**Documentação Técnica - Sistema de Pagamentos**

**1. Visão Geral**

* Objetivo do sistema
* Benefícios da arquitetura modular

**2. Arquitetura**

* Diagrama de microserviços
* Tecnologias utilizadas
* Comunicação entre serviços (REST/Kafka)

**3. Estrutura do Projeto**

* Organização por serviços (ex: auth-service, pix-service)
* Diretórios comuns (ex: common, deployment, cloud-aws)

**4. Observabilidade**

* Logs estruturados
* Tracing com OpenTelemetry
* Métricas com Prometheus/Micrometer

**5. Mensageria**

* Uso de Kafka (produtor/consumidor)
* Tópicos padronizados no KafkaTopics.java

**6. Resiliência**

* Configuração com resilience4j
* Circuit breaker, retry, rate limiter

**7. Segurança**

* TLS com keystore/truststore
* Vault para secrets
* JWT (se implementado no auth-service)

**8. Deploy**

* Dockerfile e docker-compose
* Kubernetes com YAMLs prontos
* Profiles para dev/prod

**9. Testes**

* Cada serviço com testes próprios
* Pastas tests/<nome-do-serviço>
* Integração contínua com GitHub Actions

**10. Extensibilidade**

* Plug & Play: serviços independentes
* Integração facilitada com plataformas externas (Asaas, PagBank, etc.)

**11. Referência Rápida**

* Scripts, comandos úteis, endpoints principais

**Qual o fluxo de funcionamento?**

1. **Cada microserviço funciona de forma independente**, com sua própria aplicação Spring Boot (Application.java), controller, service, model e repositório.
2. **O api-gateway** faz o roteamento das requisições para os serviços apropriados (como Auth, Payment, Pix etc).
3. **O payment-service atua como orquestrador**, ou seja, ele recebe a requisição de pagamento e decide qual serviço será acionado (boleto, pix, cartão, etc), provavelmente via chamada REST ou mensageria (Kafka).
4. Os dados fluem entre os microserviços através de **DTOs padronizados** e serializados (REST ou Kafka).

**Os microserviços aproveitam a mesma observabilidade e portal?**

**Sim**, e isso é um dos grandes diferenciais dessa arquitetura:

* O pacote common/observability traz classes como TracingConfig.java, LoggingAspect.java, MetricsConfig.java, que podem ser incluídas como **dependência comum** em todos os serviços.
* Todos os serviços compartilham:
  + **Tracing**: OpenTelemetry ou Sleuth + Jaeger
  + **Logs estruturados**: via AOP
  + **Métricas**: Micrometer + Prometheus
  + **Health Checks**: padrão Spring Actuator + customizados (ReadinessProbe.java, LivenessProbe.java)

Esses dados vão para um **painel central** (ex: Grafana, Prometheus, Jaeger) e você consegue monitorar o sistema inteiro com visibilidade total.

**E os testes? São por microserviço ou gerais?**

**Separados por microserviço** — e isso está correto.

* Cada serviço tem sua própria pasta de testes (ex: tests/auth-service, tests/payment-service).
* Isso permite rodar testes isolados, com CI/CD individual por serviço.
* Tipos de testes recomendados por serviço:
  + **Unitários** (mock de dependências)
  + **Integração** (banco, mensageria, outros serviços)
  + **Contratos** (pact ou OpenAPI validator)
  + **End-to-end** (pode ser separado, com containers ou ambiente dev)

**Como usar essa estrutura a partir de agora?**

1. **Abrir no Eclipse ou VS Code (com suporte a Maven)**.
2. Começar implementando a lógica em cada TODO criado.
3. Iniciar os testes (TDD, se possível).
4. Usar Docker para rodar localmente com docker-compose.yml.
5. Criar pipelines no GitHub Actions (CI/CD já esboçado).
6. Monitorar com Prometheus/Grafana/Jaeger, se configurado.