RELATÓRIO TÉCNICO FINAL: DELIVERY API

I. INTRODUÇÃO E ESCOPO DO PROJETO

1.1. Resumo Executivo e Arquitetura Global

O **Delivery API** é um projeto de **Arquitetura de Sistemas** desenvolvido como Prova de Conceito (PoC) para um sistema de delivery, focado em demonstrar uma **arquitetura de aplicação profissional e escalável**. O sistema é composto por uma **API RESTful** (Backend em **Spring Boot 3** e **Java 21**) e um **Frontend desacoplado** (em React). O projeto utiliza **Docker** e **Docker Compose** para garantir a portabilidade do ambiente e **JWT** para segurança e autenticação robusta de múltiplos perfis.

1.2. Definição do Escopo (MVP)

O escopo da entrega atual foca no **Mínimo Produto Viável (MVP)**, priorizando a **estrutura técnica sólida**, segurança granular e **observabilidade** sobre a cobertura total de funcionalidades. O MVP valida o **fluxo essencial de sucesso** do negócio: **Criação e Finalização do Pedido** e **Acompanhamento de Pedidos**.

II. ARQUITETURA E DESIGN TÉCNICO

2.1. Padrão Arquitetural em Camadas e Justificativa

O Backend segue o padrão **Arquitetura em Camadas** (*Layered Architecture*), que é essencial para a separação de responsabilidades, manutenibilidade e testabilidade do código. Esta estrutura é refletida nos pacotes do código:

- **`controller`:** Camada de Apresentação, responsável por receber requisições e formatar respostas.
- **`service`:** Camada de Serviço, contendo a **Regra de Negócio** (a lógica central da aplicação).
- **`repository`:** Camada de Acesso a Dados, responsável pela comunicação com o SGBD.

2.2. Otimização de Performance (RFN) e Cache Distribuído

O projeto implementou mecanismos avançados para cumprir o Requisito Não Funcional (RFN) de **Performance**:

- **Cache Distribuído com Redis:** Utiliza a abstração do **Spring Cache** com `@Cacheable` e `@CacheEvict` para reduzir a latência em consultas frequentes.
- **Decisão H2 vs. MySQL (Trade-off):** O banco de dados **H2 em memória** é usado no ambiente de testes, enquanto o **MySQL** é utilizado no ambiente de desenvolvimento/produção via Docker Compose.

2.3. Segurança, Qualidade de Código e Controle de Acesso

- **Segurança (RFN):** A autenticação é baseada em **JSON Web Tokens (JWT)** e **Controle de Acesso Baseado em Papéis (RBAC)**. O sistema suporta perfis **CLIENTE**, **RESTAURANTE**, **ENTREGADOR** e **ADMIN**.
- **Perfis e UX:** A **segregação de login** (botão exclusivo para o Restaurante/Gestão) e a distinção clara de perfis reforçam a usabilidade e a segurança, garantindo que cada usuário acesse apenas suas áreas de responsabilidade.

III. IMPLEMENTAÇÃO CRÍTICA E VALIDAÇÃO

3.1. Requisito de Pagamento Online (Mock Service)

O requisito de Pagamento Online foi cumprido através da criação de um **Mock Service Dedicado** (`PaymentServiceImpl`).

- **Justificativa:** Esta decisão arquitetural permite simular o sucesso da transação no Backend sem depender da complexidade de integração de um Gateway de Pagamento real. O *Mock* é chamado no `PedidoServiceImpl`, e a execução bem-sucedida dos testes provou que, em caso de falha simulada, o `@Transactional` garante o *Rollback* das operações, validando a **Confiabilidade** da transação.

3.2. Observabilidade e Rastreamento Distribuído (Tracing)

O projeto implementou uma arquitetura completa para monitoramento, alinhada com padrões de sistemas distribuídos:

- **Rastreamento Distribuído (Tracing):** Utiliza **Micrometer Tracing** e **Zipkin** para capturar *traces* de ponta-a-ponta, permitindo o diagnóstico de latência.

- **Logs Estruturados:** O Logback foi configurado para gerar **Logs Estruturados (JSON)**, incluindo o **TraceID** e **SpanID**, permitindo a correlação de dados.

3.3. Evidência de Qualidade por Testes Automatizados e CI

- **Sucesso dos Testes:** A execução bem-sucedida dos **Testes de Integração** e **Unitários** prova que o Mock Service é acionado corretamente e que o *rollback* transacional funciona.
- **DevOps e CI/CD:** A aplicação inclui um **Dockerfile Multi-Stage Build** e o `ci-cd.yml` (GitHub Actions), que automatizam o *build*, a execução dos testes e a geração da imagem Docker.

V. CONCLUSÃO E ESCOPO FUTURO

5.1. Conclusão

O **Delivery API** demonstra sucesso na entrega de um MVP validado, apresentando uma arquitetura em camadas correta, segurança granular e o uso proficiente de ferramentas de infraestrutura e qualidade de *software*, alinhado com os **padrões de mercado**.

5.2. Escopo Futuro e Roadmap

As seguintes funcionalidades (já planejadas) serão desenvolvidas em etapas futuras:

- **Painel do Administrador:** Criação da interface e *endpoints* dedicados.
- **Autenticação Avançada:** Implementação do fluxo **"Esqueceu a Senha"**.
- **Gestão de Perfis:** Criação de *endpoints* de registro dedicados para **Cadastro do Entregador** e **Cadastro do Restaurante**.
- **Gestão de Carrinho:** Adicionar funcionalidade para **remover ou excluir produtos** do carrinho.
- **Controle de Estoque (Visual):** Mostrar aos usuários o *status* ou a quantidade disponível de produtos em estoque.
- **Integrações Externas:** Implementação da **integração via CEP** para validação de endereço e substituição do Mock Service pela **integração de Pagamento real**.

Bibliografia Recursos de Conhecimento e Implementação

Referência	Tipo	Justificativa no Projeto
Programa Qualifica - TI - Curso: Arquitetura de Sistemas.	Conteúdo Base	Estrutura de Camadas, Requisitos, Conceitos de MVP.
Documentação Oficial do Spring Framework (Spring Boot, Spring Security).	Frameworks	Implementação de JWT, RBAC, Arquitetura em Camadas.
Documentação Oficial do Micrometer, Prometheus e Zipkin.	Observabilidade	Implementação do Tracing Distribuído e Coleta de Métricas de Sistema/Negócio.
Documentação Oficial do Redis e Spring Cache.	Performance	Implementação do Cache Distribuído (Decisão de Otimização).
Documentação Oficial do JUnit 5 e Mockito.	Qualidade	Base para o desenvolvimento e validação dos Testes Unitários e de Integração.
Documentação Oficial do Docker e Docker Compose.	Infraestrutura	Containerização e Orquestração do ambiente (docker-compose.yml).