# API de Sincronização de Dados - Consinco para Plusoft

## 1. Visão Geral

Este projeto consiste em uma API desenvolvida com **FastAPI** que atua como uma ponte (middleware) entre o banco de dados Oracle do sistema **Consinco** e a API de Carga Transacional da **Plusoft**.

O objetivo principal é extrair dados de entidades de negócio (Filiais, Produtos, Clientes e Vendas), aplicando lógicas de otimização e transformação, e enviá-los de forma segura e eficiente para os endpoints correspondentes da API externa.

## 2. Arquitetura e Estrutura de Pastas

A aplicação foi estruturada seguindo princípios de **Separação de Responsabilidades** para garantir um código limpo, reutilizável e de fácil manutenção.

- **`main.py`**: Ponto de entrada da aplicação. Responsável por inicializar o FastAPI, registrar os roteadores e gerenciar o ciclo de vida da aplicação (startup/shutdown) através do `lifespan`.
- **`config/`**: Contém os arquivos de configuração.
    - `banco.py`: Configuração da conexão com o banco de dados Oracle e gerenciamento de sessões do SQLAlchemy.
    - `ambiente.py`: Centraliza as URLs dos endpoints da API externa e outras variáveis de ambiente.
- **`modelos/`**: Define a estrutura das tabelas do banco de dados como modelos do SQLAlchemy (ORM). **Detalhe:** Cada modelo aqui é um espelho fiel da estrutura da tabela/view no Oracle.
- **`servicos/`**: Camada de lógica de negócio. É aqui que reside a inteligência da aplicação.
    - **Busca de Dados:** Contém as funções que sabem como consultar o banco de dados. Implementa lógicas de **paginação** (ex: `buscar_produtos_por_pagina`) para as rotas `GET`.
    - **Formatação e Mapeamento:** Contém as funções "tradutoras" (ex: `formatar_para_api_externa`) que convertem os objetos do banco para o formato JSON (payload) esperado pela API de destino, resolvendo divergências de nomes de campos.
- **`rotas/`**: Define os endpoints da API (`@app.get`, `@app.post`).
    - **`rotas/gets/`**: Define os endpoints `GET` para consulta de dados. Eles chamam os serviços de busca **paginada** para garantir respostas rápidas e seguras.
    - **`rotas/posts/`**: Define os endpoints `POST` para acionar a sincronização. Eles implementam a lógica de **processamento em lotes (batch processing)** para lidar com grandes volumes de dados sem sobrecarregar a memória.

## 3. Configuração do Ambiente

### Pré-requisitos

- Python 3.10+
- Poetry (https://python-poetry.org/) para gerenciamento de dependências.
- Acesso ao banco de dados Oracle (Consinco).
- Credenciais e URLs da API de destino (Plusoft).

### Instalação

1. **Clone o repositório:**

```
git clone https://github.com/MichaelOli/Api_Carga_Transacional
cd Api_Carga_Transacional
```

2. **Instale as dependências com Poetry:** O projeto utiliza Poetry para gerenciar suas dependências, que estão listadas no arquivo `pyproject.toml`. O Poetry criará automaticamente um ambiente virtual e instalará tudo o que for necessário.

   Execute o comando na raiz do projeto:

```
poetry install
```

3. **Configure as variáveis de ambiente:** Certifique-se de que o arquivo `config/ambiente.py` (Deixei um arquivo `.env` de exemplo para poder ser utilizado) contém as chamadas para as URLs corretas da API de destino.

### Executando a Aplicação

Para iniciar o servidor web da API usando o ambiente virtual gerenciado pelo Poetry, execute:

```
poetry run uvicorn main:app --reload
```

- `poetry run`: Garante que o comando seja executado com o Python e as bibliotecas do ambiente virtual do projeto.
- `--reload`: Faz com que o servidor reinicie automaticamente após qualquer alteração no código.

A API estará disponível em `http://servidor_que_tiver_rodando:8000` e a documentação interativa em `http://servidor_que_tiver_rodando:8000/docs`.

# 4. Endpoints da API (Otimizados)

### Filiais, Produtos, Clientes e Vendas

Para cada entidade, existem dois tipos de endpoints com otimizações específicas:

- `GET /{entidade}`
    - **Descrição:** Retorna uma lista **paginada** de dados. Ideal para testes, depuração e consumo por interfaces de front-end.
    - **Parâmetros de Query:**
        - `pular: int` (padrão: 0): O "offset", ou seja, o número de registros a pular do início.
        - `limite: int` (padrão: 100, máx: 500): O número de registros a retornar por página.
    - **Exemplo de Uso:** `GET /produtos?pular=100&limite=50`
        - **O que faz:** Pula os primeiros 100 registros e retorna os 50 seguintes (do 101 ao 150).
        - **Resultado:** Efetivamente, retorna a **terceira página** de resultados, assumindo que cada página tem 50 itens.
- `POST /{entidade}/sincronizar`
    - **Descrição:** Inicia o processo de sincronização de **todos** os dados da entidade. Este endpoint foi projetado para lidar com uma quantidade massiva de registros.
    - **Otimização: Processamento em Lotes (Batch Processing)**. A aplicação lê os dados do Oracle em "pedaços" (1000 de cada vez), processa e envia cada lote antes de carregar o próximo. Isso mantém o uso de memória baixo e constante, prevenindo que a aplicação caia por sobrecarga.

# 5. Padrões de Arquitetura Implementados

### Padrão de Mapeamento (Tradução de Dados)

Um dos desafios enfrentados neste projeto de integração foi a divergência entre os nomes de campos no banco de dados de origem e os nomes esperados pela API de destino. Este projeto adota um padrão de arquitetura de **Mapeamento/Tradução** para resolver isso de forma limpa e desacoplada.

#### Parte 1: O Modelo é o Espelho do Banco

A classe do modelo SQLAlchemy (ex: `Produto` em `modelos/tb_produto.py`) é um reflexo 100% fiel da estrutura da tabela ou view do banco de dados que ela representa.

**Observação:** Se o banco de dados muda, o modelo SQLAlchemy muda junto.

```
# Arquivo: modelos/tb_produto.py
class Produto(Base):
    # O nome do atributo corresponde EXATAMENTE ao nome da coluna no Oracle.
    TXTDESCRICAOREDUZIDA = Column(String)
```

#### Parte 2: O Serviço é o Tradutor para a API Externa

A função de formatação na camada de serviço (ex: `formatar_para_api_externa`) atua como um **Mapeador**. A responsabilidade dela é pegar o objeto do nosso modelo interno e criar o dicionário (payload) no formato exato que a API de destino espera, realizando a "tradução" dos nomes dos campos.

**Observação:** Se a especificação da API de destino muda, a função de formatação muda junto.

```
# Arquivo: servicos/servicos_produtos.py
def formatar_para_api_externa(produto: Produto) -> dict:
    carga = {
        # A chave do dicionário é o nome do campo na API Externa.
        "TXTDESCRICAORESUMIDA": formatar_valor(
            # O valor vem do atributo do nosso modelo, que reflete o banco.
            produto.TXTDESCRICAOREDUZIDA
        )
    }
    return carga
```

### Proteção da API (Rate Limiting)

Para prevenir sobrecarga por excesso de requisições (seja por um robô ou múltiplos usuários), a API pode ser protegida com um limitador de taxa. A biblioteca `slowapi` pode ser integrada para limitar o número de chamadas que um mesmo IP pode fazer a um endpoint em um determinado período (ex: 30 requisições por minuto). **Preparado mas não implementado, verificar com o tio elick e o Raionny se é pra usar este recurso nas rotas GET**

# 6. Sincronização Automática (Processo em Segundo Plano)

Conforme a arquitetura definida, a sincronização automática **não deve ser iniciada pelas rotas `POST` em produção**. As rotas `POST` servem para acionamento manual. Uma vez que a ideia central do projeto se baseia nos principios da separação de responsabilidades.

# 7. A sincronização em produção foi feito por um script independente agendado.

1. **Script:** `agendador/monitor.py` que importa e utiliza as funções da camada de serviço:
2. **Mantenha o script em execução:** Após a execução, o terminal exibirá a mensagem `Agendador iniciado. Aguardando chamadas programadas...`. O script precisa permanecer em execução em segundo plano para que o agendador funcione, **geralmente vejo voces executando com o nohup nos servidores**.

### Agendador de Sincronização de API

Este script em Python é um agendador de tarefas, projetado para chamar periodicamente um conjunto de endpoints de API para sincronização de dados. Ele utiliza `apscheduler` para o agendamento, `httpx` para chamadas de API assíncronas e `loguru` para um sistema de logging detalhado e organizado.

# Funcionalidades Principais

- **Agendamento Diário:** Configurado por padrão para executar as tarefas de sincronização todos os dias à 01:00 da manhã (fuso horário de `America/Sao_Paulo`).
- **Chamadas Assíncronas:** Usa `httpx` para fazer chamadas `POST` assíncronas às APIs, garantindo que o processo seja eficiente e não bloqueie a execução.
- **Logging Detalhado:** Cria logs para cada execução de rota, registrando o início da chamada, a resposta da API (sucesso ou erro) e quaisquer exceções que ocorram.
- **Estrutura de Logs Organizada:** Os arquivos de log são salvos em uma estrutura de pastas hierárquica e intuitiva: `Logs/NomeDaRota/Mês/Dia/`, facilitando a consulta e a depuração.
- **Captura de Saída do Terminal:** Redireciona a saída padrão (`sys.stdout`) durante a chamada da API para garantir que todas as mensagens impressas no console (ex: `print()`) sejam também registradas no arquivo de log correspondente.

# Estrutura de Logs

O script cria automaticamente uma estrutura de diretórios para armazenar os logs de forma organizada, facilitando a navegação e a análise posterior.

A estrutura gerada é a seguinte:

```
Logs/
├── Rota_Clientes/
│   └── Julho/
│       └── 15/
│           └── 15_07_2024_01h00_Rota_Clientes.log
├── Rota_Filial/
│   └── Julho/
│       └── 15/
│           └── 15_07_2024_01h00_Rota_Filial.log
├── Rota_Produtos/
│   └── Julho/
│       └── 15/
│           └── 15_07_2024_01h00_Rota_Produtos.log
└── Rota_Vendas/
    └── Julho/
        └── 15/
            └── 15_07_2024_01h00_Rota_Vendas.log
```

# Pré-requisitos

Para executar este script, você precisará do Python 3.7+ e das seguintes bibliotecas. Você pode instalá-las usando `pip`.

- `httpx`: Para realizar as chamadas HTTP assíncronas.
- `loguru`: Para uma gestão de logs simples e poderosa.
- `apscheduler`: Para o agendamento das tarefas.
- `pytz`: Para lidar com fusos horários.

Dependencias necessarias contidas no `requirements.txt` com o seguinte conteúdo:

```
httpx
loguru
apscheduler
pytz
```

E instale as dependências com o comando:

```
pip install -r requirements.txt
```

# Como Usar

1. **Instale as dependências** conforme descrito na seção de pré-requisitos.
2. **Execute o script** a partir do seu terminal:

   ```
   python monitor.py
   ```

# Configuração

Rotas que serão acionadas:

- **URLs das Rotas:** Para alterar, adicionar ou remover APIs a serem chamadas, modifique a lista `rotas` dentro da função `chamar_rotas()`:

  ```
  rotas = [
      {"url": "[http://127.0.0.1:8000/filiais/sincronizar](http://127.0.0.1:8000/filiais/sincronizar)", "nome": "Rota_Filial"},
      {"url": "[http://127.0.0.1:8000/clientes/sincronizar](http://127.0.0.1:8000/clientes/sincronizar)", "nome": "Rota_Clientes"
      # Adiciona ou remove as rotas aqui separando o dicionario por virgula.
  ]
  ```

- **Horário do Agendamento:** O horário é definido na função `configurar_agendador()`. Altere os parâmetros `hour` e `minute` na chamada `agendador.add_job()` para ajustar a programação.

```
# Agendar para rodar todo dia às 01:00 no fuso de Brasília
agendador.add_job(chamar_rotas, "cron", hour=1, minute=0)
```