Descrição do Case – Processo Seletivo Dev Full-Stack

Contexto

Um escritório de investimentos precisa de uma aplicação **100 % containerizada** para:

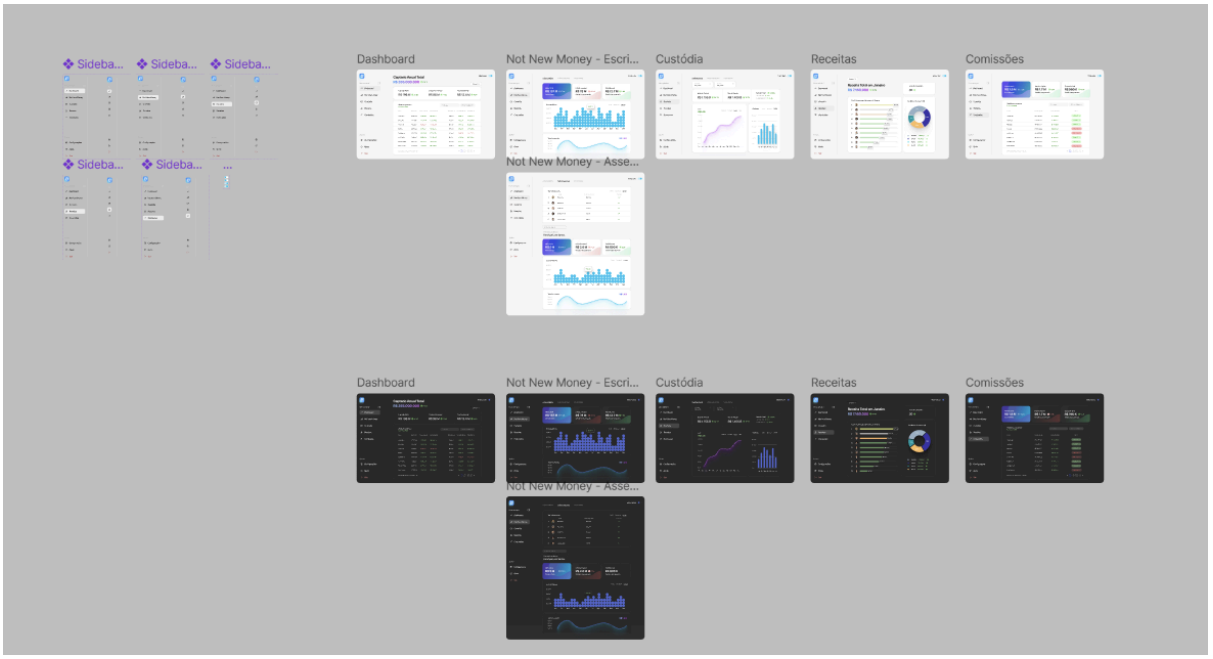
1. Cadastrar e gerenciar seus clientes;
2. Registrar as alocações de cada cliente em diferentes ativos financeiros;
3. Acompanhar, em tempo quase-real, a rentabilidade diária desses ativos usando dados públicos.

Você deve entregar **dois repositórios** (backend e frontend) prontos para subir com **Docker Compose**.

Link do Figma

https://www.figma.com/design/H04ce9DQqoLYaTuGcXrL9m/Untitled?node-id=0-1&t=6JZjAtukdYCOkScY-1

Essa parte do Figma. A parte acima da desse print é antiga.



A interface precisa ficar idêntica ao Figma e ser responsiva para zoom-in e zoom-out em desktops (mobile não é obrigatório).

Funcionalidades Obligatorias

1. Clientes

- **CRUD completo** (nome, e-mail, status ativo/inativo).
- Paginação, busca (by name/email) e filtro por status.

2. Ativos financeiros

- Cadastro de alocação por cliente (**ticker**, quantidade, preço de compra, data da compra).
- A lista de ativos disponíveis deve vir dinamicamente de uma chamada à API do **Yahoo Finance**.
- Cada alocação deve exibir: preço atual, variação % diária e rentabilidade acumulada desde a compra.

3. Rentabilidade diária

- **Tarefa agendada** (ex.: Celery Beat a cada 24 h) que:
    1. Consulta o preço de fechamento de ontem na Yahoo Finance;
    2. Atualiza uma tabela `daily_returns` (asset_id, date, close_price).
- Endpoint **`/clients/{id}/performance`** devolve a curva de rentabilidade do cliente (acumulada diária).

4. Exportação

- Endpoint CSV/Excel com o resumo das posições e rentabilidades.

Requisitos Técnicos

Backend – Python ^{311} ****

- **Framework:** FastAPI (async) + Uvicorn.
- **ORM:** SQLAlchemy 2 (com **Alembic** para migrações).
- **Validação:** Pydantic v2.
- **Autenticação básica** (JWT) com perfis admin e read-only.
- **Tarefas assíncronas:** Celery + Redis.

- **Testes:** Pytest (cobertura mínima 80 %).
- **Linters:** Ruff / Black.
- **Database:** PostgreSQL 15 (container).

Desafios de lógica

1. Calcular rentabilidade diária (IRR simples) usando série de preços.
2. Implementar caching de preços com TTL de 1 h (Redis).
3. Lidar com rate-limit da Yahoo Finance (back-off exponencial).
4. Endpoint de streaming via WebSocket que envia atualização de preços em tempo real a cada 5 s para o dashboard.

Frontend – Next.js 14 (App Router) + TypeScript

- UI baseada em **ShadCN/UI** (Button, Card, Table, etc.).
- **TanStack Query** para fetch / cache (inclua invalidation).
- **React-Hook-Form** + **Zod** para formulários.
- **Axios** para chamadas REST e WebSocket para preços.
- Tema claro/escuro (toggle).

Docker Compose (um arquivo)

```
services:
  db:
    image: postgres:15
    environment:
      POSTGRES_USER: invest
      POSTGRES_PASSWORD: investpw
      POSTGRES_DB: investdb
    volumes:
      - db_data:/var/lib/postgresql/data
  redis:
    image: redis:7
  backend:
    build: ./backend
```

```
    depends_on:
      - db
      - redis
    environment:
      DATABASE_URL: postgresql+asyncpg://invest:investpw@db/investdb
      REDIS_URL: redis://redis:6379/0
  frontend:
    build: ./frontend
    depends_on:
      - backend
    ports:
      - "3000:3000"
volumes:
  db_data:
```

Entregáveis

1. **Repositório Backend** com:

   - Código, testes, Dockerfile, README.
   - Migração inicial criando tabelas `clients`, `assets`, `allocations`, `daily_returns`.

2. **Repositório Frontend** com:

   - Código, Dockerfile, README. 3. Arquitetura explicada no README (fluxo, diagramas opcionais).

Prazo

7 dias corridos a partir do recebimento do case.

Dicas

- Leia a documentação oficial do FastAPI, Celery e TanStack Query.
- Escreva testes antes de codar a lógica de rentabilidade.
- Commits pequenos e mensagens claras valem pontos.
- Se ficar travado em algo, registre no README o que tentou.

Boa sorte e divirta-se! 😄