### Documentação Técnica - Sistema de Pagamentos

#### 1. Visão Geral

O sistema de pagamentos é uma solução baseada em microserviços desenvolvida para suportar múltiplos métodos de pagamento (Pix, boleto, cartão de crédito/débito) e integração com plataformas externas como Asaas, além de serviços internos como autenticação, gateway e autorização. A arquitetura modular visa escalabilidade, alta disponibilidade e facilidade de manutenção.

#### 2. Arquitetura

O sistema segue o padrão de microserviços desacoplados, onde cada serviço é responsável por um domínio específico. A comunicação entre os serviços ocorre via REST e mensageria assíncrona (Kafka), conforme o caso. Os serviços estão organizados em pastas independentes com configuração, testes e dependências próprias.

\*\*Principais tecnologias:\*\*

\* Java 8+

\* Spring Boot (Web, Data, Security, Cloud)

\* Apache Kafka

\* AWS (S3, SNS, SQS)

\* Docker, Kubernetes

\* GitHub Actions (CI/CD)

#### 3. Estrutura do Projeto

\* `/pix-service`, `/boleto-service`, `/card-service`: Serviços de pagamento específicos.

\* `/gateway-service`: Roteia e orquestra chamadas entre serviços externos e internos.

\* `/auth-service`: Responsável pela autenticação e emissão de tokens JWT.

\* `/asaas-service`: Integração com a API do Asaas.

\* `/common`: Módulos compartilhados entre serviços (ex: DTOs, utilitários, validações).

\* `/observability`, `/security`, `/messaging`: Submódulos para rastreamento, segurança e mensageria.

\* `/deployment`: Arquivos Docker e YAML para deploy.

#### 4. Observabilidade

\* \*\*Logs:\*\* Implementação com SLF4J e logs estruturados em JSON.

\* \*\*Tracing:\*\* Integração com OpenTelemetry para rastreamento de chamadas entre serviços.

\* \*\*Métricas:\*\* Expostas via Actuator/Micrometer e coletadas pelo Prometheus.

#### 5. Mensageria

\* Uso do Apache Kafka para eventos como "Pagamento Criado", "Pagamento Aprovado", etc.

\* Tópicos definidos centralizadamente na classe `KafkaTopics.java`.

\* Serviços atuam como produtores e consumidores conforme o contexto.

#### 6. Resiliência

\* Configuração com Resilience4j para circuit breaker, retry e rate limiter.

\* Fallbacks e timeouts configuráveis por serviço.

\* Estratégias para isolamento de falhas e degradação controlada.

#### 7. Segurança

\* Comunicação entre serviços com TLS (keystore/truststore).

\* Integração com HashiCorp Vault para gerenciamento de segredos.

\* Autenticação via OAuth2 e tokens JWT (no `auth-service`).

#### 8. Deploy

\* Cada serviço possui `Dockerfile` e `docker-compose.override.yml`.

\* Diretório `/deployment/k8s` contém manifestos prontos para Kubernetes.

\* Separação de ambientes via `application-dev.yml`, `application-prod.yml`, etc.

#### 9. Testes

\* Testes unitários e de integração organizados por serviço.

\* Mocks com Mockito, testes de contrato com Spring Cloud Contract (opcional).

\* Pipeline CI/CD com validações automáticas (build, testes, análise estática).

#### 10. Extensibilidade

\* Arquitetura plugável permite adicionar/remover métodos de pagamento sem impactar o core.

\* Serviços expostos via interfaces REST padronizadas.

\* Integrações com terceiros centralizadas em serviços específicos.

#### 11. Referência Rápida

\* Scripts de build: `./mvnw clean install -DskipTests`

\* Subir localmente: `docker-compose up`

\* Executar testes: `./mvnw test`

\* Endpoints principais descritos em `/swagger-ui.html` de cada serviço.

---

> \*Esta documentação será convertida em PDF e anexada na raiz do projeto na pasta `/docs`, para consulta técnica do cliente e do time.\*