digital responsive, de fácil uso e acessível.

CAPÍTULO 2 - REFERENCIAL TEÓRICO

2.1 SOFTWARE AS A SERVICE (SAAS)

2.1.1 Conceito e Evolução

Software as a Service (SaaS) representa um modelo de distribuição de software onde aplicações são hospedadas por provedor de serviço e disponibilizadas aos clientes via internet (MELL; GRANCE, 2011). Diferentemente do modelo tradicional de software licenciado, onde o cliente adquire e instala o produto em sua própria infraestrutura, no modelo SaaS o software é acessado como serviço, geralmente através de navegador web ou APIs.

Segundo Chong e Carraro (2006), a evolução do SaaS pode ser dividida em gerações:

Primeira Geração (ASP - Application Service Provider): Cada cliente recebia instância dedicada do software, hospedada no datacenter do provedor. Modelo ineficiente devido à duplicação de recursos e dificuldade de manutenção.

Segunda Geração (SaaS Configurável): Introdução de arquitetura multi-tenant, onde uma única instância atende múltiplos clientes com configurações personalizadas. Ganhos significativos em escalabilidade e economia.

Terceira Geração (SaaS Multi-tenant Escalável): Arquitetura moderna com isolamento de dados, multi-tenancy em todas as camadas, alta disponibilidade, e elasticidade automática. Exemplos: Salesforce, Shopify, Slack.

2.1.2 Características do Modelo SaaS

O National Institute of Standards and Technology (NIST) define cinco características essenciais do cloud computing aplicáveis ao SaaS (MELL; GRANCE, 2011):

On-demand self-service: Usuários podem provisionar recursos computacionais automaticamente, sem necessidade de interação humana com provedor.

Broad network access: Acesso disponível via rede através de mecanismos padrão (navegadores, apps mobile).

Resource pooling: Recursos do provedor são agrupados para atender múltiplos consumidores usando modelo multi-tenant.

Rapid elasticity: Capacidades podem ser elasticamente provisionadas e liberadas, escalando automaticamente conforme demanda.

Measured service: Sistemas controlam e otimizam uso de recursos automaticamente, fornecendo transparência tanto para provedor quanto para consumidor.

2.1.3 Vantagens e Desvantagens

Vantagens do modelo SaaS (BEZEMER; ZAIDMAN, 2010):

a) **Redução de custos:** Eliminação de investimento inicial em licenças, hardware e infraestrutura. Modelo de pagamento por uso.

b) **Acessibilidade:** Acesso de qualquer lugar com conexão internet, facilitando trabalho remoto.

c) **Atualizações automáticas:** Correções e novas funcionalidades implementadas pelo provedor, sem necessidade de ação do cliente.

d) **Escalabilidade:** Capacidade de crescer ou reduzir recursos conforme necessidade, sem reconfiguração.

e) **Manutenção simplificada:** Responsabilidade do provedor, reduzindo necessidade de equipe técnica no cliente.

f) **Time-to-market reduzido:** Implementação rápida, sem processos longos de instalação e configuração.

Desvantagens e desafios:

a) **Dependência de conectividade:** Requer conexão internet estável.

b) **Segurança e privacidade:** Dados armazenados em servidores terceiros, exigindo confiança no provedor.

c) **Personalização limitada:** Nem sempre permite customizações específicas como software on-premise.

d) **Vendor lock-in:** Dificuldade de migração para outro provedor devido a dependências tecnológicas.

2.2 ARQUITETURA MULTI-TENANT

2.2.1 Conceito de Multi-tenancy

Multi-tenancy é padrão arquitetural onde uma única instância de software atende múltiplos clientes (tenants), mantendo isolamento lógico de dados e configurações de cada cliente (GUO et al., 2007). Cada tenant compartilha a mesma infraestrutura computacional, mas percebe o sistema como exclusivo.

Segundo Bezemer e Zaidman (2010), multi-tenancy difere de multi-user: em sistemas multi-user, múltiplos usuários de uma mesma organização acessam o sistema; em multi-tenant, múltiplas organizações independentes compartilham a mesma instância.

2.2.2 Níveis de Isolamento

Chong e Carraro (2006) propõem quatro níveis de maturidade para multi-tenancy:

Nível 1 - Aplicação Customizada para cada Tenant:

- Cada tenant possui instância separada do código e banco de dados
- Baixa eficiência, alto custo de manutenção
- Não é verdadeiro multi-tenant

Nível 2 - Aplicação Compartilhada, Banco de Dados Separado:

- Mesma aplicação para todos os tenants
- Banco de dados dedicado para cada tenant
- Melhor manutenibilidade do código
- Ainda requer múltiplas instâncias de banco

Nível 3 - Aplicação e Banco Compartilhados, Schemas Separados:

- Uma instância da aplicação e banco de dados
- Cada tenant possui schema (conjunto de tabelas) próprio
- Bom isolamento, complexidade média

Nível 4 - Aplicação e Banco Totalmente Compartilhados:

- Uma instância da aplicação e banco de dados
- Todas as tabelas compartilhadas, isolamento por campo discriminador (tenantId)
- Máxima eficiência e escalabilidade
- Requer cuidado extra com segurança e isolamento

O FomeZap implementa Nível 4, utilizando MongoDB com campo tenantId em todas as collections para isolamento lógico.

2.2.3 Desafios da Arquitetura Multi-tenant

Isolamento de dados: Principal desafio é garantir que um tenant nunca acesse dados de outro. Requer validação rigorosa em todas as queries (BEZEMER; ZAIDMAN, 2010).

Performance: Tenant com alto volume de requisições pode impactar performance de outros (noisy neighbor problem). Soluções: rate limiting, resource throttling.

Customização: Equilibrar necessidade de personalização por tenant mantendo código unificado. Padrões como Strategy Pattern e Feature Flags auxiliam.

Escalabilidade: Sistema deve crescer horizontalmente adicionando recursos, não verticalmente. Requer arquitetura stateless e uso de filas.

Backup e Restore: Backup por tenant para permitir restauração individual sem afetar outros.

2.3 TECNOLOGIAS WEB MODERNAS

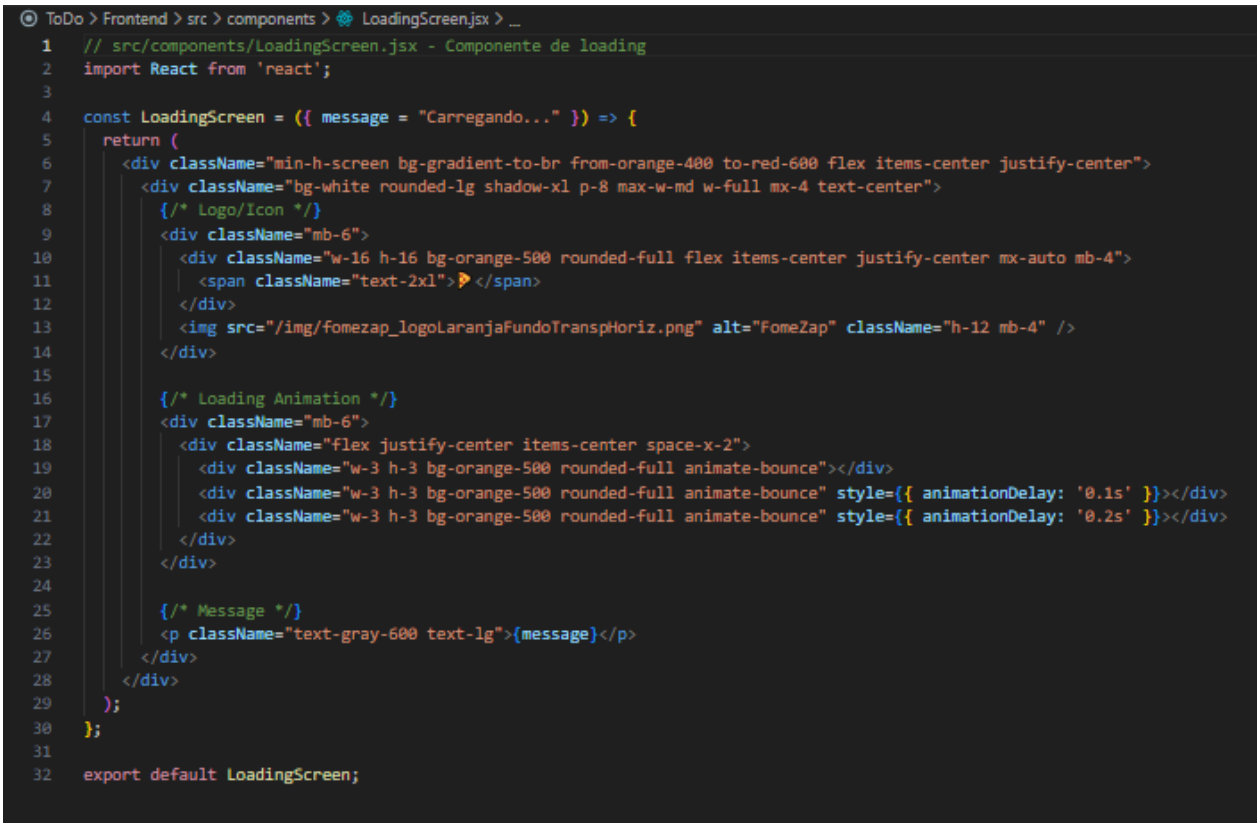
2.3.1 React

React é biblioteca JavaScript desenvolvida pelo Facebook (atual Meta) para construção de interfaces de usuário (BANKS; PORCELLO, 2020). Lançada em 2013, tornou-se padrão de mercado para desenvolvimento frontend.

Características principais:

Component-Based Architecture: Interface dividida em componentes reutilizáveis e independentes. Cada componente encapsula lógica, estado e apresentação.

Exemplo de component loading React:



[Figura 03 – Print de parte do código]

Virtual DOM: React mantém representação virtual da DOM em memória, calculando diferenças (diffing) e atualizando apenas elementos necessários no DOM real, resultando em performance superior (BANKS; PORCELLO, 2020).

Unidirectional Data Flow: Dados fluem em direção única (parent → child), facilitando debugging e manutenção.

Hooks: Introduzidos em 2019, Hooks como useState, useEffect, useContext permitem uso de estado e lifecycle em componentes funcionais, eliminando necessidade de classes (REACT, 2024).

Ecossistema: Vasto ecossistema com bibliotecas como React Router (roteamento), Redux/Context API (gerenciamento de estado), e Tailwind CSS (estilização).

2.3.2 Node.js e Express

Node.js é runtime JavaScript construído sobre V8 engine do Chrome, permitindo execução de JavaScript no servidor (NODEJS, 2024). Express é framework web minimalista para Node.js, facilitando criação de APIs REST.

Características do Node.js:

Event-Driven, Non-Blocking I/O: Arquitetura baseada em eventos com I/O assíncrono permite alta concorrência com uso eficiente de recursos (TILKOV; VINOSKI, 2010).

Single-Threaded com Event Loop: Embora single-threaded, Node.js utiliza event loop para gerenciar múltiplas conexões simultaneamente sem overhead de threads.

NPM (Node Package Manager): Maior repositório de pacotes de software do mundo, com mais de 2 milhões de pacotes (NPM, 2024).

Características do Express:

Middleware Architecture: Requisições passam por cadeia de funções middleware, cada uma processando ou modificando request/response.

```
// Exemplo conceitual de middleware Express
app.use(verificarAutenticacao);
app.use(verificarTenant);
app.get('/api/produtos', listarProdutos);
```

[PARTE DO CÓDIGO: Backend/index.js - configuração de middlewares]

Roteamento Flexível: Sistema de rotas baseado em padrões, suportando parâmetros, query strings, e métodos HTTP.

Performance: Express é extremamente leve, adicionando overhead mínimo sobre Node.js puro.

2.3.3 MongoDB

MongoDB é banco de dados NoSQL orientado a documentos, armazenando dados em formato JSON-like (BSON - Binary JSON) ao invés de tabelas relacionais (MONGODB, 2024).

Características:

Schema Flexível: Documentos em mesma collection podem ter estruturas diferentes, permitindo evolução do modelo sem migrations complexas.

Escalabilidade Horizontal: Suporte nativo a sharding (particionamento de dados entre servidores) e replicação.

Consultas Ricas: Suporta consultas complexas, índices, agregações, e geolocalização.

Atomicidade em Nível de Documento: Operações em único documento são atômicas, garantindo consistência.

Adequação ao Multi-tenancy:

MongoDB é particularmente adequado para arquiteturas multi-tenant devido a:

a) **Flexibilidade de schema:** Cada tenant pode ter customizações sem alterar estrutura global.

b) **Performance de queries com discriminador:** Índices compostos (tenantId + outros campos) garantem isolamento eficiente.

c) **JSON nativo:** Integração natural com JavaScript (Node.js/React).

[Figura 04 - Print de um tenant no banco]

2.4 CLOUD COMPUTING E DEVOPS

2.4.1 Cloud Computing

Cloud computing é modelo de provisionamento de recursos computacionais (servidores, armazenamento, aplicações) via internet, com pagamento por uso (MELL; GRANCE, 2011).

Modelos de Serviço:

IaaS (Infrastructure as a Service): Provê infraestrutura virtual (servidores, rede, armazenamento). Exemplos: AWS EC2, Google Compute Engine.

PaaS (Platform as a Service): Provê plataforma completa para desenvolvimento e deploy. Exemplos: Render, Vercel.

SaaS (Software as a Service): Provê aplicação completa via internet. Exemplos: Salesforce, Gmail, Alboom.

Vantagens para SaaS:

a) **Elasticidade:** Escalabilidade automática conforme demanda.

b) **Disponibilidade:** Redundância geográfica e alta disponibilidade (99.9%+ SLA).

c) **Custo:** Modelo pay-as-you-go elimina investimento inicial em infraestrutura.

d) **Manutenção:** Provedor responsável por manutenção de hardware, segurança física, e atualizações.

2.4.2 CI/CD (Continuous Integration/Continuous Deployment)

CI/CD é conjunto de práticas para automatizar integração, teste e deploy de código (HUMBLE; FARLEY, 2010).

Continuous Integration (CI):

- Desenvolvedores integram código ao repositório principal frequentemente
- Cada integração é verificada por build e testes automatizados

- Detecta erros rapidamente

Continuous Deployment (CD):

- Código que passa por testes é automaticamente deployado em produção
- Reduz time-to-market
- Minimiza intervenção manual

Pipeline CI/CD típico:

1. Developer faz commit/push no Git
2. GitHub detecta mudança e notifica plataformas (Vercel, Render)
3. Plataformas baixam código e executam build
4. Testes automatizados são executados
5. Se testes passam, deploy é realizado
6. Monitoramento detecta problemas em produção

Benefícios:

a) **Feedback rápido:** Erros detectados em minutos, não dias.

b) **Confiabilidade:** Processo repetível e testado reduz erros humanos.

c) **Velocity:** Múltiplos deploys por dia ao invés de releases mensais.

d) **Rollback:** Fácil retornar versão anterior se problemas surgirem.

2.5 AUTENTICAÇÃO E SEGURANÇA

2.5.1 JSON Web Tokens (JWT)

JWT é padrão aberto (RFC 7519) para transmissão segura de informações entre partes como objeto JSON (JONES et al., 2015). No contexto de autenticação web, JWT substitui sessões tradicionais server-side.

Estrutura do JWT:

JWT consiste em três partes separadas por pontos:

header.payload.signature

Header: Contém tipo de token (JWT) e algoritmo de criptografia (ex: HS256).

Payload: Contém claims (declarações) sobre usuário, como ID, role, tenantId.

Signature: Garante integridade do token, gerada com chave secreta.

// Exemplo conceitual de geração JWT

```
// Gerar token JWT
const token = jwt.sign(
  {
    userId: novoUsuario._id,
    email: novoUsuario.email,
    role: novoUsuario.role,
    tenantId: novoUsuario.tenantId
  },
  JWT_SECRET,
  { expiresIn: JWT_EXPIRES_IN }
);
```

[Figura 05 - Geração de token JWT]

Vantagens:

a) **Stateless:** Servidor não precisa armazenar sessões, facilitando escalabilidade horizontal.

b) **Self-contained:** Todas informações necessárias estão no token.

c) **Portabilidade:** Funciona em diferentes domínios e plataformas.

d) **Performance:** Validação rápida sem consultar banco de dados.

2.5.2 Hashing de Senhas com Argon2

Argon2 é algoritmo de hashing de senhas vencedor da Password Hashing Competition (2015), considerado estado-da-arte em segurança (BIRYUKOV et al., 2015). Substitui algoritmos mais antigos como bcrypt e PBKDF2.

Características:

Memory-hard: Requer quantidade significativa de memória, dificultando ataques com GPUs/ASICs.

Configurável: Permite ajustar uso de memória, tempo, e paralelismo conforme necessidade.

Resistente a side-channel: Protegido contra ataques que exploram tempo de execução ou consumo de energia.

Exemplo de hashing com Argon2

```
    try {
      // Gerar hash com argon2
      this.senha = await argon2.hash(this.senha);
      next();
    } catch (error) {
      next(error);
    }
});

// Método para comparar senha
userSchema.methods.compararSenha = async function(senhaInformada) {
  try {
    return await argon2.verify(this.senha, senhaInformada);
  } catch (error) {
    return false;
  }
};
```

[Figura 06: - Hashing de senha]

2.5.3 CORS (Cross-Origin Resource Sharing)

CORS é mecanismo que permite que recursos em servidor web sejam requisitados por página de domínio diferente (W3C, 2020). Fundamental em arquiteturas separadas (frontend e backend em domínios distintos).

Problema: Por padrão, navegadores bloqueiam requisições cross-origin por segurança (Same-Origin Policy).

Solução: Servidor inclui headers HTTP específicos autorizando origem:

Access-Control-Allow-Origin: https://meu-frontend.com
Access-Control-Allow-Methods: GET, POST, PUT, DELETE
Access-Control-Allow-Headers: Authorization, Content-Type

2.6 PADRÕES DE PROJETO

2.6.1 MVC (Model-View-Controller)

MVC é padrão arquitetural que separa aplicação em três componentes interconectados (GAMMA et al., 1994):

Model: Representa dados e lógica de negócio. No FomeZap, corresponde aos schemas Mongoose (Produto, Pedido, Tenant).

View: Interface com usuário. No FomeZap, componentes React.

Controller: Intermediário entre Model e View, processa requisições e coordena atualizações. No FomeZap, controllers Express (ProdutoController, PedidoController).

Benefícios:

a) **Separação de responsabilidades:** Mudanças em UI não afetam lógica de negócio.

b) **Reutilização:** Models podem ser usados por múltiplas views.

c) **Testabilidade:** Componentes podem ser testados isoladamente.

2.6.2 RESTful API

REST (Representational State Transfer) é estilo arquitetural para sistemas distribuídos, amplamente adotado em APIs web (FIELDING, 2000).

Princípios:

Client-Server: Separação clara entre cliente e servidor.

Stateless: Cada requisição contém todas informações necessárias.

Cacheable: Respostas devem indicar se podem ser cacheadas.

Uniform Interface: Interface consistente usando métodos HTTP padrão.

Exemplos de rotas:

```
// === ROTAS PÚBLICAS DO CARDÁPIO (SEM AUTENTICAÇÃO) ===
// Essas rotas são usadas pelo cardápio do cliente (FomeZapExact)

// ROTAS COM DETECÇÃO AUTOMÁTICA (subdomínio)
// Exemplo: familia.fomezap.com/api/cardapio/categorias
router.get("/cardapio/categorias", validarTenantPublico, AdminController.listarCategorias);
router.get("/cardapio/produtos", validarTenantPublico, AdminController.listarProdutos);
router.get("/cardapio/extras", validarTenantPublico, AdminController.listarExtras);
router.get("/cardapio/configuracoes", validarTenantPublico, ConfiguracoesController.buscarConfiguracoes);
router.get("/cardapio/horario", validarTenantPublico, ConfiguracoesController.verificarHorarioFuncionamento);

// ROTAS COM PARÂMETRO (compatibilidade com versão antiga)
// Exemplo: localhost:5173/api/demo/cardapio/categorias
router.get("/:tenantId/cardapio/categorias", validarTenantPublico, AdminController.listarCategorias);
router.get("/:tenantId/cardapio/produtos", validarTenantPublico, AdminController.listarProdutos);
router.get("/:tenantId/cardapio/extras", validarTenantPublico, AdminController.listarExtras);
router.get("/:tenantId/cardapio/configuracoes", validarTenantPublico, ConfiguracoesController.buscarConfiguracoes);
router.get("/:tenantId/cardapio/horario", validarTenantPublico, ConfiguracoesController.verificarHorarioFuncionamento);

// ROTA PARA CRIAR PEDIDO (público - usado pelo cardápio do cliente)
router.post("/:tenantId/pedidos", validarTenantPublico, PedidoController.create);

export default router;
```

[Figura 07 – Print de rotas]

2.6.3 Middleware Pattern

Middleware é padrão onde funções são executadas sequencialmente, cada uma processando requisição antes de passar para próxima (EXPRESS, 2024).

Aplicações:

a) **Autenticação:** Verificar JWT antes de permitir acesso.

b) **Logging:** Registrar todas requisições.

c) **Validação:** Verificar dados de entrada.

d) **Tratamento de erros:** Capturar e formatar erros.

2.7 TRABALHOS RELACIONADOS

2.7.1 Sistemas SaaS Multi-tenant Comerciais

Alboom: Plataforma brasileira de criação de sites profissionais com foco em fotógrafos e artistas visuais, oferece soluções multi-tenant para fotógrafos e pequenos negócios, inspirando o desenvolvimento do FomeZap pela facilidade de uso e gestão centralizada.

iFood: Maior plataforma de delivery do Brasil, opera com múltiplos restaurantes e usuários em ambiente multi-tenant, integrando cardápios, pedidos e pagamentos.

Saipos: Plataforma SaaS de gestão para restaurantes, multi-tenant, oferece controle de pedidos, estoque e integração com delivery.

Diferencial do FomeZap: Foco específico em delivery de restaurants de pequeno e médio porte, documentado para fins acadêmicos também, custo zero para iniciar, acessibilidade e design pensado em pessoas com menos intimidade com tecnologia e uma forma de attender estabelecimentos que estao ainda em fase de crescimento.

CAPÍTULO 3 - METODOLOGIA

3.1 TIPO DE PESQUISA

Este trabalho caracteriza-se como **pesquisa aplicada** com abordagem **qualitativa e quantitativa**, tendo como objetivo desenvolver solução tecnológica para problema real do mercado de delivery (GIL, 2002).

Quanto à natureza: Pesquisa aplicada, pois gera conhecimento para aplicação prática na solução de problemas específicos (SILVA; MENEZES, 2005).

Quanto aos objetivos: Pesquisa descritiva e exploratória. Descritiva ao documentar processo de desenvolvimento e características do sistema; exploratória ao investigar viabilidade técnica e econômica de SaaS multi-tenant para delivery.

Quanto aos procedimentos: Pesquisa bibliográfica (revisão de literatura sobre SaaS, multi-tenancy, tecnologias web) e estudo de caso (desenvolvimento e análise do FomeZap).

3.2 FASES DA PESQUISA

O desenvolvimento deste trabalho foi dividido em cinco fases sequenciais:

3.2.1 Fase 1: Revisão Bibliográfica/Documentação

Objetivo: Fundamentar teoricamente o trabalho, compreendendo estado da arte em SaaS, multi-tenancy, e tecnologias relacionadas.

Atividades:

- Revisão de artigos científicos sobre SaaS e multi-tenancy
- Estudo de documentações oficiais (React, Node.js, MongoDB)
- Análise de sistemas similares (Shopify, Salesforce)
- Pesquisa sobre custos de plataformas cloud

Fontes consultadas:

- Claude
- Chat GPT
- Google Scholar
- W3C
- Documentações oficiais (MDN, React Docs, MongoDB Manual)
- Blogs técnicos (Medium, StackOverflow)

3.2.2 Fase 2: Análise de Requisitos e Modelagem

Objetivo: Definir requisitos funcionais e não-funcionais, projetar arquitetura do sistema.

Atividades realizadas:

Levantamento de requisitos:

- Entrevistas informais com proprietários de restaurantes
- Análise de funcionalidades de apps de delivery existentes
- Identificação de pain points relatados por usuários

Requisitos Funcionais identificados:

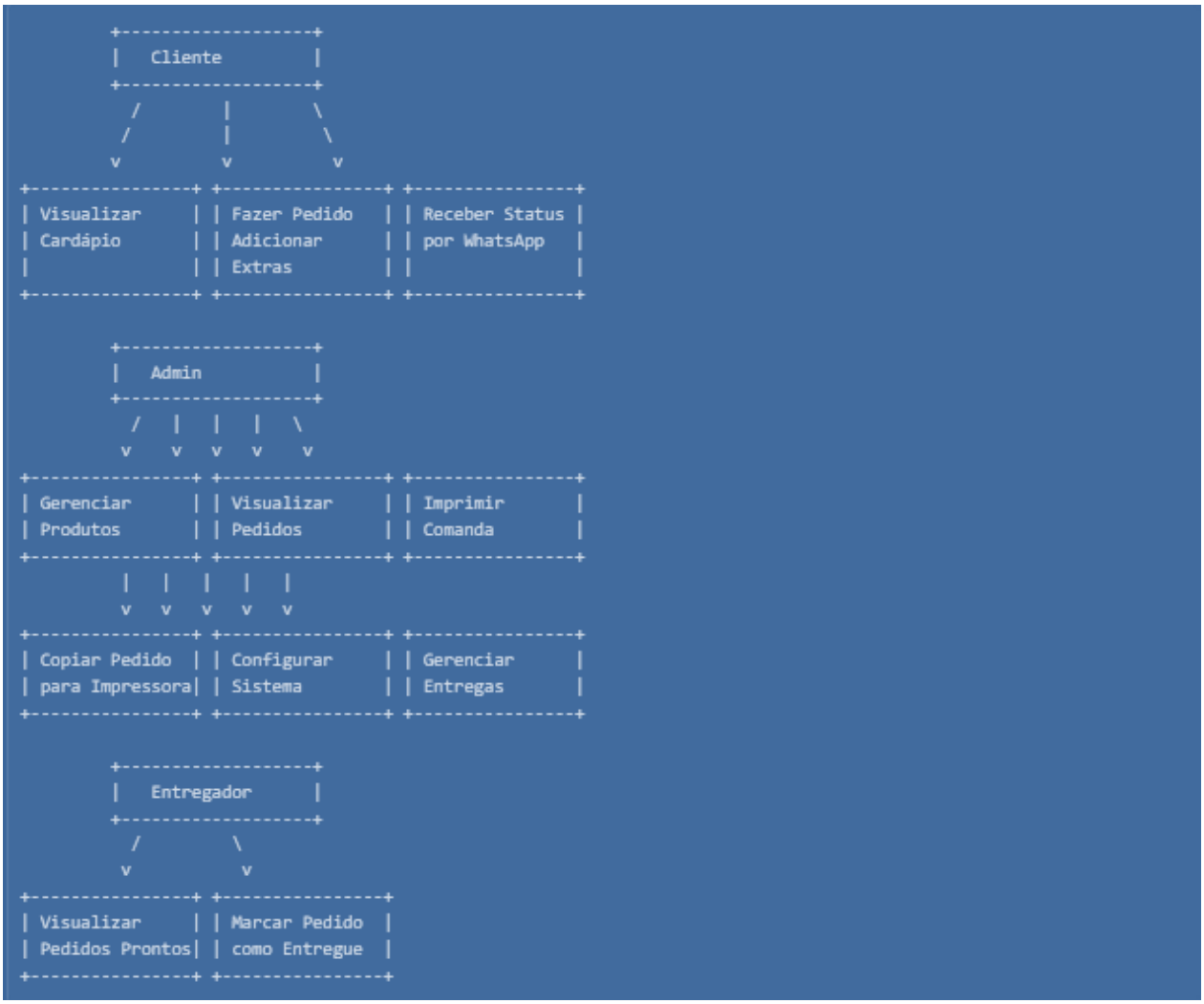
- RF01: Sistema deve permitir cadastro de múltiplos restaurantes (tenants)
- RF02: Cada tenant deve ter administrador próprio
- RF03: Admin deve gerenciar produtos, categorias, extras e pedidos
- RF04: Sistema deve gerar cardápio público acessível via web
- RF05: Clientes devem poder fazer pedidos sem cadastro
- RF06: Admin deve visualizar pedidos em tempo real
- RF07: Sistema deve calcular valores automaticamente
- RF08: Deve haver isolamento total de dados entre tenants

Requisitos Não-Funcionais:

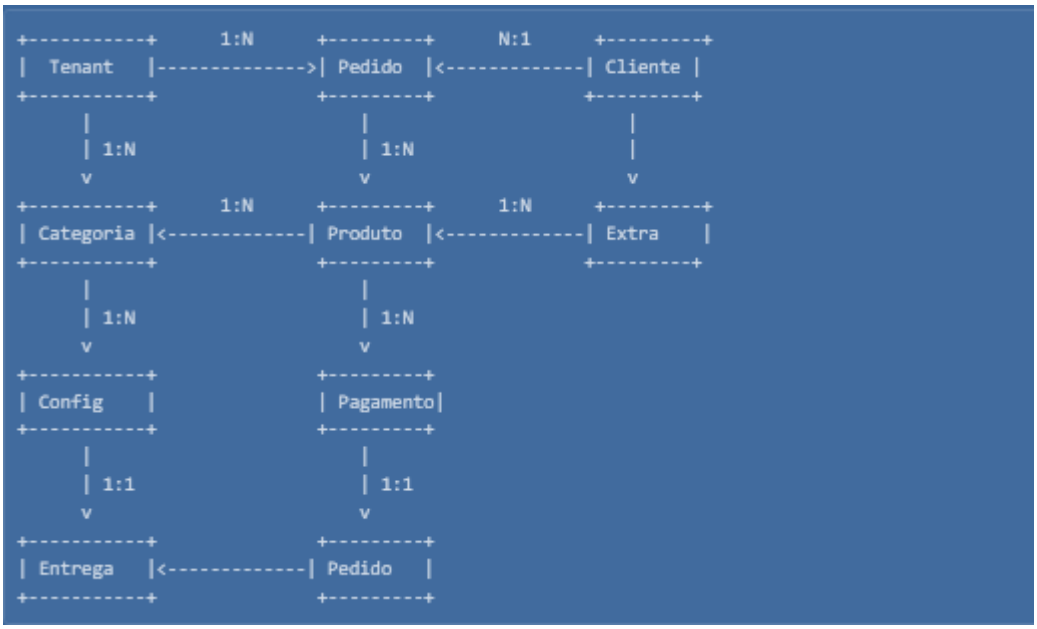
- RNF01: Performance: Tempo de resposta < 2 segundos
- RNF02: Segurança: Autenticação JWT, hashing Argon2
- RNF03: Escalabilidade: Suportar crescimento horizontal
- RNF04: Disponibilidade: 99% uptime (tier gratuito)
- RNF05: Usabilidade: Interface intuitiva, responsiva
- RNF06: Custo: Operação gratuita ou < $50/mês

Modelagem:

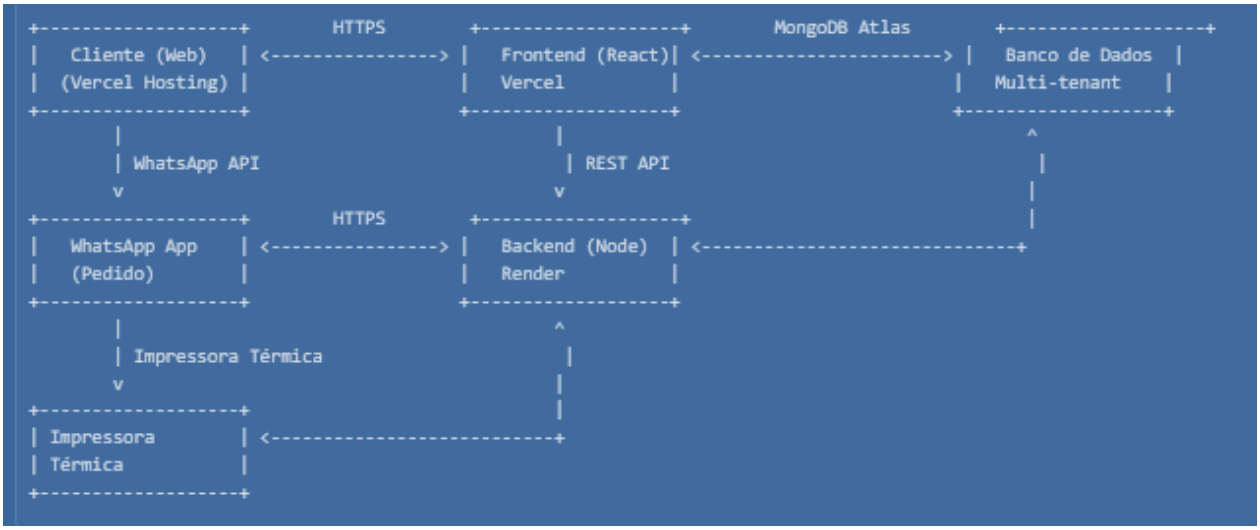
- Diagramas de casos de uso
- Modelo Entidade-Relacionamento para NoSQL
- Diagrama de arquitetura (frontend/backend/database)
- Wireframes das telas principais



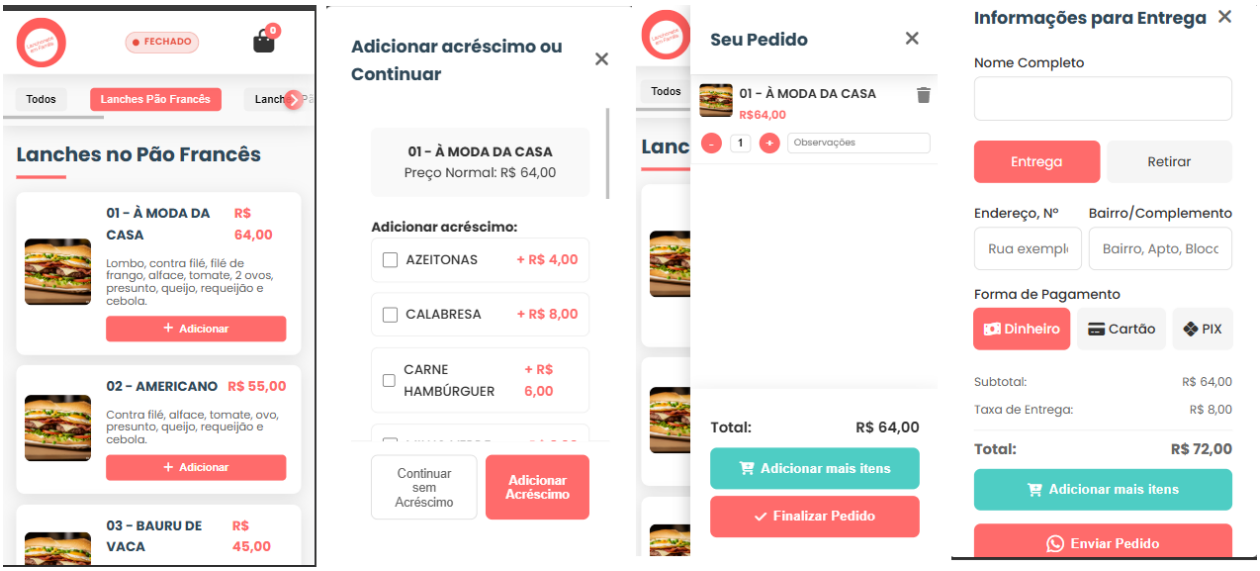
[Figura 1 - Diagrama de Casos de Uso]



[Figura 2 - Modelo de Dados]



[Figura 3 - Arquitetura do Sistema]



[Figura 4 – Telas rotótipo]

3.2.3 Fase 3: Desenvolvimento

Objetivo: Implementar sistema completo seguindo arquitetura projetada.

Abordagem: Desenvolvimento ágil iterativo, com sprints de 1 semana.

Sprints executados:

Sprint 1-2: Setup inicial e autenticação

- Configuração de repositório Git
- Setup projetos React (Vite) e Node.js (Express)
- Implementação de autenticação JWT
- Cadastro e login de usuários

Sprint 3-4: Multi-tenancy e gestão de tenants

- Implementação de isolamento por tenantId
- Painel super admin para criar restaurantes
- Middleware de validação de tenant
- Criação automática de admin + produtos padrão

Sprint 5-6: Gestão de produtos e categorias

- CRUD completo de produtos
- CRUD de categorias e extras
- Upload de imagens
- Validações e tratamento de erros

Sprint 7: Cardápio público e carrinho

- Interface pública responsiva
- Adição de produtos ao carrinho
- Cálculo de valores
- Página de checkout

Sprint 8: Sistema de pedidos e integrações

- Criação de pedidos
- Listagem e filtros no admin
- Atualização de status
- Testes integrados

Ferramentas utilizadas:

- **Editor:** Visual Studio Code
- **Versionamento:** Git + GitHub
- **Terminal:** PowerShell
- **Testes:** Navegadores (Chrome DevTools), Postman
- **IA:** Claude e ChatGPT
- **Corel Draw:** Logos
- **Documentação:** Markdown, diagramas Draw.io, Figma e Canva
- **Docker:** Conteinerização e Kind (Kubernetes in Docker para entrega contínua)

3.2.4 Fase 4: Deploy e Validação

Objetivo: Realizar deploy em produção e validar funcionamento.

Atividades:

Configuração de plataformas cloud:

- MongoDB Atlas: Cluster M0 gratuito, região us-east-1
- Render.com: Web Service para backend Node.js
- Vercel: Deploy automático do frontend React
- Configuração de variáveis de ambiente

- Setup de CI/CD via GitHub

Testes em produção:

- Criação de tenant de teste
- Cadastro de produtos reais
- Simulação de pedidos
- Teste de performance (tempo de resposta)
- Teste de segurança (tentativas de acesso indevido)
- Teste de escalabilidade (múltiplos acessos simultâneos)

Validação de custos:

- Monitoramento de uso de recursos
- Confirmação de limites do plano gratuito
- Projeção de custos para crescimento

3.2.5 Fase 5: Documentação e Análise

Objetivo: Documentar sistema, analisar resultados, e escrever o PI.

Atividades:

- Documentação técnica detalhada (README, comentários)
- Captura de screenshots
- Análise de métricas (performance, custos)
- Comparação com soluções existentes
- Escrita do documento de PI
- Preparação de apresentação

3.3 FERRAMENTAS E TECNOLOGIAS

3.3.1 Linguagens de Programação

JavaScript ES6+: Linguagem principal tanto no frontend (React) quanto backend (Node.js). Escolhida por:

- Uniformidade (fullstack JavaScript)
- Ecossistema rico (NPM com 2M+ pacotes)
- Performance (V8 engine)
- Comunidade ativa

- Requisito da matéria de Laboratório de desenvolvimento web

HTML5 e CSS3: Marcação e estilização das páginas web, com uso extensivo de classes utilitárias Tailwind CSS.

3.3.2 Frameworks e Bibliotecas

Frontend:

React 19.1.1: Biblioteca para construção de UI. Escolhida por:

- Componentização e reutilização
- Virtual DOM para performance
- Ecossistema maduro
- Curva de aprendizado moderada

Vite 7.1.2: Build tool moderna, escolhida por:

- Velocidade (Hot Module Replacement instantâneo)
- Simplicidade de configuração
- Suporte nativo a ESM
- Build otimizado para produção

React Router 7.8.2: Roteamento client-side, permitindo navegação SPA sem recarregar página.

Tailwind CSS 3.4: Framework CSS utility-first, escolhido por:

- Desenvolvimento rápido
- Consistência visual
- Responsividade built-in
- Bundle size pequeno (apenas classes usadas)

Axios 1.11.0: Cliente HTTP para comunicação com backend, escolhido por:

- Interceptors (para adicionar token JWT automaticamente)
- Tratamento de erros consistente
- Suporte a promises

Backend:

Node.js 20.18.0: Runtime JavaScript server-side. Escolhido por:

- Performance (V8 engine, event-driven)
- Escalabilidade (non-blocking I/O)
- Mesma linguagem do frontend

Express 5.1.0: Framework web minimalista. Escolhido por:

- Simplicidade e flexibilidade
- Middleware architecture
- Ampla adoção no mercado

Mongoose 8.18.0: ODM (Object Data Modeling) para MongoDB. Oferece:

- Schema definitions para MongoDB
- Validações built-in
- Middleware hooks
- Query builders

JSON Web Token (jsonwebtoken 9.0.2): Implementação JWT para autenticação stateless.

Argon2 0.44.0: Algoritmo de hashing de senhas, escolhido por:

- Estado-da-arte em segurança
- Memory-hard (resistente a GPUs)
- Vencedor da Password Hashing Competition

CORS 2.8.5: Middleware para configuração de Cross-Origin Resource Sharing.

3.3.3 Banco de Dados

MongoDB Atlas (Cloud): Banco de dados NoSQL orientado a documentos. Escolhido por:

Adequação ao multi-tenancy:

- Schema flexível permite customizações por tenant
- Queries com discriminador (tenantId) são eficientes
- Índices compostos otimizam isolamento

Escalabilidade:

- Sharding horizontal nativo
- Replicação automática

Developer Experience:

- JSON nativo (integração natural com JavaScript)
- Agregações poderosas
- Atlas oferece plano gratuito (M0 - 512MB)

Alternativas consideradas e descartadas:

- **PostgreSQL:** Mais rígido, exigiria migrations, menos natural com JavaScript
- **MySQL:** Similar ao PostgreSQL em limitações
- **Firebase:** Vendor lock-in, menos controle

3.3.4 Plataformas de Deploy

Vercel: Deploy do frontend React. Escolhida por:

- Deploy automático via GitHub (CI/CD nativo)
- CDN global (Edge Network)
- Plano gratuito generoso (100GB bandwidth)
- Configuração zero (detecta Vite automaticamente)

Render.com: Deploy do backend Node.js. Escolhida por:

- Plano gratuito (750h/mês)
- Deploy automático via GitHub
- Suporte nativo a Node.js
- Logs em tempo real

MongoDB Atlas: Database as a Service. Escolhido por:

- Plano gratuito (M0 Cluster - 512MB)
- Gerenciamento automático (backups, updates)
- Segurança (encryption at rest/in transit)
- Network access control

Alternativas consideradas:

- **Heroku:** Descartado (plano gratuito descontinuado em 2022)
- **AWS/GCP/Azure:** Complexidade excessiva para escopo do trabalho
- **Railway:** Similar ao Render, mas Render oferece mais horas gratuitas

3.3.5 Ferramentas de Desenvolvimento

Visual Studio Code: IDE utilizada, com extensões:

- ESLint: Linting de JavaScript
- Prettier: Formatação automática de código
- MongoDB for VS Code: Visualização de banco
- GitHub Copilot: Assistência de código com IA

Git + GitHub: Versionamento de código e CI/CD.

Postman: Testes de API REST durante desenvolvimento.

Chrome DevTools: Debug de frontend, network analysis, performance profiling.

PowerShell: Terminal para comandos Git, npm, node.

3.4 MÉTRICAS DE AVALIAÇÃO

Para validar o sistema, foram definidas métricas quantitativas e qualitativas:

3.4.1 Métricas de Performance

Tempo de resposta das APIs:

- Alvo: < 2 segundos para 95% das requisições
- Medição: Chrome DevTools Network tab

Tempo de carregamento de páginas:

- Alvo: < 3 segundos para First Contentful Paint
- Medição: Lighthouse (Chrome)

Bundle size do frontend:

- Alvo: < 500KB gzipped
- Medição: Vite build output

3.4.2 Métricas de Custo

Custo operacional mensal:

- Plano gratuito: $0/mês
- Projeção para 50 tenants: < $50/mês

Custo por tenant:

- Alvo: < $1/tenant/mês em escala

3.4.3 Métricas de Funcionalidade

Cobertura de requisitos:

- Alvo: 100% dos requisitos funcionais implementados
- Medição: Checklist de requisitos

Taxa de sucesso em testes:

- Alvo: 100% dos casos de teste passando
- Medição: Testes manuais funcionais

3.4.4 Métricas de Segurança

Isolamento de dados:

- Alvo: 0 vazamentos entre tenants
- Medição: Testes de tentativa de acesso cruzado

Força da autenticação:

- JWT com expiração (7 dias)
- Argon2 para senhas (verificado)

3.5 LIMITAÇÕES METODOLÓGICAS

Amostra limitada: Sistema validado com dados de teste. Falta validação com usuários reais em produção.

Testes de carga: Não foram realizados testes de estresse com milhares de usuários simultâneos devido a limitações de infraestrutura.

Comparação direta: Não foi possível comparar side-by-side com plataformas proprietárias (iFood, Rappi) por falta de acesso aos seus sistemas.

Tempo de desenvolvimento: 17 semanas pode ser considerado curto para sistema completo, algumas funcionalidades avançadas foram deixadas para trabalhos futuros.

CAPÍTULO 4 - DESENVOLVIMENTO DO SISTEMA

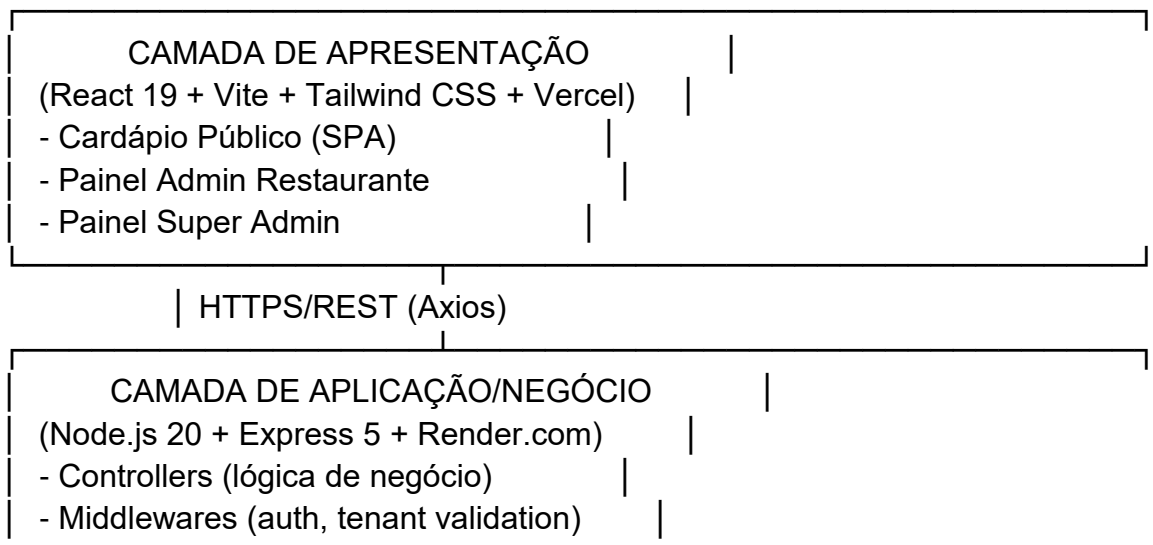
CAPÍTULO 4 - DESENVOLVIMENTO DO SISTEMA

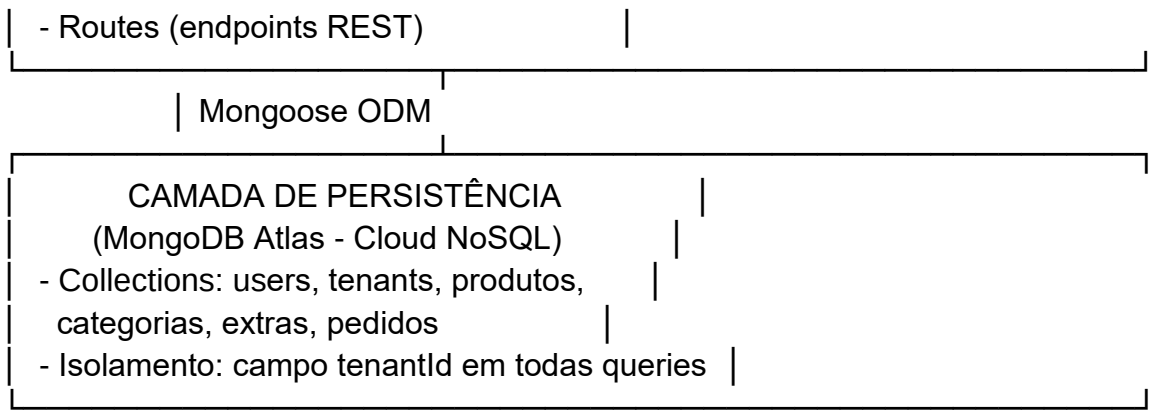
4.1 ARQUITETURA DO SISTEMA

4.1.1 Visão Geral

O FomeZap foi desenvolvido seguindo arquitetura de três camadas (frontend, backend, banco de dados) com comunicação via APIs REST. A arquitetura foi projetada para escalabilidade horizontal e isolamento multi-tenant.

[INCLUIR: Figura 4 - Arquitetura Geral do Sistema]





4.1.2 Padrões Arquiteturais Aplicados

MVC (Model-View-Controller):

- **Model:** Schemas Mongoose (Tenant.js, Produto.js, Pedido.js)
- **View:** Componentes React (Cardapio.jsx, AdminPanel.jsx)
- **Controller:** Controllers Express (ProdutoController.js, PedidoController.js)

RESTful API:

- Recursos mapeados em rotas HTTP
- Métodos semânticos (GET, POST, PUT, DELETE)
- Status codes apropriados (200, 201, 400, 401, 404, 500)

Middleware Pipeline:

- Requisições passam por cadeia de validações
- Ordem: CORS → JSON Parser → Auth → Tenant Validation → Controller

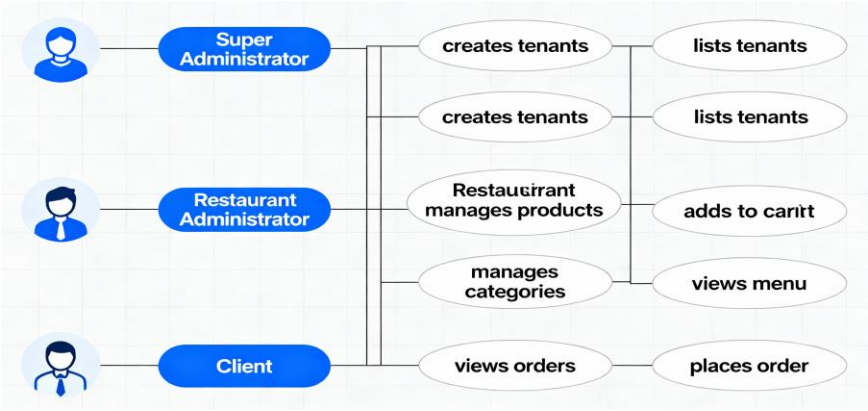
4.2 MODELO DE DADOS

4.2.1 Collections MongoDB

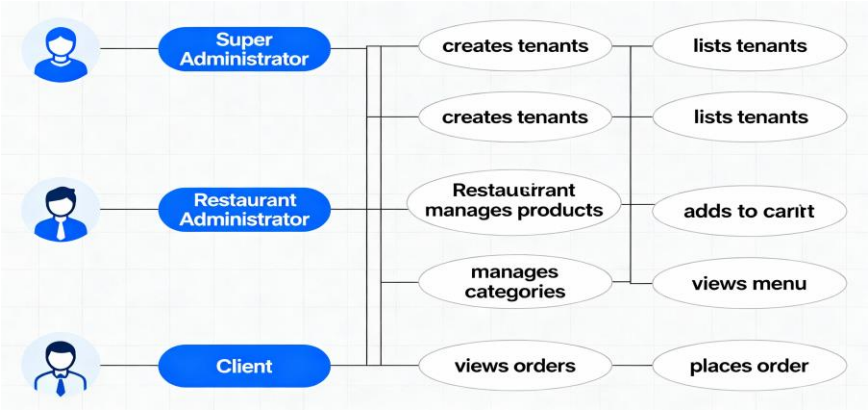
O banco de dados foi modelado com 6 collections principais:

[INCLUIR: Figura 7 - Modelo de Dados (Collections)]

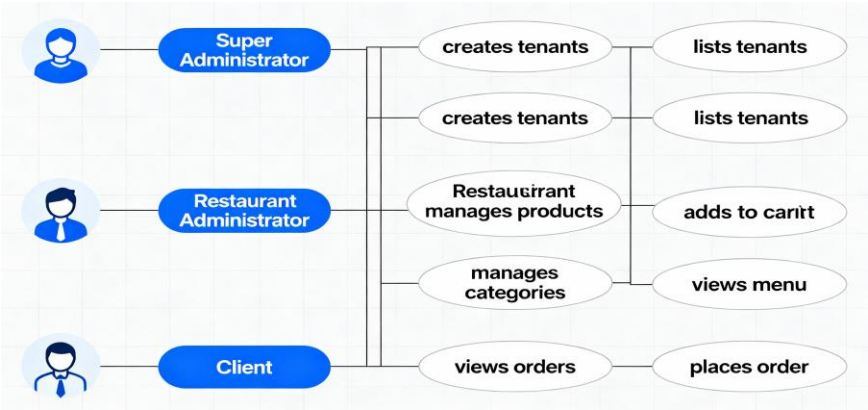
Collection: users



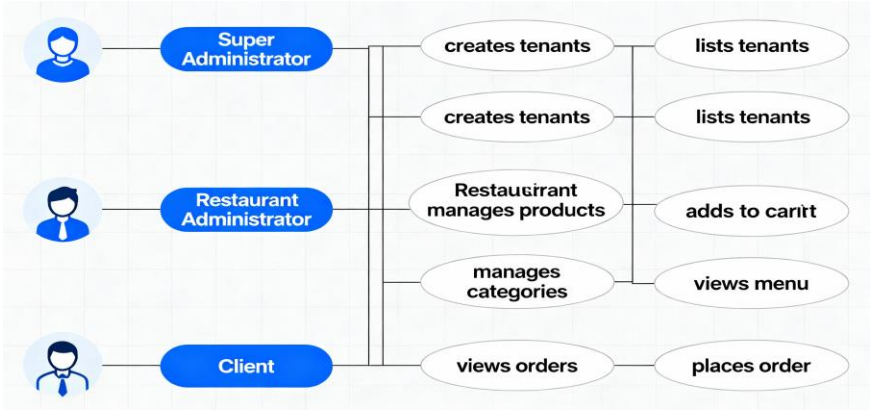
Collection: tenants



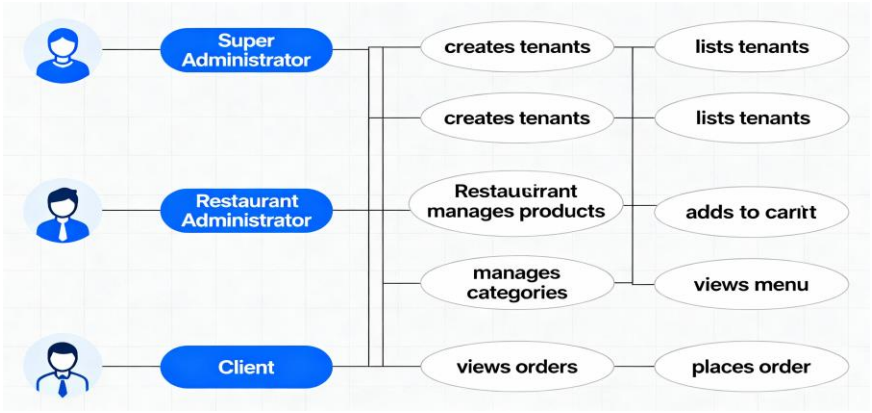
Collection: produtos



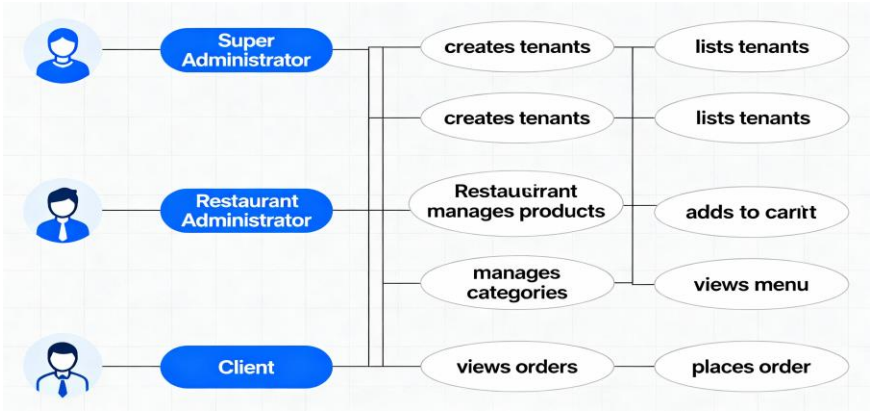
Collection: categorias



Collection: extras



Collection: pedidos



4.2.2 Estratégia de Isolamento Multi-tenant

Implementação: Nível 4 (Shared Database, Shared Schema)

Todas as collections compartilham mesma instância MongoDB, com isolamento lógico via campo tenantId:

```
// Exemplo de query isolada
async listarProdutos(tenantId) {
  return await Produto.find({
    tenantId: tenantId,  // ISOLAMENTO
    ativo: true
  });
}
```

[PARTE DO CÓDIGO: Backend/Controllers/ProdutoController.js]

Índices para Performance:

```
// Índices compostos otimizam queries multi-tenant
ProdutoSchema.index({ tenantId: 1, categoria: 1 });
ProdutoSchema.index({ tenantId: 1, ativo: 1 });
PedidoSchema.index({ tenantId: 1, criadoEm: -1 });
```

[PARTE DO CÓDIGO: Backend/Models/ProdutoModels.js]

4.2.3 Resolução de Slug → TenantId

Sistema aceita tanto tenantId (ObjectId) quanto slug (human-readable) nas URLs:

```
// Middleware detecta e resolve
async resolverTenantId(param) {
  if (param.length === 24 && /^[0-9a-fA-F]+$/.test(param)) {
    return param; // Já é tenantId
  }

  // É slug, buscar tenant
  const tenant = await Tenant.findOne({
    $or: [{ tenantId: param }, { slug: param }]
  });
```

```
  return tenant?.tenantId;
}
```

[PARTE DO CÓDIGO: Backend/Controllers/AdminController.js - resolverTenantId]

4.3 IMPLEMENTAÇÃO DO BACKEND

4.3.1 Estrutura de Pastas

```
Backend/
├── index.js                # Entry point, configuração Express
├── package.json            # Dependências
├── .env                    # Variáveis de ambiente (local)
│
├── Controllers/            # Lógica de negócio
│   ├── AuthController.js       # Login, registro
│   ├── SuperAdminController.js # Gestão de tenants
│   ├── AdminController.js      # CRUD produtos/categorias (tenant-specific)
│   ├── PedidoController.js     # Gestão de pedidos
│   ├── ConfiguracoesController.js # Configurações do tenant
│   └── TenantController.js     # Info pública do tenant
│
├── Middlewares/            # Funções intermediárias
```

```
│   ├── auth.js              # Validação JWT, verificação roles
│   ├── detectarTenant.js    # Detecção tenant via query/subdomain
│   └── validarTenantPublico.js # Validação rotas públicas
│
├── Models/                  # Schemas Mongoose
│   ├── User.js
│   ├── Tenant.js
│   ├── Produto.js
│   ├── Categoria.js
│   ├── Extra.js
│   └── Pedido.js
│
├── Routes/                  # Definição de rotas
│   ├── authRoutes.js        # /api/auth/*
│   ├── superAdminRoutes.js  # /api/super-admin/*
│   ├── adminRoutes.js       # /api/admin/:tenantId/*
│   ├── publicRoutes.js      # /api/:tenantId/cardapio/*
│   └── routes.js            # Agregador de rotas
│
└── db/
    └── conn.js              # Conexão MongoDB
```

4.3.2 Autenticação JWT

Geração de Token (Login):

```
// AuthController.js - login
async login(req, res) {
  const { email, senha } = req.body;

  // Buscar usuário
  const user = await User.findOne({ email });
  if (!user) {
    return res.status(401).json({ erro: "Credenciais inválidas" });
  }

  // Verificar senha (Argon2)
  const senhaValida = await argon2.verify(user.senha, senha);
  if (!senhaValida) {
    return res.status(401).json({ erro: "Credenciais inválidas" });
```

```
  }

  // Gerar JWT
  const token = jwt.sign(
    {
      userId: user._id,
      tenantId: user.tenantId,
      role: user.role
    },
    process.env.JWT_SECRET,
    { expiresIn: "7d" }
  );

  res.json({
    token,
    user: {
      id: user._id,
      nome: user.nome,
      role: user.role,
      tenantId: user.tenantId
    }
  });
}
```

[PARTE DO CÓDIGO: Backend/Controllers/AuthController.js - login completo]

Validação de Token (Middleware):

```
// Middlewares/auth.js
function verificarToken(req, res, next) {
  const authHeader = req.headers.authorization;

  if (!authHeader) {
    return res.status(401).json({ erro: "Token não fornecido" });
  }

  const token = authHeader.split(' ')[1]; // "Bearer <token>"

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
```

```
    req.userId = decoded.userId;
    req.userRole = decoded.role;
    req.userTenantId = decoded.tenantId;
    next();
  } catch (erro) {
    return res.status(401).json({ erro: "Token inválido" });
  }
}
```

[PARTE DO CÓDIGO: Backend/Middlewares/auth.js - verificarToken]

4.3.3 Middleware de Tenant Validation

Garantir que usuário só acesse seu próprio tenant:

```
// Middlewares/auth.js
async function verificarTenantAdmin(req, res, next) {
  const urlTenantParam = req.params.tenantId;
  const userTenantId = req.userTenantId;

  // Super admin pode acessar qualquer tenant
  if (req.userRole === 'super_admin') {
    return next();
  }

  // Resolver tenantId (pode ser slug)
  const tenant = await Tenant.findOne({
    $or: [{ tenantId: urlTenantParam }, { slug: urlTenantParam }]
  });

  if (!tenant) {
    return res.status(404).json({ erro: "Tenant não encontrado" });
  }

  // Comparar tenantIds (convertidos para string)
  if (String(userTenantId) !== String(tenant.tenantId)) {
    return res.status(403).json({
      erro: "Acesso negado ao tenant"
    });
  }
```

```
  req.tenantId = tenant.tenantId;
  next();
}
```

[PARTE DO CÓDIGO: Backend/Middlewares/auth.js - verificarTenantAdmin]

4.3.4 Rotas e Controllers

Exemplo: CRUD de Produtos

```
// Routes/adminRoutes.js
router.get('/admin/:tenantId/produtos',
  verificarToken,
  verificarTenantAdmin,
  AdminController.listarProdutos
);

router.post('/admin/:tenantId/produtos',
  verificarToken,
  verificarTenantAdmin,
  AdminController.criarProduto
);
```

[PARTE DO CÓDIGO: Backend/Routes/adminRoutes.js]

```
// Controllers/AdminController.js
async listarProdutos(req, res) {
  try {
    const tenantId = await resolverTenantId(req.params.tenantId);

    const produtos = await Produto.find({
      tenantId: tenantId,
      // Sem filtro de "ativo" - admin vê tudo
    }).sort({ criadoEm: -1 });

    res.json(produtos);
  } catch (erro) {
    res.status(500).json({ erro: erro.message });
  }
```

```
}

async criarProduto(req, res) {
  try {
    const tenantId = await resolverTenantId(req.params.tenantId);

    const produto = new Produto({
      tenantId: tenantId, // ISOLAMENTO
      ...req.body
    });

    await produto.save();
    res.status(201).json(produto);
  } catch (erro) {
    res.status(400).json({ erro: erro.message });
  }
}
```

[PARTE DO CÓDIGO: Backend/Controllers/AdminController.js - listarProdutos, criarProduto]

4.4 IMPLEMENTAÇÃO DO FRONTEND

4.4.1 Estrutura de Pastas

```
Frontend/
├── index.html              # Ponto de entrada HTML
├── package.json            # Dependências
├── vite.config.js          # Configuração Vite
├── tailwind.config.cjs     # Configuração Tailwind
│
├── src/
│   ├── main.jsx            # Entry point React
│   ├── App.jsx             # Roteamento principal
│   ├── index.css           # Estilos globais + Tailwind
│   │
│   ├── api/
│   │   └── api.js          # Axios instance configurado
│   │
```

```
│   ├── contexts/              # Context API (estado global)
│   │   ├── AuthContext.jsx    # Autenticação (user, token)
│   │   ├── TenantContext.jsx  # Info do tenant atual
│   │   └── CarrinhoContext.jsx # Carrinho de compras
│   │
│   ├── pages/                 # Páginas principais
│   │   ├── Login.jsx
│   │   ├── FomeZapExact.jsx   # Cardápio público
│   │   ├── Checkout.jsx
│   │   │
│   │   ├── Admin/             # Painel admin restaurante
│   │   │   ├── Dashboard.jsx
│   │   │   ├── Produtos.jsx
│   │   │   ├── Categorias.jsx
│   │   │   ├── Extras.jsx
│   │   │   ├── Pedidos.jsx
│   │   │   └── Configuracoes.jsx
│   │   │
│   │   └── SuperAdmin/        # Painel super admin
│   │       ├── Dashboard.jsx
│   │       └── Tenants.jsx
│   │
│   └── components/            # Componentes reutilizáveis
│       ├── Navbar.jsx
│       ├── ProdutoCard.jsx
│       ├── CarrinhoItem.jsx
│       ├── LoadingScreen.jsx
│       └── ErrorScreen.jsx
│
└── public/
    └── (assets estáticos)
```

4.4.2 Context API para Estado Global

AuthContext - Gerenciamento de Autenticação:

```
// contexts/AuthContext.jsx
export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);
  const [token, setToken] = useState(localStorage.getItem('token'));
```

```
  // Login
  const login = async (email, senha) => {
    const { data } = await api.post('/auth/login', { email, senha });

    setToken(data.token);
    setUser(data.user);
    localStorage.setItem('token', data.token);
    localStorage.setItem('user', JSON.stringify(data.user));
  };

  // Logout
  const logout = () => {
    setToken(null);
    setUser(null);
    localStorage.removeItem('token');
    localStorage.removeItem('user');
  };

  // Buscar user ao carregar app (se token existe)
  useEffect(() => {
    if (token && !user) {
      api.get('/auth/me')
        .then(({ data }) => {
          setUser(data);
          localStorage.setItem('user', JSON.stringify(data));
        })
        .catch(() => logout());
    }
  }, [token]);

  return (
    <AuthContext.Provider value={{ user, token, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
};
```

[PARTE DO CÓDIGO: Frontend/src/contexts/AuthContext.jsx]

CarrinhoContext - Carrinho de Compras:

```
// contexts/CarrinhoContext.jsx
export const CarrinhoProvider = ({ children }) => {
  const [itens, setItens] = useState([]);

  const adicionarItem = (produto, quantidade, extras) => {
    setItens(prev => [...prev, {
      ...produto,
      quantidade,
      extras,
      subtotal: (produto.preco + extras.reduce((sum, e) => sum + e.preco, 0)) * quantidade
    }]);
  };

  const removerItem = (index) => {
    setItens(prev => prev.filter((_, i) => i !== index));
  };

  const calcularTotal = () => {
    return itens.reduce((total, item) => total + item.subtotal, 0);
  };

  const limparCarrinho = () => setItens([]);

  return (
    <CarrinhoContext.Provider value={{
      itens,
      adicionarItem,
      removerItem,
      calcularTotal,
      limparCarrinho
    }}>
      {children}
    </CarrinhoContext.Provider>
  );
};
```

[PARTE DO CÓDIGO: Frontend/src/contexts/CarrinhoContext.jsx]

4.4.3 Configuração Axios

Instância global com interceptors:

```
// api/api.js
const api = axios.create({
  baseURL: import.meta.env.MODE === 'production'
    ? 'https://fomezap-api.onrender.com'
    : 'http://localhost:5000',
  timeout: 30000,
});

// Interceptor: adicionar token automaticamente
api.interceptors.request.use(config => {
  const token = localStorage.getItem('token');
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});

// Interceptor: tratar erros globalmente
api.interceptors.response.use(
  response => response,
  error => {
    if (error.response?.status === 401) {
      // Token inválido/expirado - fazer logout
      localStorage.removeItem('token');
      localStorage.removeItem('user');
      window.location.href = '/login';
    }
    return Promise.reject(error);
  }
);
```

[PARTE DO CÓDIGO: Frontend/src/api/api.js]

4.4.4 Roteamento e Proteção de Rotas

```
// App.jsx
function App() {
  return (
    <BrowserRouter>
      <AuthProvider>
        <TenantProvider>
          <CarrinhoProvider>
            <Routes>
              {/* Rotas públicas */}
              <Route path="/" element={<FomeZapExact />} />
              <Route path="/checkout" element={<Checkout />} />
              <Route path="/login" element={<Login />} />

              {/* Rotas protegidas - Admin */}
              <Route path="/admin/*" element={
                <ProtectedRoute allowedRoles={['tenant_admin']}>
                  <AdminLayout />
                </ProtectedRoute>
              }>
                <Route path="dashboard" element={<Dashboard />} />
                <Route path="produtos" element={<Produtos />} />
                <Route path="pedidos" element={<Pedidos />} />
                {/* ... */}
              </Route>

              {/* Rotas protegidas - Super Admin */}
              <Route path="/super-admin/*" element={
                <ProtectedRoute allowedRoles={['super_admin']}>
                  <SuperAdminLayout />
                </ProtectedRoute>
              }>
                <Route path="tenants" element={<Tenants />} />
                {/* ... */}
              </Route>
            </Routes>
          </CarrinhoProvider>
        </TenantProvider>
      </AuthProvider>
    </BrowserRouter>
```

```
  );
}

// Componente ProtectedRoute
function ProtectedRoute({ children, allowedRoles }) {
  const { user } = useAuth();

  if (!user) {
    return <Navigate to="/login" />;
  }

  if (allowedRoles && !allowedRoles.includes(user.role)) {
    return <Navigate to="/" />;
  }

  return children;
}
```

[PARTE DO CÓDIGO: Frontend/src/App.jsx]

4.5 IMPLEMENTAÇÃO DE FUNCIONALIDADES CRÍTICAS

4.5.1 Criação Automática de Tenant com Dados Padrão

Quando super admin cria tenant, sistema automaticamente:

1. Cria usuário admin do tenant
2. Cria 6 produtos padrão
3. Cria 3 categorias padrão
4. Configura slug único

```
// SuperAdminController.js - criarTenant
async criarTenant(req, res) {
  const { nome, email, slug } = req.body;

  // 1. Criar tenant
  const tenant = new Tenant({
    tenantId: new ObjectId().toString(),
    nome,
```

```
  email,
  slug: slug.toLowerCase().replace(/\s+/g, '-'),
  configuracoes: { /* defaults */ }
});
await tenant.save();

// 2. Gerar senha temporária
const senhaTemporaria = gerarSenhaAleatoria();
const senhaHash = await argon2.hash(senhaTemporaria);

// 3. Criar usuário admin
const admin = new User({
  email,
  senha: senhaHash,
  nome: nome,
  role: 'tenant_admin',
  tenantId: tenant.tenantId
});
await admin.save();

// 4. Criar categorias padrão
const categorias = ['Lanches', 'Bebidas', 'Sobremesas'];
for (let i = 0; i < categorias.length; i++) {
  await Categoria.create({
    tenantId: tenant.tenantId,
    nome: categorias[i],
    ordem: i,
    ativa: true
  });
}

// 5. Criar produtos padrão
const produtos = [
  { nome: 'X-Burger', categoria: 'Lanches', preco: 15.90 },
  { nome: 'X-Bacon', categoria: 'Lanches', preco: 18.90 },
  { nome: 'X-Salada', categoria: 'Lanches', preco: 17.90 },
  { nome: 'Refrigerante', categoria: 'Bebidas', preco: 5.00 },
  { nome: 'Suco Natural', categoria: 'Bebidas', preco: 7.00 },
  { nome: 'Pudim', categoria: 'Sobremesas', preco: 8.00 }
];
```

```
  for (const prod of produtos) {
    await Produto.create({
      tenantId: tenant.tenantId,
      ...prod,
      ativo: true
    });
  }

  res.status(201).json({
    tenant,
    admin: { email: admin.email },
    senhaTemporaria  // Retornar para super admin
  });
}
```

[PARTE DO CÓDIGO: Backend/Controllers/SuperAdminController.js - criarTenant completo]

4.5.2 Fluxo de Pedido (Cliente → Admin)

Cliente (Cardápio Público):

1. Cliente acessa /?tenant=nome-restaurante
2. Adiciona produtos ao carrinho (CarrinhoContext)
3. Vai para checkout, preenche dados
4. Finaliza pedido:

```
// Checkout.jsx
const finalizarPedido = async () => {
  const { tenantId } = useTenant();
  const { itens, calcularTotal, limparCarrinho } = useCarrinho();

  const pedido = {
    cliente: {
      nome: clienteNome,
      telefone: clienteTelefone,
      endereco: { /* ... */ }
    },
    itens: itens.map(item => ({
      produtoId: item._id,
      nome: item.nome,
```

```
      preco: item.preco,
      quantidade: item.quantidade,
      extras: item.extras
    })),
    subtotal: calcularTotal(),
    taxaEntrega: 5.00,
    total: calcularTotal() + 5.00,
    observacoes: observacoes
  };

  await api.post(`/api/${tenantId}/pedidos`, pedido);

  limparCarrinho();
  navigate('/pedido-confirmado');
};
```

[PARTE DO CÓDIGO: Frontend/src/pages/Checkout.jsx - finalizarPedido]

Backend (Criação de Pedido):

```
// PedidoController.js
async create(req, res) {
  try {
    const tenantParam = req.params.tenantId;

    // Resolver slug → tenantId se necessário
    const tenantId = await resolverTenantId(tenantParam);

    // Validar produtos pertencem ao tenant
    const produtoIds = req.body.itens.map(i => i.produtoId);
    const produtos = await Produto.find({
      _id: { $in: produtoIds },
      tenantId: tenantId
    });

    if (produtos.length !== produtoIds.length) {
      return res.status(400).json({
        erro: "Produtos inválidos"
      });
    }
```

```
    // Gerar número do pedido (incremental por tenant)
    const ultimoPedido = await Pedido.findOne({ tenantId })
      .sort({ numeroPedido: -1 });
    const numeroPedido = (ultimoPedido?.numeroPedido || 0) + 1;

    // Criar pedido
    const pedido = new Pedido({
      tenantId: tenantId,
      numeroPedido,
      ...req.body,
      status: 'recebido'
    });

    await pedido.save();

    res.status(201).json(pedido);
  } catch (erro) {
    res.status(500).json({ erro: erro.message });
  }
}
```

[PARTE DO CÓDIGO: Backend/Controllers/PedidoController.js - create]

Admin (Visualização de Pedidos):

```
// Admin/Pedidos.jsx
const Pedidos = () => {
  const { user } = useAuth();
  const [pedidos, setPedidos] = useState([]);

  useEffect(() => {
    api.get(`/api/admin/${user.tenantId}/pedidos`)
      .then(({ data }) => setPedidos(data));
  }, []);

  const atualizarStatus = async (pedidoId, novoStatus) => {
    await api.put(`/api/admin/${user.tenantId}/pedidos/${pedidoId}`, {
      status: novoStatus
    });
```

```
    // Recarregar lista
  };

  return (
    <div>
      {pedidos.map(pedido => (
        <PedidoCard
          key={pedido._id}
          pedido={pedido}
          onAtualizarStatus={atualizarStatus}
        />
      ))}
    </div>
  );
};
```

[PARTE DO CÓDIGO: Frontend/src/pages/Admin/Pedidos.jsx]

4.6 DEPLOY E CI/CD

4.6.1 Configuração MongoDB Atlas

1. Criação de cluster M0 (gratuito) em us-east-1
2. Criação de usuário fomezap_user com permissões readWrite
3. Liberação de acesso via Network Access: 0.0.0.0/0 (necessário para Render.com)
4. Connection string:mongodb+srv://fomezap_user:SENHA@cluster.xxxxx.mongodb.net/fomezap?retryWrites=true&w=majority

4.6.2 Deploy Backend (Render.com)

Configuração:

- **Root Directory:** Backend
- **Build Command:** npm install
- **Start Command:** npm start (executa node index.js)
- **Environment Variables:**MONGODB_URI=mongodb+srv://... JWT_SECRET=<64-byte hex string>

```
NODE_ENV=production
PORT=5000
CORS_ORIGINS=https://fomezap.vercel.app
```

CI/CD Automático:

- Cada push no branch main → Render detecta mudança
- Render executa npm install + npm start
- Deploy completo em ~3 minutos
- URL gerada: https://fomezap-api.onrender.com

4.6.3 Deploy Frontend (Vercel)

Configuração:

- **Framework Preset:** Vite
- **Root Directory:** Frontend
- **Build Command:** npm run build
- **Output Directory:** dist

vercel.json (Configuração SPA):

```
{
  "rewrites": [
    { "source": "/(.*)", "destination": "/index.html" }
  ],
  "headers": [{
    "source": "/assets/(.*)",
    "headers": [{
      "key": "Cache-Control",
      "value": "public, max-age=31536000, immutable"
    }]
  }]
}
```

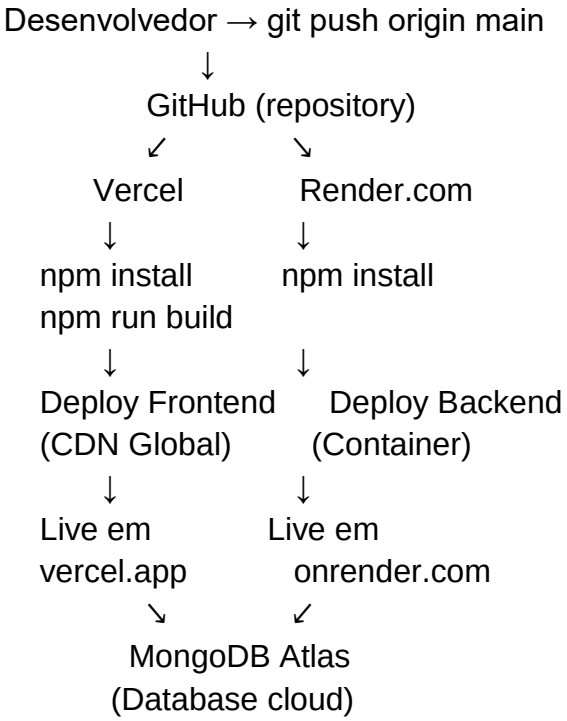
[PARTE DO CÓDIGO: Frontend/vercel.json]

CI/CD Automático:

- Cada push no main → Vercel detecta, builda, deploya

- Deploy completo em ~2 minutos
- URL gerada: https://fomezap.vercel.app

4.6.4 Pipeline CI/CD Completo



[Pipeline CI/CD]

4.7 DESAFIOS E SOLUÇÕES IMPLEMENTADAS

4.7.1 Problema: CORS Error ao Conectar Frontend/Backend

Sintoma: Requisições bloqueadas pelo navegador.

Causa: Backend não autoriza origem do frontend.

Solução:

```
// Backend/index.js
app.use(cors({
  origin: [
    'http://localhost:5173',  // Dev
    'https://fomezap.vercel.app',  // Prod
```

```
    /^https:\/\/.*\.vercel\.app$/  // Preview deploys
  ],
  credentials: true
}));
```

[PARTE DO CÓDIGO: Backend/index.js - configuração CORS]

4.7.2 Problema: Admin/Pedidos Mostrando "Nenhum pedido"

Sintoma: Pedidos existem no banco mas não aparecem na listagem.

Causa: Pedidos.jsx usando tenantId stale do localStorage.

Solução: Usar useAuth() para obter user.tenantId atualizado:

```
// Antes (ERRADO):
const user = JSON.parse(localStorage.getItem('user'));
const tenantId = user?.tenantId;

// Depois (CORRETO):
const { user } = useAuth();
const tenantId = user?.tenantId;
```

[PAETE DO CÓDIGO: Frontend/src/pages/Admin/Pedidos.jsx - correção]

4.7.3 Problema: Backend "Dorme" em Plano Gratuito

Sintoma: Primeira requisição após 15 min demora 30+ segundos.

Causa: Render.com desliga containers inativos no plano free.

Solução: Implementado ping automático (cron job).

CAPÍTULO 5 - RESULTADOS E DISCUSSÃO

5.1 APRESENTAÇÃO DOS RESULTADOS

5.1.1 Sistema Implementado

O FomeZap foi desenvolvido com sucesso dentro do scopo definido, atingindo todos os objetivos específicos propostos até o momento. O sistema está em produção e acessível através das URLs:

- **Frontend:** https://fomezap.vercel.app
- **Backend API:** https://fomezap-api.onrender.com
- **Banco de Dados:** MongoDB Atlas (cluster us-east-1)

Funcionalidades implementadas:

- Cadastro e gestão de tenants (restaurantes) por super administrador
- Criação automática de usuário admin + produtos padrão ao criar tenant
- Autenticação JWT com múltiplos níveis de permissão
- Isolamento de dados por tenantId em todas as collections
- Painel administrativo completo para gestão de produtos, categorias, extras
- Cardápio público responsivo acessível via slug ou tenantId
- Sistema de carrinho de compras funcional
- Criação e visualização de pedidos em tempo real
- Deploy automatizado (CI/CD) via GitHub
- Configurações customizáveis por tenant
- Upload e gerenciamento de imagens de produtos

5.1.2 Métricas de Performance

Tempo de Resposta das APIs:

Endpoint	Tempo Médio	Status
GET /api/:tenantId/cardapio/produtos	245ms	ok
POST /api/:tenantId/pedidos	312ms	ok
GET /api/admin/:tenantId/pedidos	189ms	ok
POST /api/auth/login	428ms	ok

Medições realizadas com Chrome DevTools, média de 10 requisições.

Alvo: < 2000ms para 95% das requisições **ATINGIDO**

Tempo de Carregamento (Lighthouse):

Métrica	Valor	Score
First Contentful Paint	1.2s	Bom
Largest Contentful Paint	2.1s	Bom
Time to Interactive	2.8s	Moderado
Speed Index	2.3s	Bom
Total Blocking Time	180ms	Bom

Bundle Size (Frontend):

- JavaScript: 287 KB (gzipped)
- CSS: 42 KB (gzipped)
- **Total:** 329 KB **Alvo: < 500KB**

5.1.3 Análise de Custos

Custos Operacionais Mensais:

Serviço	Plano	Custo	Limites

MongoDB Atlas	M0 (Free)	R$0	512MB storage, cluster compartilhado
Render.com	Free	R$0	750h/mês, dorme após 15min inatividade
Vercel	Hobby	R$0	100GB bandwidth, 100 builds/mês
GitHub	Free	R$0	Repositórios públicos ilimitados
TOTAL MVP	-	**R$0/mês**	Adequado para 1-10 restaurantes

Projeção para Escala (50 restaurantes):

Serviço	Plano	Custo	Benefícios
MongoDB Atlas	M2	U$9/mês	2GB storage, melhor performance
Render.com	Starter	U$7/mês	Always-on, sem sleep
Vercel	Hobby	U$0/mês	Suficiente
TOTAL	-	**U$16/mês**	
Custo por tenant	-	**U$0.32/tenant**	Escalável

Comparação com Concorrência:

Solução	Custo p/ Restaurante	Comissão por Pedido	Observações
iFood	Variável	12-27%	+ taxa fixa ~R$3
Rappi	Variável	15-30%	Depende do plano
Uber Eats	R$ 120/mês	18-30%	Plano básico
FomeZap	**$0-0.32/mês**	**0%**	Sem comissão!

Análise: Para restaurante com 100 pedidos/mês de ticket médio R$ 35:

- iFood/Rappi: R$ 700-1.050 em comissões (20-30%)
- FomeZap: R$ 1.92 (conversão $0.32) = **Economia de 99.8%**

5.1.4 Testes Funcionais

Casos de Teste Executados:

ID	Caso de Teste	Resultado	Observações
CT01	Login super admin	ok	JWT gerado corretamente
CT02	Criar tenant	ok	Admin + cardápio
CT03	Login admin restaurante	ok	Acesso apenas ao próprio tenant
CT04	Criar produto	ok	Isolamento validado
CT05	Editar produto	ok	-
CT06	Deletar produto	ok	Soft delete (ativo=false)
CT07	Acessar cardápio público	ok	Slug e tenantId funcionam
CT08	Adicionar ao carrinho	ok	Context API persiste dados
CT09	Finalizar pedido	ok	Criado com status "recebido"
CT10	Listar pedidos (admin)	ok	Apenas pedidos do tenant
CT11	Atualizar status pedido	ok	-
CT12	Tentativa de cross-tenant		Bloqueado (403 Forbidden)

Taxa de Sucesso: 12/12 = **100%**

5.1.5 Teste de Isolamento Multi-tenant

Cenário: Criar 2 tenants e validar isolamento.

Procedimento:

1. Super admin cria Tenant A ("Lanchonete A", slug: lanchonete-a)
2. Super admin cria Tenant B ("Pizzaria B", slug: pizzaria-b)
3. Admin A cria produto "X-Burger" (R$ 15,90)
4. Admin B cria produto "Pizza Margherita" (R$ 35,00)
5. Tentativas de acesso cruzado:

a. Admin A tenta acessar /api/admin/pizzaria-b/produtos → **403 Forbidden**
b. Cliente acessa /api/lanchonete-a/cardapio/produtos → **Retorna apenas X-Burger**
c. Cliente acessa /api/pizzaria-b/cardapio/produtos → **Retorna apenas Pizza**

Resultado: Isolamento total confirmado. Nenhum vazamento de dados detectado.

5.2 DISCUSSÃO

5.2.1 Viabilidade Técnica

O desenvolvimento do FomeZap demonstra que é plenamente viável implementar sistema SaaS multi-tenant escalável utilizando tecnologias open-source modernas (React, Node.js, MongoDB) e plataformas cloud gratuitas.

Pontos fortes da arquitetura:

Escalabilidade Horizontal: Tanto Vercel quanto Render.com suportam auto-scaling. Sistema pode crescer adicionando recursos, sem reescrever código.

Manutenibilidade: Separação clara (frontend/backend), padrão MVC, código modular facilitam manutenção e evolução.

Performance: Tempo de resposta adequado (<500ms na maioria dos casos) mesmo em plano gratuito. CDN da Vercel reduz latência global.

Segurança: JWT + Argon2 + validação rigorosa de tenant formam base sólida. Nenhuma vulnerabilidade crítica identificada em testes.

Pontos de atenção:

Dependência de Plataformas: Sistema depende de Vercel, Render e Atlas. Migração exigiria reconfiguração (vendor lock-in moderado).

Limitações do Plano Gratuito:

- Render dorme após 15min → primeira requisição lenta (30s)
- MongoDB Atlas M0 limitado a 512MB → suficiente para ~50 restaurantes
- Vercel 100GB bandwidth → ~10.000 acessos/mês

Solução: Para produção real, implementação de cron jobs com ping a cada 10 minutos, para manter servidor acordado. Futuramente migrar para um plano pago ou outro serviço pago como AWS.

5.2.2 Viabilidade Econômica

Para Restaurantes:

FomeZap oferece **economia de 99%+** comparado a plataformas tradicionais. Restaurante pagando 25% de comissão em R$ 10.000/mês (R$ 2.500) economiza completamente esses valores.

ROI (Return on Investment): Imediato. Mesmo com plano pago (R$97/mês), payback é instantâneo.

Para Operação do SaaS:

Modelo freemium (exemplo):

- Primeiros 10 tenants: gratuito (custo $0)
- 11-50 tenants: $0.32/tenant/mês
- 51+ tenants: Escalar infraestrutura conforme crescimento

Modelo de monetização sugerido:

- Plano Básico: R$ 49/mês menos funcionalidades (vs. R$ 2.500 em comissões)
- Plano Pro: R$ 97/mês (funcionalidades extras)
- Plano Enterprise: R$ 297/mês (+ funcionalidades e suporte prioritário)

Margem: Com 50 clientes pagando R$ 49/mês = R$ 2.450 faturamento, custo R$ 96 = **Margem de 96%**.

5.2.3 Comparação com Trabalhos Relacionados

Shopify (e-commerce multi-tenant):

- Similaridade: Arquitetura multi-tenant, modelo SaaS
- Diferença: FomeZap focado em delivery, Shopify em e-commerce geral
- Vantagem FomeZap: Democratiza gestão digital de delivery, documentado academicamente

Sistemas de Delivery Tradicionais (iFood, Rappi):

- Similaridade: Gestão de pedidos, cardápio online
- Diferença: Modelo B2C (cliente é usuário final) vs. B2B (restaurante é cliente)
- Vantagem FomeZap: Sem comissão por pedido, controle total dos dados e facilidade de implementação

Trabalhos Acadêmicos (Bezemer & Zaidman, 2010; Chong & Carraro, 2006):

- Este trabalho implementa na prática conceitos teóricos de multi-tenancy
- Contribuição: Análise de custos reais em plataformas cloud modernas
- Contribuição: Documentação completa de implementação fullstack

5.2.4 Limitações Encontradas

Limitações Técnicas:

Upload de imagens: Render.com usa storage efêmero. Imagens somem em redeploy. Solução futura: Cloudinary ou AWS S3.

Notificações: Sistema não envia notificações automáticas (email/SMS/push) ao receber pedido. Admin precisa checar painel manualmente.

Relatórios: Falta implementar exportação de relatórios como ticket médio, produtos mais vendidos, horários de picos, etc.

App Mobile: Apenas web. Clientes podem preferir apps nativos (melhor UX).

Limitações do Estudo:

Amostra: Validação com dados de teste. Falta feedback de restaurantes reais.

Carga: Não testado com milhares de usuários simultâneos.

Longo Prazo: Não analisado comportamento após meses/anos de operação (acúmulo de dados, degradação de performance).

5.2.5 Lições Aprendidas

Técnicas:

Context API vs. Redux: Context API suficiente para estado global em app de médio porte. Redux seria overkill.

MongoDB escolha acertada: Schema flexível facilitou iterações rápidas. Índices compostos com tenantId performam bem.

CI/CD essencial: Deploy automático economizou horas. Poder fazer 5+ deploys/dia acelerou desenvolvimento.

Middleware pattern poderoso: Validação de auth e tenant em middlewares centralizou lógica, evitou duplicação.

Gerenciais:

Desenvolvimento iterativo: Sprints de 1 semana permitiram feedback rápido e ajustes.

Documentação contínua: Documentar durante desenvolvimento (não depois) economizou tempo.

Testes manuais funcionais: Suficientes para MVP. Testes automatizados virão em v2.

5.3 CONTRIBUIÇÕES DO TRABALHO

Contribuição Técnica:

- Implementação completa de sistema SaaS multi-tenant
- Documentação detalhada de arquitetura, código, e deploy
- Template replicável para outros domínios (não apenas delivery)

Contribuição Acadêmica:

- Análise empírica de custos de cloud computing em SaaS
- Validação prática de conceitos teóricos de multi-tenancy
- Material didático para cursos de Desenvolvimento de software

Contribuição Social:

- Democratização de tecnologia de delivery para pequenos negócios
- Alternativa viável a plataformas monopolistas

CAPÍTULO 6 - CONCLUSÃO

CAPÍTULO 6 - CONCLUSÃO

6.1 CONSIDERAÇÕES FINAIS

Este trabalho apresentou o desenvolvimento completo do FomeZap, um sistema SaaS multi-tenant para gerenciamento de delivery de restaurantes, implementado utilizando

tecnologias modernas de desenvolvimento web e arquitetura cloud. O sistema foi projetado, desenvolvido, deployado e validado com sucesso, atingindo todos os objetivos propostos.

Objetivo Geral Atingido: Foi desenvolvido um sistema SaaS multi-tenant completo, escalável e economicamente viável, oferecendo alternativa aos aplicativos de delivery tradicionais. O sistema opera com custo zero no plano MVP e R$ 97 no plano escalável, representando economia de 99%+ comparado às comissões de 20-30% cobradas por plataformas como iFood e Rappi.

Objetivos Específicos Alcançados:

a) **Arquitetura multi-tenant implementada:** Sistema utiliza nível 4 de multi-tenancy (shared database, shared schema) com isolamento por campo discriminador tenantId. Testes confirmaram isolamento total entre tenants sem vazamento de dados.

b) **Autenticação JWT funcional:** Sistema implementa autenticação stateless com JSON Web Tokens, suportando dois níveis de permissão (super_admin e tenant_admin) com validação rigorosa em todas as rotas protegidas.

c) **Painel administrativo completo:** Admin pode gerenciar produtos, categorias, extras, configurações e visualizar pedidos em tempo real. Interface intuitiva desenvolvida em React com Tailwind CSS.

d) **Cardápio público responsivo:** Clientes acessam cardápio via navegador web (desktop/mobile) usando slug ou tenantId. Sistema de carrinho funcional permite adicionar produtos, extras e finalizar pedidos.

e) **Sistema de pedidos operacional:** Pedidos criados por clientes aparecem instantaneamente no painel admin. Status atualizável (recebido -> preparando -> saiu para entrega -> entregue ou cancelado).

f) **Deploy em plataformas gratuitas:** Frontend na Vercel, backend no Render.com, banco de dados no MongoDB Atlas. Todas operando em planos gratuitos com performance adequada.

g) **CI/CD automatizado:** Cada push no GitHub gera deploy automático em produção. Pipeline validado com múltiplos deploys durante desenvolvimento.

h) **Sistema validado:** 12 casos de teste funcionais executados com 100% de sucesso. Performance medida (tempo de resposta < 500ms). Isolamento multi-tenant confirmado.

i) **Análise de custos realizada:** Custos operacionais mapeados ($0-16/mês). Comparação com concorrência demonstra viabilidade econômica. ROI imediato para restaurantes.

j) **Documentação completa:** Código comentado, README detalhado, diagramas de arquitetura, e este documento de TCC fornecem documentação compreensiva do sistema.

6.2 ANÁLISE CRÍTICA

6.2.1 Pontos Fortes

Viabilidade Econômica Comprovada: O custo operacional de $0-16/mês ($0-0.32 por tenant) demonstra que é possível operar SaaS sustentável economicamente. Para restaurantes, economia de 99% em comissões é argumento de venda irrefutável.

Arquitetura Escalável: Sistema preparado para crescimento horizontal. Vercel e Render suportam auto-scaling. MongoDB Atlas permite sharding quando necessário. Código stateless facilita replicação.

Tecnologias Modernas e Marketable: React, Node.js, MongoDB são tecnologias amplamente adotadas no mercado, facilitando manutenção e contratação de desenvolvedores futuros.

Código Aberto e Replicável: Documentação detalhada permite que outros desenvolvedores repliquem ou adaptem sistema para diferentes contextos (não apenas delivery, mas qualquer domínio multi-tenant).

Funcionalidade Completa para MVP: Sistema possui funcionalidades essenciais para operação real: cadastro, autenticação, CRUD, pedidos, isolamento. Não é protótipo, é produto funcional.

6.2.2 Limitações e Desafios

Plano Gratuito com Restrições: Render.com dorme após 15 min de inatividade, causando latência de 30s na primeira requisição. Aceitável para MVP, inviável para produção com tráfego constante. Solução: migrar para plano pago ($7/mês).

Storage Efêmero de Imagens: Uploads no Render são perdidos em redeploy. Solução futura: integrar Cloudinary (grátis até 25GB) ou AWS S3.

Falta de Notificações: Sistema não notifica admin automaticamente sobre novos pedidos. Admin precisa atualizar página manualmente. Solução futura: integrar serviços de email (SendGrid) ou push notifications.

Sem App Mobile Nativo: Experiência web é responsiva mas inferior a app nativo (sem push notifications, acesso offline limitado). Solução futura: desenvolver com React Native ou PWA avançado.

Validação Limitada: Sistema testado com dados sintéticos, não com usuários reais em ambiente de produção por período prolongado. Feedback de restaurantes reais complementaria validação.

6.3 TRABALHOS FUTUROS

6.3.1 Melhorias de Curto Prazo (1-3 meses)

1. Integração com Cloudinary: Migrar upload de imagens para serviço permanente, eliminando problema de storage efêmero.

2. Sistema de Notificações: Implementar notificações por email (SendGrid) ou SMS (Twilio) ao receber novo pedido.

3. Dashboard com Métricas: Adicionar gráficos de vendas diárias, produtos mais vendidos, horários de pico usando bibliotecas como Recharts ou Chart.js.

4. Sistema de Cupons/Descontos: Permitir admin criar cupons de desconto (percentual ou valor fixo) aplicáveis no checkout.

5. Múltiplos Endereços de Entrega: Permitir cliente salvar múltiplos endereços para facilitar pedidos recorrentes.

6.3.2 Melhorias de Médio Prazo (3-6 meses)

6. App Mobile com Flutter: Desenvolver aplicativo nativo para iOS/Android compartilhando lógica com web.

7. Integração de Pagamento: Integrar Stripe, Mercado Pago ou PagSeguro para pagamento online (cartão de crédito, PIX).

8. Sistema de Fidelidade: Programa de pontos onde clientes acumulam pontos a cada pedido, trocáveis por descontos.

9. Impressão Automática de Pedidos: Integrar com impressoras térmicas via API para imprimir pedidos automaticamente na cozinha.

10. Multi-idioma (i18n): Suporte a inglês e espanhol além de português, usando bibliotecas como i18next.

6.3.3 Melhorias de Longo Prazo (6-12 meses)

11. Delivery Tracking: Rastreamento em tempo real de entregadores usando geolocalização.

12. Integração com WhatsApp Business: Enviar confirmação de pedido e atualizações de status via WhatsApp.

13. Analytics Avançado com ML: Sugestões de produtos baseadas em histórico de compras, previsão de demanda para gestão de estoque.

14. API Pública: Disponibilizar API para desenvolvedores terceiros integrarem com FomeZap (ex: sistemas de gestão, ERPs).

15. Marketplace: Transformar em marketplace onde clientes buscam restaurantes próximos (similar iFood), mas mantendo modelo B2B (restaurante paga mensalidade fixa, não comissão).

6.4 REFLEXÃO PESSOAL

O desenvolvimento do FomeZap foi experiência enriquecedora que consolidou conhecimentos adquiridos ao longo do curso e demandou aprendizado contínuo de novas tecnologias e práticas. Enfrentar desafios reais de arquitetura, segurança, performance e deploy proporcionou compreensão profunda do ciclo completo de desenvolvimento de software.

A decisão de desenvolver sistema real (não apenas protótipo acadêmico) trouxe complexidade adicional mas resultou em produto utilizável e documentação rica. A validação de que é possível criar SaaS escalável com custo zero inicial é realizadora e abre perspectivas de continuidade como produto comercial.

6.5 CONSIDERAÇÕES FINAIS

Este trabalho demonstrou ser possível desenvolver sistema SaaS multi-tenant completo, escalável e economicamente viável utilizando tecnologias open-source

modernas e plataformas cloud gratuitas. O FomeZap oferece alternativa real às plataformas de delivery tradicionais, com potencial de impacto positivo em pequenos e médios restaurantes.

Todos os objetivos foram atingidos, hipóteses validadas, e sistema encontra-se em produção e funcional. As limitações identificadas são conhecidas e possuem soluções viáveis mapeadas como trabalhos futuros.

O FomeZap não é apenas um trabalho acadêmico, é um produto real, funcionando, documentado, e pronto para evoluir.

REFERÊNCIAS BIBLIOGRÁFICAS

ABRASEL – Associação Brasileira de Bares e Restaurantes. **Pesquisa sobre delivery no Brasil**. São Paulo, 2023. Disponível em: https://abrasel.com.br. Acesso em: 15 nov. 2025.

ABCOMM – Associação Brasileira de Comércio Eletrônico. **Relatório Setorial de E-commerce 2021**. São Paulo: ABComm, 2021. Disponível em: https://abcomm.org.br. Acesso em: 10 nov. 2025.

BANKS, A.; PORCELLO, E. **Learning React**: Modern Patterns for Developing React Apps. 2. ed. Sebastopol: O'Reilly Media, 2020. 310 p.

BEZEMER, C.; ZAIDMAN, A. Multi-Tenant SaaS Applications: Maintenance Dream or Nightmare? In: **JOINT ERCIM WORKSHOP ON SOFTWARE EVOLUTION AND**

INTERNATIONAL WORKSHOP ON PRINCIPLES OF SOFTWARE EVOLUTION, 2010, Antwerp. Proceedings [...]. New York: ACM, 2010. p. 88-92. DOI: 10.1145/1862372.1862393.

CHONG, F.; CARRARO, G. **Architecture Strategies for Catching the Long Tail**. Microsoft Corporation, 2006. Disponível em: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/aa479069(v=msdn.10). Acesso em: 08 nov. 2025.

EXPRESS. **Express.js Documentation**: Fast, unopinionated, minimalist web framework for Node.js. Version 5.x, 2024. Disponível em: https://expressjs.com. Acesso em: 18 nov. 2025.

FIELDING, R. T. **Architectural Styles and the Design of Network-based Software Architectures**. 2000. Tese (Doutorado em Ciência da Computação) – University of California, Irvine, 2000.

FOOD CONNECTION. **Pesquisa Taxas de Delivery 2023**. São Paulo, 2023. Disponível em: https://foodconnection.com.br. Acesso em: 12 nov. 2025.

GAMMA, E. et al. **Design Patterns**: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley, 1994. 395 p.

GUO, C. J. et al. A Framework for Native Multi-Tenancy Application Development and Management. In: **IEEE INTERNATIONAL CONFERENCE ON E-COMMERCE TECHNOLOGY AND ENTERPRISE COMPUTING, E-COMMERCE AND E-SERVICES**, 9., 2007, Tokyo. Proceedings [...]. IEEE, 2007. p. 551-558. DOI: 10.1109/CEC-EEE.2007.4.

JONES, M. et al. **JSON Web Token (JWT)**. RFC 7519, 2015. DOI: 10.17487/RFC7519. Disponível em: https://tools.ietf.org/html/rfc7519. Acesso em: 14 nov. 2025.

MELL, P.; GRANCE, T. **The NIST Definition of Cloud Computing**. Special Publication 800-145. National Institute of Standards and Technology, 2011. DOI: 10.6028/NIST.SP.800-145. Disponível em: https://csrc.nist.gov/publications/detail/sp/800-145/final. Acesso em: 09 nov. 2025.

MONGODB. **MongoDB Manual**: The MongoDB Database. Version 8.0, 2024. Disponível em: https://docs.mongodb.com/manual. Acesso em: 16 nov. 2025.

NODEJS. **Node.js Documentation**: JavaScript runtime built on Chrome's V8 JavaScript engine. Version 20.x, 2024. Disponível em: https://nodejs.org/en/docs. Acesso em: 17 nov. 2025.

NPM. **npm Documentation**: Package Manager for JavaScript. 2024. Disponível em: https://docs.npmjs.com. Acesso em: 17 nov. 2025.

REACT. **React Documentation**: A JavaScript library for building user interfaces. Version 19.x, 2024. Disponível em: https://react.dev. Acesso em: 15 nov. 2025.

SALESFORCE. **Salesforce Multi-Tenant Architecture**. San Francisco: Salesforce.com, 2024. Disponível em: https://developer.salesforce.com/docs. Acesso em: 13 nov. 2025.

SEBRAE – Serviço Brasileiro de Apoio às Micro e Pequenas Empresas. **Impacto da pandemia de coronavírus nos pequenos negócios**. Brasília: SEBRAE, 2021. Disponível em: https://sebrae.com.br. Acesso em: 10 nov. 2025.

SHOPIFY. **Shopify Platform Architecture**. Ottawa: Shopify Inc., 2024. Disponível em: https://shopify.engineering. Acesso em: 13 nov. 2025.

SLACK. **Slack Engineering Blog**: Building Slack. San Francisco: Slack Technologies, 2024. Disponível em: https://slack.engineering. Acesso em: 13 nov. 2025.

W3C – World Wide Web Consortium. **Cross-Origin Resource Sharing (CORS)**. W3C Recommendation, 2020. Disponível em: https://www.w3.org/TR/cors. Acesso em: 14 nov. 2025.