Seleção VLAB - Backend

1. O Objetivo

Este desafio foi desenhado para candidatos à vaga de estágio/graduação na V-Lab, focada no time de **Backend** do projeto do **setor de transportes**. O objetivo é avaliar a capacidade do estudante de construir APIs robustas, lidar com dados sensíveis e orquestrar ambientes de desenvolvimento utilizando **Docker**.

Nível Esperado (Estágio/Júnior): Avaliaremos sua lógica de programação, organização de código, entendimento de protocolos e capacidade de entregar um ambiente configurado.

2. O Desafio

O setor de transportes está construindo um "Data Lake" para centralizar informações de abastecimento da frota nacional. Sua missão é construir a API de Entrada (Gateway) que receberá milhares de requisições de postos de gasolina e sistemas embarcados nos veículos.

Precisamos garantir que os dados recebidos sejam válidos, persistidos com segurança em um banco relacional e que a estrutura do banco seja versionada. Além disso, a aplicação deve ser capaz de suportar alta demanda de leitura para os dashboards de monitoramento.

Stack Obrigatória:

- **Linguagem:** Python (3.11+).
- **Framework:** FastAPI.
- **API:** REST ou GraphQL (Sua escolha).
- **Infraestrutura:** Docker e Docker Compose.

Stack Desejável (Diferenciais):

1. **Banco de Dados:** PostgreSQL e Redis.
2. **Migrations:** Alembic.
3. **Qualidade:** Pytest, Linters (Black/Ruff).

3. Os Requisitos do Projeto

Essas são as funcionalidades que devem ser implementadas, faça o que for possível.

A. Ingestão de Dados (Escrita)

- **Regra de Negócio (Flag de Anomalia):** Ao salvar o registro, o sistema deve verificar se o preço informado é **25% superior** à média histórica daquele tipo de combustível (pode ser uma média fixa mockada ou calculada no banco). Se for, marcar o registro com uma flag improper_data = true.

- **Endpoint:** POST /api/v1/abastecimentos
- **Payload:** Deve receber um JSON contendo:
  - id_posto (Inteiro)
  - data_hora (ISO 8601)
  - tipo_combustivel (Enum: GASOLINA, ETANOL, DIESEL)
  - preco_por_litro (Decimal/Float)
  - volume_abastecido (Decimal/Float)
  - cpf_motorista (String - Deve ser validado).

B. Consulta e Relatórios (Leitura)

- **Endpoint:** GET /api/v1/abastecimentos
  - Deve permitir paginação (page e size).
  - Filtros por tipo_combustivel e data.
- **Endpoint:** GET /api/v1/motoristas/{cpf}/historico
    - Retorna todos os abastecimentos de um motorista específico.

C. Script de Carga (Teste de Stress)

Como precisamos validar a performance da API, você deve criar um script separado (ex: load_data.py) utilizando a biblioteca **Faker**.

- O script deve gerar e enviar **X requisições** para a sua API via HTTP.
- Deve gerar dados aleatórios, mas válidos (CPFs válidos, datas coerentes).
- Isso pode ser mais um dos serviços que sobem junto com o Docker Compose, de forma que vai ficar alimentando o backend a cada X tempos, mas assim como as demais funcionalidades, apenas é necessário fazer o possível no tempo estipulado.

D. Diferenciais Técnicos (Bônus)

Estes itens demonstram maturidade técnica e cuidado com a qualidade do software. Escolha os que conseguir implementar:

1. **Testes Automatizados (Pytest):** Implementar testes unitários simples. Ex: testar se a validação de CPF funciona ou se a flag de anomalia é marcada corretamente.
2. **Padronização de Código (Linters):** Configurar ferramentas como **Black**, **Isort** ou **Ruff** para garantir a formatação automática do código.
3. **Health Check:** Criar um endpoint GET /health que retorne status 200 e informações básicas da API (ex: versão, status do banco).
4. **Autenticação:** Proteger a rota de Ingestão (POST) com um Token simples ou API Key.

4. Requisitos Técnicos e Arquitetura

A. Python & FastAPI (Obrigatório)

- **Pydantic:** Uso extensivo de modelos Pydantic para validação rigorosa dos dados de entrada (Data Types, Validação de CPF).
- **Assincronismo:** Uso de async/await nos controllers e acesso ao banco.
- **Tipagem:** Uso de Type Hints em todo o código.

B. Versionamento de Banco (Alembic)

- Utilize o **Alembic** para gerenciar a criação das tabelas e versionamento do esquema.
- **Infra:** Configure o Alembic para rodar as migrações automaticamente na inicialização do container.

C. Qualidade de Código e Arquitetura

- **Clean Code:** Nomes de variáveis consistentes, funções pequenas e com responsabilidade única.
- **Estrutura:** Separe seu código em camadas lógicas (ex: routers, services, repositories, models). Não faça tudo em um arquivo main.py.
- **Escolha Arquitetural:** Você é livre para escolher o padrão que julgar melhor (MVC, Clean Architecture, etc.). Esteja preparado para justificar sua escolha verbalmente.

5. A Entrega

- **Prazo:** 1 semana (7 dias corridos).
- **Formato de Entrega:** Link para repositório público (GitHub/GitLab).

6. Algumas Instruções

Sobre Entregas Parciais

Não é obrigatório entregar o projeto 100% completo. Priorize a qualidade do que for entregue: é melhor ter um dashboard bem estruturado e fiel ao design, mesmo que falte uma tela de detalhe, do que entregar tudo com bugs e código desorganizado. Avaliaremos o seu potencial e a qualidade da sua implementação até onde você conseguiu chegar.

Política de Uso de Inteligência Artificial (IA)

O uso de IA é **permitido e encorajado**.

- **Condição:** Na entrevista técnica, você deverá explicar as decisões tomadas. Ex: Se a IA gerou uma query SQL, você deve saber explicar como ela funciona.