Documentação Técnica do Projeto: Sistema de Reserva De Restaurante

I. Resumo Executivo

Este documento apresenta uma análise técnica detalhada do projeto Unireserva. O objetivo principal do projeto é o desenvolvimento de um sistema web completo para o gerenciamento de reservas em restaurantes, utilizando uma arquitetura moderna com um back-end robusto em Java (Spring Framework) e um front-end em Angular, com persistência de dados em PostgreSQL.[1]

Uma característica fundamental destacada na documentação do projeto é a ênfase em boas práticas de engenharia de software, incluindo versionamento de código, documentação abrangente e integração contínua.[1] Esta abordagem sugere um nível de maturidade no desenvolvimento, visando não apenas a entrega de um software funcional, mas também a aplicação de metodologias e padrões valorizados no ambiente acadêmico e profissional. A escolha por desenvolver um "sistema web completo" [1] indica um escopo considerável, abrangendo tanto a interface do usuário quanto a lógica de negócios e a camada de dados no servidor. Tal amplitude posiciona o projeto como uma peça significativa para o portfólio dos estudantes envolvidos, demonstrando um conjunto diversificado de habilidades técnicas.

II. Introdução

A. Título e Origem do Projeto

O projeto é oficialmente intitulado "Unireserva".[1] Trata-se de uma iniciativa acadêmica, desenvolvida por um grupo de estudantes universitários, com o propósito de aplicar conhecimentos teóricos na construção de uma solução de software prática e relevante.

B. Descrição Detalhada e Objetivos do Projeto

O projeto visa criar um sistema web abrangente e funcional para o gerenciamento de reservas em restaurantes. A arquitetura planejada compreende um back-end desenvolvido em Java, utilizando o Spring Framework (especificamente Spring Boot), e um front-end interativo construído com Angular, embora os detalhes específicos sobre a implementação do front-end não estejam extensivamente cobertos nas informações primárias disponíveis.[1] O sistema deve permitir que usuários realizem, visualizem e gerenciem suas reservas, enquanto, potencialmente, oferece funcionalidades para que a equipe do restaurante administre mesas e agendamentos.

Um dos objetivos centrais, explicitamente mencionado, é a aderência a "boas práticas

em versionamento, documentação e integração contínua".[1] Este foco transcende a mera codificação, indicando uma preocupação com a qualidade, manutenibilidade e profissionalismo no ciclo de vida do desenvolvimento de software. A descrição do projeto como um "sistema web completo" [1] aponta para um esforço de desenvolvimento full-stack. Embora as informações extraídas do repositório, principalmente do arquivo README.md, concentrem-se predominantemente nos componentes do back-end em Java/Spring, a menção ao Angular sinaliza a intenção de uma interface de usuário rica. A ausência de detalhes aprofundados sobre o front-end sugere que a documentação principal analisada prioriza a descrição da infraestrutura do servidor, tornando este relatório, por consequência, mais detalhado nesse aspecto.

É notável que o objetivo de seguir "boas práticas em documentação" [1] é intrinsecamente realizado pela própria existência do detalhado arquivo README.md no repositório do projeto, e, por extensão, pela elaboração deste documento técnico. A riqueza de informações contidas no README.md, como diagramas, planos de teste e descrições arquiteturais, é uma manifestação direta do cumprimento dessa meta.[1]

III. Arquitetura e Design do Sistema

A. Arquitetura Geral

O Unireserva é concebido como uma aplicação web multi-camadas. Conforme descrito, ele consiste em um back-end desenvolvido em Java com o framework Spring e um front-end em Angular.[1] O back-end expõe APIs REST, que são consumidas pelo front-end para realizar as operações do sistema. O diagrama de interação mencionado na documentação do projeto ilustra o fluxo de alto nível: o Usuário interage com o Frontend (Angular), que por sua vez se comunica com o Backend (Spring), o qual acessa o Banco de Dados (PostgreSQL) para persistência e recuperação de dados.[1]

A escolha por uma arquitetura desacoplada, com um front-end Angular independente consumindo uma API REST do back-end Spring, reflete um padrão moderno no desenvolvimento web. Esta separação promove maior escalabilidade, facilita a manutenção e permite uma clara divisão de responsabilidades entre as equipes de desenvolvimento do front-end e do back-end, ou mesmo entre diferentes módulos de desenvolvimento. Tal abordagem permite que o desenvolvimento de cada camada prossiga com relativa autonomia, utilizando as tecnologias e ferramentas mais adequadas para cada contexto.

B. Arquitetura do Back-end (Java Spring Boot)

A estrutura do projeto back-end, baseada em Spring Boot, segue convenções bem estabelecidas, conforme detalhado na documentação do projeto.[1] Essa organização

modular facilita a compreensão e a manutenção do código:

- **controller/**: Responsável por manipular requisições HTTP (GET, POST, PUT, DELETE) e expor os endpoints REST da aplicação. Atua como a porta de entrada para as interações do cliente com o sistema.
- **model/**: Contém as entidades do sistema, que são representações das tabelas no banco de dados (Plain Old Java Objects - POJOs). Estas classes são tipicamente anotadas com JPA para mapeamento objeto-relacional.
- **repository/**: Abriga interfaces que facilitam a comunicação com o banco de dados. Estas interfaces comumente utilizam JpaRepository para simplificar as operações CRUD (Create, Read, Update, Delete).
- **service/**: Implementa a lógica de negócios da aplicação. Os controllers invocam os serviços, que por sua vez utilizam os repositórios para interagir com a camada de dados.
- **resources/application.properties**: Arquivo de configuração central do Spring Boot, contendo definições como a porta do servidor e detalhes de conexão com o banco de dados.
- **resources/static/**: Diretório para armazenar arquivos estáticos como imagens, CSS e JavaScript, caso a aplicação sirva diretamente o front-end (o que é menos provável neste projeto, dada a menção a um front-end Angular separado).
- **resources/templates/**: Pasta para templates HTML, utilizados por motores como Thymeleaf, se aplicável (novamente, menos provável com um front-end Angular dedicado).
- **test/**: Contém os testes automatizados da aplicação, organizados de forma similar ao código principal.

A adoção desta estrutura de pastas é altamente convencional para projetos Spring Boot.[1] Isso indica uma aderência a padrões estabelecidos, o que é benéfico para a integração de novos desenvolvedores ao projeto e para a sua manutenção a longo prazo, pois a localização e o propósito de cada componente são previsíveis. É importante notar que, devido à impossibilidade de inspecionar o conteúdo específico dessas pastas durante a análise [2], esta documentação se baseia nas descrições de suas funções fornecidas no arquivo README.md, em vez de uma análise direta dos arquivos de código-fonte contidos nelas. As descrições em [1] refletem os papéis genéricos e esperados para esses diretórios em uma aplicação Spring Boot.

C. Arquivos de Configuração Chave

Dois arquivos são centrais para a configuração e construção do projeto back-end:

- **pom.xml**: O Project Object Model (POM) do Maven. Este arquivo XML define as

dependências do projeto, plugins de build, metadados do projeto e o ciclo de vida da construção. O README.md lista dependências chave configuradas neste arquivo, como Java 21, Spring Boot 3.4.5, Spring Data JPA, driver PostgreSQL, Flyway, MapStruct, Lombok, Springdoc OpenAPI e JUnit 5.[1]

- **application.properties**: Localizado em src/main/resources/, este arquivo contém configurações específicas da aplicação. Isso inclui strings de conexão com o banco de dados PostgreSQL, a porta do servidor [1], níveis de log e outras configurações personalizadas da aplicação.[1]

O uso do Maven (através do pom.xml) para gerenciamento de dependências e do arquivo application.properties para configurações externalizadas são práticas padrão em projetos Spring Boot. Essa abordagem simplifica significativamente o gerenciamento de bibliotecas e permite que a aplicação seja configurada para diferentes ambientes (desenvolvimento, teste, produção) sem a necessidade de alterações no código-fonte, o que é crucial para a flexibilidade e manutenibilidade do sistema.

D. Design do Banco de Dados

O design do banco de dados é um componente crítico do sistema de reservas. As informações disponíveis [1] indicam o uso do PostgreSQL como sistema de gerenciamento de banco de dados relacional.

- **Visão Geral das Entidades:** Com base nas menções a um classDiagram (Diagrama de Classes) e um erDiagram (Diagrama de Entidade-Relacionamento) no README.md [1], as entidades centrais do sistema são: Usuario (Usuário), Restaurante (Restaurante), Mesa (Mesa) e Reserva (Reserva). Embora os diagramas em si não tenham sido diretamente visualizados, a descrição de seu conteúdo é fundamental.
- **Relacionamentos** [1]**:**
  - Um Usuario pode realizar múltiplas Reservas.
  - Uma Reserva pertence a um Usuario.
  - Uma Reserva é para uma Mesa.
  - Uma Mesa pode ter múltiplas Reservas (em horários diferentes).
  - Uma Mesa pertence a um Restaurante.
  - Um Restaurante possui múltiplas Mesas.
  - Os relacionamentos entre Restaurante e Reserva podem ser indiretos via Mesa, ou diretos se uma reserva estiver vinculada primeiramente a um restaurante.
- **Modelos de Dados** [1]**:**
  - Usuario: Provavelmente contém detalhes do usuário como ID, nome,

informações de contato e credenciais.
    - Restaurante: Detalhes sobre o restaurante, como ID, nome, endereço e capacidade.
    - Mesa: Informações da mesa, como ID, número da mesa, capacidade e ID do restaurante ao qual pertence.
    - Reserva: Detalhes da reserva, incluindo ID, ID do usuário, ID da mesa, data, hora, número de pessoas, status e observações. A tabela "Campos da Reserva" detalhada em [1] fornece uma base para esses atributos.
- **Migração de Banco de Dados:** O projeto utiliza Flyway para gerenciar as migrações de esquema do banco de dados.[1] Isso significa que scripts SQL, localizados em src/main/resources/db/migration (como os hipotéticos V1__create_tables.sql e V2__insert_data.sql), são responsáveis por evoluir o esquema do banco de dados de forma versionada e controlada. A inacessibilidade do conteúdo específico desses arquivos de migração [2] impede a análise das alterações exatas do esquema, mas a presença do Flyway é, por si só, significativa.

A utilização do Flyway [1] é uma prática recomendada para o gerenciamento de alterações no esquema do banco de dados de maneira automatizada e controlada por versão. Isso é particularmente importante em ambientes de equipe e para a implementação de pipelines de Integração Contínua/Entrega Contínua (CI/CD), pois garante consistência do esquema entre diferentes ambientes e estágios de desenvolvimento, minimizando erros manuais.

Os relacionamentos entre entidades, inferidos a partir das descrições dos diagramas [1], sugerem uma estrutura de banco de dados relacional normalizada. Esta é uma abordagem padrão para sistemas transacionais como uma plataforma de reservas, pois ajuda a reduzir a redundância de dados e a melhorar a integridade dos dados. Os campos específicos listados para a entidade Reserva na seção "Campos da Reserva" do README.md [1] oferecem uma visão clara dos dados capturados para esta entidade central do sistema.

A tabela a seguir resume as entidades de banco de dados inferidas e seus atributos chave, com base nas informações do README.md [1] e em práticas comuns de modelagem de dados para sistemas similares.

Tabela: Entidades de Banco de Dados e Atributos Chave (Inferidos)

Entidade	Atributos Chave Prováveis
Usuario	id, nome, email, senha, telefone
Restaurante	id, nome, endereco, tipo_cozinha, capacidade_total
Mesa	id, numero_mesa, capacidade, status, restaurante_id
Reserva	id, data, hora, nome_cliente (ou usuario_id), telefone_cliente, observacao, numero_pessoas, status_reserva, mesa_id

Esta tabela fornece uma visão geral estruturada das principais estruturas de dados do sistema, facilitando a compreensão do modelo de dados subjacente.

IV. Tecnologias Utilizadas

O projeto "Sistema de Reserva De Restaurante" emprega um conjunto de tecnologias modernas e consolidadas para o desenvolvimento de suas funcionalidades. A seleção dessas ferramentas, detalhada no README.md [1], abrange o desenvolvimento back-end, versionamento de código, e organização e documentação do projeto.

A. Back-end

- **Java:** Linguagem de programação principal para o desenvolvimento do back-end. A documentação especifica o uso do Java 21.[1]
- **Spring Framework (Spring Boot, Spring Security, etc.):** Framework principal para a construção da aplicação. Spring Boot simplifica a configuração e o desenvolvimento de aplicações Spring. A menção a "etc." e o uso de JWT sugerem a provável inclusão do Spring Security para lidar com autenticação e autorização.
- **PostgreSQL:** Sistema de gerenciamento de banco de dados relacional escolhido para a persistência dos dados da aplicação.[1]
- **JWT (JSON Web Tokens):** Utilizado para autenticação baseada em tokens, uma abordagem comum para proteger APIs RESTful, garantindo que apenas usuários autenticados possam acessar determinados recursos.[1]
- **Swagger (Springdoc OpenAPI):** Ferramenta para documentação de APIs e testes interativos. O README.md menciona tanto "Swagger (API documentation)" quanto "Springdoc OpenAPI" como dependência, indicando uma abordagem robusta para a documentação da API.[1]

- **Maven:** Ferramenta de gerenciamento de dependências e automação de build para projetos Java.[1]
- **Railway:** Plataforma opcional para deployment em nuvem, indicando uma consideração pela hospedagem e escalabilidade da aplicação em ambientes de nuvem.[1]

B. Versionamento

- **Git & GitHub:** Sistema de controle de versão distribuído (Git) e plataforma de hospedagem de repositórios (GitHub) para colaboração e gerenciamento do código-fonte. O próprio link do projeto fornecido confirma o uso do GitHub.[1]

C. Organização e Documentação

- **Trello:** Ferramenta para gerenciamento de tarefas e prazos, indicando a aplicação de práticas de gerenciamento de projetos para organizar o desenvolvimento.[1]
- **Swagger:** Reafirmado aqui por seu papel crucial na documentação interativa da API REST, facilitando o entendimento e o uso dos endpoints pelos desenvolvedores do front-end ou outros consumidores da API.[1]

O conjunto de tecnologias selecionado (Java 21, Spring Boot 3.x, PostgreSQL, JWT, Maven) é moderno e amplamente utilizado no desenvolvimento de aplicações Java corporativas.[1] Esta escolha reflete as melhores práticas atuais da indústria e proporciona aos estudantes experiência com tecnologias relevantes para o mercado de trabalho. A utilização de versões recentes, como Java 21 e Spring Boot 3.4.5, demonstra uma preocupação em manter o projeto atualizado com os avanços tecnológicos.

Ademais, a escolha do Spring Boot frequentemente influencia a inclusão de outras tecnologias do seu ecossistema. Por exemplo, para implementar a autenticação baseada em JWT [1] em uma aplicação Spring Boot, o Spring Security é uma escolha natural. Similarmente, para a interação com o banco de dados PostgreSQL [1], o Spring Data JPA oferece uma camada de abstração que simplifica as operações de persistência. Para a documentação da API com Swagger [1], a biblioteca Springdoc OpenAPI é a integração padrão para aplicações Spring Boot. Essa coesão tecnológica contribui para um desenvolvimento mais eficiente e padronizado.

A tabela abaixo resume as principais tecnologias utilizadas no projeto e seus respectivos propósitos:

Tabela: Resumo das Tecnologias Utilizadas

Categoria	Tecnologia	Propósito
Back-end	Java 21	Linguagem de programação principal
	Spring Framework (Boot, Security)	Desenvolvimento de aplicações, segurança
	PostgreSQL	Banco de dados relacional
	JWT	Autenticação baseada em tokens
	Swagger (Springdoc OpenAPI)	Documentação e teste de APIs
	Maven	Gerenciamento de dependências, automação de build
	Railway	Plataforma opcional de deployment em nuvem
Versionamento	Git & GitHub	Controle de versão, colaboração
Organização	Trello	Gerenciamento de tarefas e projetos

V. Componentes Centrais do Sistema (Detalhado)

A arquitetura do back-end do "Sistema de Reserva De Restaurante" é organizada em componentes distintos, cada um com responsabilidades específicas, seguindo o padrão promovido pelo Spring Boot. Esta seção detalha as funções desses componentes principais, com base nas descrições fornecidas pela estrutura de pastas no README.md.[1] Dada a limitação de acesso direto ao código-fonte [2], as descrições se baseiam em suas responsabilidades padrão em uma aplicação Spring Boot.

A. Controladores (controller/)

Os controladores são a camada de entrada da API do back-end. Suas principais

responsabilidades incluem:

- Receber e processar todas as requisições HTTP provenientes do cliente (neste caso, o front-end Angular), como GET, POST, PUT, DELETE.
- Mapear URLs de requisição e métodos HTTP para métodos manipuladores específicos dentro das classes de controller.
- Extrair e validar parâmetros de requisição, cabeçalhos e o corpo da requisição.
- Delegar o processamento da lógica de negócios para os componentes de Service apropriados.
- Receber os resultados dos serviços e formular as respostas HTTP, geralmente no formato JSON, juntamente com os códigos de status HTTP adequados.
- É comum o uso de anotações do Spring MVC como @RestController, @GetMapping, @PostMapping, @PutMapping, @DeleteMapping para definir os endpoints e o comportamento dos controladores. A documentação do projeto confirma este papel: "[esta pasta] lida com requisições HTTP (GET, POST, PUT, DELETE) e expõe os endpoints REST da aplicação".[1]

B. Modelos (model/)

A pasta model/ contém as entidades de dados da aplicação. Estas são:

- Estruturas de dados (classes Java) que representam os objetos de domínio do sistema, como Usuario, Restaurante, Mesa e Reserva, conforme inferido dos diagramas mencionados em [1].
- Tipicamente, são Plain Old Java Objects (POJOs) anotados com anotações da Java Persistence API (JPA), como @Entity, @Table, @Id, @Column, e anotações de relacionamento (@ManyToOne, @OneToMany, etc.). Essas anotações permitem o mapeamento objeto-relacional (ORM) para as tabelas e colunas correspondentes no banco de dados PostgreSQL. O README.md afirma que "[este diretório] contém as entidades do sistema, que representam tabelas no banco de dados".[1]

C. Repositórios (repository/)

Os repositórios formam a camada de acesso a dados, abstraindo as interações com o banco de dados:

- São interfaces que definem as operações de acesso a dados para as entidades do modelo.
- Comumente, estendem a interface JpaRepository<Entity, ID_Type> do Spring Data JPA, que fornece implementações para operações CRUD comuns (Create, Read, Update, Delete) sem a necessidade de código boilerplate, além de funcionalidades para paginação e ordenação.

- Métodos de consulta personalizados podem ser definidos seguindo convenções de nomenclatura de métodos ou utilizando a anotação @Query para consultas JPQL ou SQL nativas.
- Abstraem o mecanismo de persistência de dados subjacente da camada de serviço. A descrição no README.md é: "[esta pasta contém] interfaces que facilitam a comunicação com o banco de dados. Essas interfaces utilizam JpaRepository para simplificar as operações CRUD".[1]

D. Serviços (service/)

A camada de serviço é onde reside a lógica de negócios central da aplicação:

- Implementam as regras de negócios, orquestram chamadas aos Repository para buscar ou persistir dados e realizam validações complexas.
- Podem envolver cálculos, transformações de dados (potencialmente utilizando MapStruct, conforme mencionado nas dependências [1]), e a aplicação de regras específicas do domínio do restaurante e das reservas.
- Frequentemente gerenciam transações, utilizando a anotação @Transactional do Spring para garantir a atomicidade das operações.
- São invocados pelos componentes Controller para executar as ações solicitadas pelo usuário. Conforme o README.md: "[este módulo] implementa a lógica de negócios da aplicação. Os controllers chamam esses serviços, que por sua vez usam os repositórios".[1]

A clara separação de responsabilidades entre controladores, serviços, repositórios e modelos é uma característica fundamental de padrões arquiteturais como o MVC (Model-View-Controller) e arquiteturas em camadas, que o Spring Boot facilita. Esta modularidade promove a testabilidade (pois cada camada pode ser testada isoladamente), a manutenibilidade (alterações em uma camada têm menor impacto nas outras) e a organização geral do código.

A menção ao MapStruct como uma dependência do projeto [1] sugere uma prática importante na camada de serviço: o mapeamento entre Data Transfer Objects (DTOs) e entidades de persistência. DTOs são frequentemente usados para moldar os dados que são expostos pela API, enquanto as entidades representam a estrutura do banco de dados. O uso de MapStruct para automatizar esse mapeamento indica uma conscientização sobre a necessidade de desacoplar essas duas representações de dados. Isso melhora o design da API, tornando-a mais flexível a mudanças e evitando a exposição direta das entidades de persistência, o que também contribui para a segurança. A seção "Casos de Teste de Validação (DTO/Entidade)" no README.md [1]

reforça a probabilidade do uso de DTOs.

VI. Documentação e Interação da API

A comunicação entre o front-end e o back-end é realizada através de uma API REST. A documentação e o entendimento do fluxo de interação dessa API são cruciais para o desenvolvimento e integração do sistema.

A. Documentação da API com Swagger (Springdoc OpenAPI)

O projeto utiliza Swagger para a documentação de sua API, conforme indicado pela inclusão de "Swagger (API documentation)" e da dependência "Springdoc OpenAPI".1 Esta ferramenta permite a geração automática de documentação interativa da API diretamente a partir do código-fonte do back-end, tipicamente através de anotações nos controladores Spring.
Com a interface do Swagger UI, os desenvolvedores (especialmente os do front-end Angular) e outros possíveis consumidores da API podem:

- Visualizar todos os endpoints da API disponíveis.
- Entender os formatos de requisição e resposta esperados para cada endpoint, incluindo parâmetros, cabeçalhos e corpos de mensagem.
- Testar os endpoints da API diretamente pelo navegador, enviando requisições e observando as respostas.

A integração do Swagger/OpenAPI desde o início do projeto demonstra um compromisso com os princípios de "API-first" ou, no mínimo, com a boa prática de garantir a descoberta e usabilidade da API. Isso é de grande valor para os desenvolvedores do front-end que consomem a API, bem como para qualquer integração futura com sistemas de terceiros, pois reduz o atrito na integração e melhora a produtividade geral da equipe de desenvolvimento ao fornecer uma fonte clara e interativa de verdade sobre os contratos da API.

B. Fluxo de Interação (Exemplo de Reserva)

Para ilustrar como os componentes do sistema colaboram, o README.md menciona um sequenceDiagram (Diagrama de Sequência) que detalha o fluxo de interação para uma operação chave, como a realização de uma reserva.[1] Embora o diagrama em si não esteja visível nos dados extraídos, sua descrição permite inferir o seguinte fluxo típico:

1. **Usuário:** O usuário inicia a ação de reserva interagindo com a interface do **Frontend (Angular)** (por exemplo, preenchendo um formulário de reserva e clicando em "reservar").
2. **Frontend (Angular):** A aplicação front-end coleta os dados da reserva e envia uma requisição HTTP (provavelmente um POST com os detalhes da reserva em

formato JSON) para o endpoint apropriado no **Backend (Spring REST API)**.

3. **Backend (Spring) - Controller:** Um Controller no back-end recebe a requisição. Ele valida os dados de entrada (possivelmente com o auxílio de anotações de validação ou DTOs) e, se válidos, chama um método no componente de Service correspondente.
4. **Backend (Spring) - Service:** O método de Service executa a lógica de negócios principal. Isso pode incluir verificar a disponibilidade da mesa para a data e hora solicitadas, validar o status do usuário, aplicar regras de reserva específicas do restaurante, e interagir com os Repositories para acessar ou modificar dados.
5. **Backend (Spring) - Repository e Banco de Dados (PostgreSQL):** Os Repositories executam as operações necessárias no **Banco de Dados (PostgreSQL)**, como salvar a nova reserva na tabela de reservas e, possivelmente, atualizar o status da mesa.
6. **Retorno da Resposta:** A resposta (sucesso ou falha, juntamente com dados relevantes como a confirmação da reserva ou uma mensagem de erro) flui de volta através das camadas: do banco de dados para o repositório, para o serviço, para o controlador, que então formula uma resposta HTTP para o front-end. O front-end, por sua vez, atualiza a interface do usuário para informar o resultado da operação.

A documentação explícita de um fluxo de interação através de um diagrama de sequência [1] é uma ferramenta valiosa para o entendimento do comportamento dinâmico do sistema durante operações críticas. Ela clarifica as responsabilidades de cada componente e os caminhos de comunicação entre as diferentes partes do sistema, facilitando a depuração, a manutenção e a evolução da funcionalidade de reserva.

VII. Estratégia e Roteiro de Testes

O README.md do projeto detalha uma estratégia de testes abrangente, indicando um forte compromisso com a qualidade e a robustez do software.[1] Esta seção descreve os objetivos, abordagens, casos de teste e tecnologias utilizadas para garantir o correto funcionamento do sistema, com foco especial na funcionalidade de cadastro de reservas.

A. Objetivos e Abordagem de Testes

O objetivo primário da estratégia de testes é "assegurar que todas as regras de negócio para o cadastro de reserva sejam validadas".1 Isso demonstra um foco na correção funcional e na robustez do sistema, especialmente para sua funcionalidade central.
A abordagem de testes é multi-camadas, englobando:

- Validação de Dados em DTOs (Data Transfer Objects) e/ou Entidades.

- Testes Unitários para Controladores.
- Testes Unitários para a lógica de negócios e consultas na Camada de Serviço.

B. Campos da Reserva e Regras de Validação

Uma parte crucial da documentação de testes é a especificação dos campos necessários para uma reserva e seus respectivos critérios de validação (por exemplo, obrigatoriedade, tipo de dado, formatos específicos). O README.md descreve uma tabela "Campos da Reserva" [1] que serve de base para esta definição. Os campos incluem data, hora, nome, telefone, observação, número de pessoas e status.

Tabela: Campos da Reserva e Requisitos (Exemplificativo)
1

Campo	**Requisito (Obrigatório, Tipo/Validação Esperada)**
data	Obrigatório, Formato de Data (ex: dd/MM/yyyy)
hora	Obrigatório, Formato de Hora (ex: HH:mm)
nome	Obrigatório, Texto
telefone	Obrigatório, Formato de Telefone válido
observação	Opcional, Texto
número de pessoas	Obrigatório, Inteiro Positivo
status	(ex: Confirmada, Pendente, Cancelada)

Esta tabela define o contrato de dados para a entidade de reserva, servindo como referência para o desenvolvimento da interface do usuário, da lógica de back-end e, crucialmente, dos testes de validação.

C. Casos de Teste [1]

O roteiro de testes detalhado no README.md [1] especifica diferentes categorias de casos de teste:

- **Casos de Teste de Validação (DTO/Entidade):** Focados em garantir a integridade dos dados e a aplicação das regras de validação no nível dos DTOs ou das

entidades. Isso inclui testes para valores nulos em campos obrigatórios, formatos de dados incorretos, violações de restrições de tamanho ou intervalo. O README.md menciona uma tabela específica para esses casos.[1]
    - *Exemplo conceitual*: Testar o cadastro de uma reserva com o campo data nulo deve resultar em um erro de validação, impedindo que a reserva seja salva.
- **Casos de Teste Unitário - Controller:** Testam os métodos dos controladores, geralmente com o uso de mocks para as dependências de serviço. O objetivo é verificar o correto tratamento das requisições, o mapeamento de parâmetros e a geração das respostas HTTP. O README.md descreve testes para funcionalidades como salvar, atualizar, buscar, listar e deletar reservas.[1]
    - *Exemplo conceitual*: Um teste para o endpoint de criação de reserva deve verificar se, ao receber dados válidos, o controller invoca o método de salvar do serviço e retorna um status HTTP 201 (Created) com os dados da reserva criada.
- **Casos de Teste Unitário - Service (Lógica):** Testam a lógica de negócios implementada nas classes de serviço, utilizando mocks para as dependências de repositório. O README.md detalha testes para salvar, atualizar e deletar entidades.[1]
    - *Exemplo conceitual*: Um teste para o serviço de salvar reserva deve verificar se as regras de negócio (ex: disponibilidade de mesa) são aplicadas corretamente antes de persistir a reserva.
- **Casos de Teste Unitário - ServiceQuery (Consulta):** Testam os métodos de serviço que envolvem principalmente a consulta de dados, garantindo que a recuperação e filtragem de informações ocorram conforme o esperado. O README.md menciona testes para buscar clientes existentes/inexistentes e listar todos os clientes.[1]
    - *Exemplo conceitual*: Um teste para buscar um cliente por ID deve retornar os dados corretos do cliente se ele existir, ou um resultado nulo/vazio caso contrário.

D. Tecnologias de Teste

As seguintes tecnologias são listadas para a implementação da estratégia de testes [1]:

- **JUnit 5:** Principal framework de testes para Java.
- **Mockito:** Framework de mocking para criar dublês de teste (test doubles), permitindo isolar unidades de código durante os testes.
- **Hibernate Validator:** Implementação de referência da especificação Bean Validation (JSR 380), usada para implementar e testar restrições de validação em beans Java (DTOs e entidades).

- **Jacoco (opcional):** Ferramenta para medição da cobertura de código pelos testes.

E. Instruções para Executar os Testes

Os testes podem ser executados utilizando o Maven através do comando: mvn clean test.[1] A menção à execução de testes via Visual Studio Code também sugere a integração com IDEs para facilitar o desenvolvimento e a execução de testes.

A estratégia de testes descrita, com sua abordagem multi-camadas e o detalhamento de casos de teste específicos [1], evidencia uma forte ênfase na qualidade e confiabilidade do software. Este nível de rigor no planejamento de testes é um indicador positivo das práticas de engenharia de software adotadas pelo projeto. A inclusão do Hibernate Validator [1] sugere que a validação de beans é utilizada de forma sistemática, provavelmente anotando DTOs ou entidades com restrições de validação (como @NotNull, @Size, @Pattern). Isso permite que as validações sejam tratadas de forma declarativa e eficiente, reduzindo a necessidade de código de validação boilerplate nos controladores ou serviços.

VIII. Dependências, Build e Deploy

A gestão de dependências, o processo de construção (build) e as diretrizes para execução do projeto são aspectos fundamentais do ciclo de desenvolvimento de software. O "Sistema de Reserva De Restaurante" utiliza ferramentas e práticas padrão para esses fins.

A. Principais Dependências do Projeto (do pom.xml)

O arquivo pom.xml define as bibliotecas e frameworks dos quais o projeto depende. As principais dependências, conforme listado no README.md [1], incluem:

- **Java 21**
- **Spring Boot 3.4.5** (gerenciado via spring-boot-starter-parent)
- **Spring Data JPA:** Para persistência de dados e interações com o banco de dados usando a Java Persistence API.
- **Spring Web:** Para a construção de APIs RESTful e aplicações web.
- **PostgreSQL (driver):** Driver JDBC para conectar a aplicação ao banco de dados PostgreSQL.
- **Flyway:** Para gerenciamento de migrações de esquema do banco de dados.
- **MapStruct:** Para mapeamento eficiente entre DTOs (Data Transfer Objects) e entidades.
- **Lombok:** Para reduzir código boilerplate em classes Java (ex: getters, setters, construtores).

- **Springdoc OpenAPI:** Para geração automática de documentação da API no formato OpenAPI (Swagger).
- **JUnit 5:** Framework para a escrita e execução de testes unitários.

A tabela a seguir resume estas dependências e seus propósitos:

Tabela: Principais Dependências do Projeto e Propósito

Dependência	Versão (se especificada)	Propósito
Java	21	Linguagem de programação principal
Spring Boot	3.4.5	Framework de aplicação, simplifica configuração Spring
Spring Data JPA	(parte do Boot)	Persistência de dados, ORM
Spring Web	(parte do Boot)	Construção de serviços web RESTful
Driver PostgreSQL	(gerenciado)	Conectividade com banco de dados PostgreSQL
Flyway	(gerenciado)	Migração de esquema de banco de dados
MapStruct	(não especificada)	Mapeamento DTO para Entidade
Lombok	(não especificada)	Redução de código boilerplate
Springdoc OpenAPI	(não especificada)	Documentação automática de API Swagger
JUnit 5	(não especificada)	Framework de testes unitários

Esta lista de dependências [1] forma a espinha dorsal tecnológica do back-end, fornecendo as ferramentas necessárias para desenvolvimento, persistência de dados, documentação e teste.

B. Processo de Build (Maven)

O projeto utiliza o Apache Maven para gerenciar seu ciclo de vida de construção e dependências. O arquivo pom.xml é central para este processo, definindo como o projeto é compilado, testado e empacotado. Metas padrão do Maven como clean (limpar artefatos de build anteriores), compile (compilar o código-fonte), test (executar testes automatizados) e package (empacotar o código compilado em um formato distribuível, como um JAR) seriam utilizadas. O comando mvn clean test é especificamente mencionado para a execução dos testes [1], o que implicitamente também executa a compilação do código.

A presença dos arquivos mvnw (para Linux/macOS) e mvnw.cmd (para Windows) na raiz do repositório [1] indica o uso do Maven Wrapper. Esta é uma prática recomendada que simplifica o processo de build para novos desenvolvedores ou em ambientes de integração contínua (CI). O Wrapper garante que uma versão consistente do Maven seja usada para construir o projeto, baixando-a automaticamente se necessário, sem exigir uma instalação global do Maven na máquina do desenvolvedor ou no servidor de CI. Isso torna o processo de build mais portátil e reproduzível.

C. Executando o Projeto

As instruções sobre como executar a aplicação são fornecidas no README.md. Especificamente, é afirmado que a API estará disponível em http://localhost:8080 após a inicialização do projeto.[1] Isso implica que a aplicação Spring Boot pode ser iniciada, por exemplo, através do comando Maven mvn spring-boot:run ou executando o arquivo JAR empacotado (gerado pela meta package do Maven) com java -jar nome-do-arquivo.jar.

IX. Conclusão e Suporte

A. Resumo do Estado Documentado

O projeto "Sistema de Reserva De Restaurante", conforme documentado em seu arquivo README.md [1] e na estrutura de alto nível do repositório [1], apresenta-se como um sistema web bem arquitetado, utilizando uma pilha tecnológica moderna e robusta centrada em Java Spring Boot para o back-end e Angular para o front-end. A arquitetura segue padrões de separação de camadas (controller, service, repository, model), e o projeto demonstra um forte compromisso com boas práticas de engenharia de software. Isso é evidenciado pela utilização de ferramentas para gerenciamento de dependências (Maven), migração de banco de dados (Flyway), documentação de API (Swagger/Springdoc OpenAPI) e, notavelmente, por uma estratégia de testes

detalhada e abrangente.

O próprio README.md é um artefato significativo, refletindo o empenho do grupo em produzir uma documentação clara e completa, cobrindo desde a visão geral do projeto até detalhes de implementação, configuração e teste.[1] Este nível de detalhe na documentação é um indicativo positivo da organização e da abordagem metódica adotada pela equipe do projeto.

É fundamental reiterar que esta análise e documentação são baseadas predominantemente nas informações fornecidas no README.md e na estrutura de pastas observável no nível raiz do repositório. Devido a limitações no acesso ao conteúdo detalhado de arquivos de código-fonte específicos [2], este relatório documenta o design e a estrutura *pretendidos e descritos*, em vez de uma verificação exaustiva da implementação em nível de código. No entanto, as informações disponíveis sugerem um projeto sólido e bem planejado, alinhado com as práticas atuais de desenvolvimento de software.

B. Informações de Suporte

Para quaisquer dúvidas ou necessidade de suporte relacionadas ao projeto "Sistema de Reserva De Restaurante", o contato principal fornecido no README.md é o endereço de e-mail: caliel1023@yahoo.com.[1]