Documento de Decisões Arquiteturais (ADR)

Projeto: DataFood Analytics

Contexto

O objetivo deste projeto é resolver a dor da Maria, uma dona de restaurante que possui dados operacionais massivos, mas não consegue `explorá-los livremente` ou `responder perguntas complexas`. Ela está presa a `dashboards fixos` e ferramentas genéricas.

O desafio principal é construir uma ferramenta que seja, ao mesmo tempo, `flexível` para exploração e `simples o suficiente para um usuário não-técnico`.

Filosofia Principal de Design

A arquitetura foi guiada por dois princípios:

1. **Flexibilidade no Frontend:** O usuário deve ter controle total para "construir" suas próprias perguntas.
2. **Performance no Backend:** O "trabalho pesado" (agregação, filtragem de 1M+ de linhas) deve ser feito pelo servidor/banco, não pelo navegador do cliente.

Decisões Arquiteturais Chave

1. API: Endpoint Único de "Query Builder" (`POST /api/query`)

- **Decisão:** Em vez de uma API REST tradicional (com dezenas de endpoints como `/sales`, `/products/by-store`, etc.), optamos por um único endpoint `POST /api/query`.
- **Justificativa:** Esta é a decisão de arquitetura mais importante do projeto. Ela ataca diretamente o problema de `analytics customizável e flexível`.
    - O frontend envia um objeto JSON que descreve a análise desejada (métricas, dimensões, filtros, ordenação).
    - O backend (em FastAPI e SQLAlchemy Core) atua como um "construtor de queries", traduzindo esse JSON em uma única e eficiente consulta SQL.

- **Alternativa Rejeitada:** Uma API REST tradicional. Foi rejeitada por ser inflexível. Se a Maria quisesse um novo cruzamento de dados, isso exigiria um novo endpoint no backend, violando o critério `Sem depender de desenvolvedores`.

2. Ambiente de Desenvolvimento: Híbrido (Docker + NPM Local)

- **Decisão:** Os serviços "stateful" (com estado), como o **PostgreSQL** e a **API Backend**, rodam no Docker Compose. O serviço "stateless" (sem estado), o **Frontend React**, roda localmente (`npm run dev`).
- **Justificativa:** Esta foi uma pivotagem crítica. Tentativas iniciais de rodar o frontend no Docker levaram a uma série de problemas complexos de `node_modules`, cache do `npm` e incompatibilidades de arquitetura (Alpine vs. Glibc).
    - A abordagem híbrida nos dá o melhor dos dois mundos: a **estabilidade** do Docker para o banco de dados e a **velocidade** e o *hot-reload* instantâneo do Vite rodando nativamente no WSL para o frontend.

3. Otimização de Performance: Indexação Explícita

- **Decisão:** Após a geração dos dados (1.1M+ de vendas), executar um script SQL (`02-indices.sql`) que cria índices explícitos em todas as colunas de chave estrangeira (`store_id`, `channel_id`, `product_id`, etc.) e colunas de filtro comuns (`created_at`).
- **Justificativa:** Testes de carga iniciais mostraram que consultas de `JOIN` e `WHERE` (o núcleo da nossa API) estavam levando de 5 a 6 segundos. Isso falhava no critério de `queries rápidas`. Após a indexação, o tempo de resposta caiu para menos de 1 segundo.

4. Stack de Backend: FastAPI + SQLAlchemy Core

- **Decisão:** Usar FastAPI pela sua performance e SQLAlchemy Core (em vez do ORM completo).
- **Justificativa:** Para construir queries analíticas complexas (`GROUP BY`, `JOIN`s dinâmicos), precisávamos de controle total sobre o SQL gerado. Um ORM completo (como o do Django ou o SQLAlchemy ORM) seria restritivo e tornaria a lógica do `QueryBuilder` desnecessariamente complexa. O SQLAlchemy Core nos deu o poder de construir o SQL dinamicamente, de forma segura e performática.

5. Frontend UX: Filtros Inteligentes (Contextuais)

- **Decisão:** O componente de filtro não é apenas um campo de texto. Ele muda dinamicamente com base no campo selecionado (ex: "Canal" mostra um dropdown, "Hora" mostra um input numérico).
- **Justificativa:** Isso é crucial para o critério `simples o suficiente para usar sem treinamento técnico`. Um usuário não-técnico não pode adivinhar o ID de uma loja ou o formato de um status (`COMPLETED` vs. `Canceled`). Para resolver isso,

o frontend busca opções nos endpoints `/api/options/...` do backend, guiando o usuário e prevenindo erros.

6. Frontend UX: Estado Global de Período

- **Decisão:** O seletor de período (`DateRangePicker`) foi colocado no topo da aplicação, e seu estado (o `timeRangeFilter`) é usado para alimentar *tanto* os KPIs quanto as consultas de análise principais.
- **Justificativa:** Isso atende diretamente ao critério de `Ver overview do faturamento do mês`. O usuário pode definir um período uma vez (ex: "Este Mês") e ver todos os módulos da página (KPIs e gráficos) serem atualizados instantaneamente, permitindo `comparações temporais` de forma intuitiva.

7. Estilização do Frontend: Pivot para CSS Modules

- **Decisão:** A estratégia de estilização inicial (Tailwind CSS) foi abandonada em favor de **CSS Modules**.
- **Justificativa:** A configuração do Tailwind CSS em um ambiente Docker "híbrido" (com `postcss.config.cjs`, `package.json` dessincronizados, etc.) provou ser extremamente frágil e consumiu um tempo de desenvolvimento significativo. A prioridade era entregar uma `solução funcionando`. Fizemos uma pivotagem estratégica para CSS Modules, que é suportado nativamente pelo Vite, não requer arquivos de configuração extras e garantiu uma fundação de UI 100% estável.