Desafio Técnico – Desenvolvedor Java Sênior

O objetivo deste desafio é avaliar sua capacidade de projetar, implementar e documentar uma API resiliente, performática e bem estruturada em Java, dentro de um contexto realista de integração com múltiplos sistemas externos.

Contexto

Você deve implementar o endpoint **GET /api/v1/veiculos/{idveiculo}/analise**, responsável por realizar uma análise unificada de dados veiculares a partir de múltiplas fontes externas (SOAP e REST). O identificador de entrada `{idveiculo}` pode ser placa, RENAVAM ou VIN, devendo ser normalizado para o VIN — considerado o identificador canônico.

Regras de Negócio

1. Identificar o tipo do identificador recebido (placa, RENAVAM ou VIN) e convertê-lo em VIN.
2. Consultar as integrações F1 (SOAP) e F3 (REST) **em paralelo**.
3. Caso a resposta de F1 indique `restricoes.renajud = true` ou `restricoes.recall = true`, chamar também a integração F2 (REST).
4. Consolidar todos os dados em uma resposta única e registrar o log da consulta no banco de dados.

Requisitos Não-Funcionais

- **Tráfego**: 120k/dia; pico de 10k/h entre 14h e 17h.
- **SLOs**: P95 ≤ 500ms sem F2; P95 ≤ 900ms com F2.
- **Rate limit F1**: 2 RPS. Demonstrar a estratégia (por processo, nó ou global).
- **Resiliência**: aplicar timeouts, retry com jitter, circuit breaker e bulkheads.
- **Observabilidade**: tracing, métricas (latência, erros, custo) e logs estruturados (JSON).
- **Segurança**: JWT, redaction de PII e segredos externos.
- **Idempotência**: uso de header `Idempotency-Key`.

Contrato da API (OpenAPI 3)

Forneça o arquivo `openapi.yaml` com os schemas:
- `VehicleAnalysis`

- `SupplierStatus`
- `Constraints`
- `Infractions`

A resposta deve conter dados parciais quando houver falhas, incluindo status de cada fornecedor e métricas de latência.

Persistência

Criar a tabela `vehicle_analysis_log` com os campos:
- id (UUID)
- timestamp
- idInputType / idInputValue
- vinCanonical
- supplierCalls (JSONB)
- hasConstraints
- estimatedCostCents
- traceId

Arquitetura

Utilizar arquitetura limpa (hexagonal):
- **api**: controllers, OpenAPI
- **application**: use cases e orquestração
- **domain**: modelos canônicos
- **infrastructure**: adapters SOAP/REST, config, mapeadores

Testes

- Unitários: mappers, validações VIN/placa/RENAVAM
- Integração: WireMock (F2/F3), MockWebServiceServer (F1)
- Persistência: Testcontainers
- Carga: Gatling/JMeter simulando 10k requisições/hora e respeitando 2 RPS para F1

Entrega

Entregue um projeto Gradle com:
- Dockerfile e docker-compose (mocks + DB)
- `openapi.yaml`
- Coleção Postman e scripts `.http`

- README com decisões arquiteturais e trade-offs

Critérios de Avaliação (100 pontos)

- Modelagem canônica e desambiguação VIN (15)
- Arquitetura limpa e separação de camadas (15)
- Resiliência (timeouts, retry, breaker, rate-limit) (20)
- Observabilidade (tracing, métricas, logs) (10)
- Contrato OpenAPI e resposta parcial (10)
- Testes (unit, integração, carga) (15)
- Segurança e idempotência (10)
- Clareza do README e scripts (5)