Guia do Desenvolvedor: DeliveryTech API

Este guia serve como referência técnica para desenvolvedores que desejam configurar, manter ou evoluir a API do sistema DeliveryTech.

I. ARQUITETURA, ESTRUTURA DE CÓDIGO E PADRÕES

1.1. Estrutura e Padrão Arquitetural

O Backend segue o padrão **Arquitetura em Camadas** (*Layered Architecture*), fundamental para a manutenibilidade. Esta organização é visível nos pacotes do código:

Pacote	Exemplo de Classe	Padrão / Responsabilidade
controller	**PedidoController**	**Camada de Apresentação (Endpoints REST).**
service	**PedidoServiceImpl, AuthService**	**Regra de Negócio e Transacional (Camada de Serviço).**
repository	**PedidoRepository**	**Acesso a Dados (Spring Data JPA).**
entity / dto	**Pedido.java, LoginRequest.java**	**Modelos de Domínio e Objetos de Transferência de Dados.**
validation	**CpfValidator.java**	**Lógica de validação customizada e regras de negócio.**

1.2. Padrões de Projeto e Qualidade

- **Controle de Acesso (RBAC):** Autorização granular via **Spring Security** e **JWT**. O sistema suporta perfis **CLIENTE**, **RESTAURANTE**, **ENTREGADOR** e **ADMIN**.
- **Mock Service Dedicado:** O `PaymentService` é um *mock* utilizado para simular serviços externos de pagamento, demonstrando o princípio de **desacoplamento**.

- **Documentação Automática:** A API é documentada em tempo real via **Swagger/OpenAPI**.

II. INFRAESTRUTURA E OBSERVAÇÃO AVANÇADA

2.1. Containerização Completa (Docker Compose)

O ambiente completo (API, MySQL, Redis, Zipkin) é replicável via Docker Compose, garantindo a portabilidade e a execução idêntica em qualquer ambiente.

1. **Comando de Inicialização:** Execute na raiz do projeto (`delivery-api/`):

   Bash

   docker-compose up --build

- **Links Úteis:** Swagger UI (`http://localhost:8080/swagger-ui.html`), Zipkin Tracing (`http://localhost:9411`), Endpoint de Saúde (`/actuator/health`).

2.2. Performance e Observabilidade (RFNs)

O projeto aborda requisitos não funcionais (RFNs) críticos:

- **Performance: Redis** e **Spring Cache** (`@Cacheable`) são usados para cache distribuído, reduzindo a latência e a carga do banco de dados.
- **Distributed Tracing: Micrometer Tracing** e **Zipkin** capturam *traces* de ponta-a-ponta, permitindo o diagnóstico de latência e o rastreamento de requisições.
- **CI/CD:** O arquivo `.github/workflows/ci-cd.yml` configura a automação do *build*, a execução dos testes e a geração da imagem Docker.

III. GUIA DE CONTRIBUIÇÃO E TESTES

3.1. Práticas de Contribuição

1. **Estrutura:** Mantenha a separação em camadas (`controller` deve chamar apenas `service`).
2. **Documentação:** Mantenha o código limpo, modular e bem documentado, conforme a prática de **Manutenibilidade**.

3. **Onboarding:** Certifique-se de que o `README.md` contenha as instruções claras de instalação e uso.

3.2. Estratégia de Testes

O projeto conta com uma suíte robusta de testes para garantir a qualidade:

- **Execução Total:** Utilize `./mvnw clean install` para rodar todos os testes (Unitários e de Integração).
- **Validação:** A suíte de testes verifica tanto as **Funcionalidades Testadas** quanto os **Fluxos sem Bugs**. O teste de integração para a falha de pagamento garante a **Confiabilidade** e o *Rollback* transacional.

3.3. Dados de Acesso Padrão (Ambiente de Teste)

Para testar a API, use as seguintes credenciais:

Usuário	E-mail	Role
Administrador	admin@delivery.com	ADMIN
Cliente	joao@email.com	CLIENTE
Cliente Secundário	maria@email.com	CLIENTE
Restaurante Padrão	pizza@palace.com	RESTAURANTE
Restaurante Secundário	burger@king.com	RESTAURANTE
Entregador	carlos@entrega.com	ENTREGADOR

Utilize estas credenciais para obter o token JWT via `POST /api/auth/login` e testar os endpoints.

2.