Questionário de Análise e Reflexão de Código

Projeto: Sistema de Gestão Pet Shop "Amigo Fiel"

1. Análise da Lógica de Venda

a) Qual é o nome da função/método que você criou para processar a venda?

A lógica de processamento de venda está dividida em duas partes principais:

1. **Backend (Django):** A lógica de negócio principal está no arquivo `api/views.py`, dentro da `VendaViewSet`. O método que processa a venda é o `create` (que responde ao `POST`), pois ele foi sobreposto para incluir a lógica de verificação de stock.
2. **Frontend (React):** A função que captura os dados do formulário e envia a requisição para o backend chama-se `handleSubmit`, localizada no arquivo `frontend/src/pages/NovaVenda.jsx`.

b) Descreva a mensagem de erro exata (palavra por palavra) que o seu sistema exibe ao usuário se o estoque for insuficiente.

A mensagem de erro é gerada no backend (`api/views.py`) e exibida no frontend. A mensagem exata é:

```
f"Estoque insuficiente. Produto '{produto.nome}' possui apenas {produto.quantidade_estoque} unidades."
```

(Onde `{produto.nome}` e `{produto.quantidade_estoque}` são substituídos pelos valores reais do produto no momento da falha).

c) Como você garantiu que o estoque só é atualizado *depois* que a venda é 100% validada?

Isto foi garantido usando o `transaction.atomic()` do Django no backend. O fluxo é o seguinte:

1. A função `create` inicia um `transaction.atomic()`, que trata todas as operações de base de dados como uma única unidade.
2. **Primeiro, a validação:** O código faz uma verificação (`if produto.quantidade_estoque < quantidade_desejada:`).

3. **Se a validação falhar:** A função retorna uma resposta de erro (`Response(..., status=400)`). Isto interrompe a execução e o `transaction.atomic()` automaticamente executa um `ROLLBACK`, desfazendo qualquer alteração.
4. **Se a validação passar:** Só então o código prossegue para atualizar o stock (`produto.quantidade_estoque -= ...` e `produto.save()`) e registar a `Venda`.
5. Se tudo for executado sem erros, o `transaction.atomic()` faz o `COMMIT` ao final do bloco, salvando as alterações de forma permanente.

Desta forma, é impossível que o stock seja atualizado se a validação de stock falhar.

2. Decisão de Implementação (Ordenação)

a) Qual foi o algoritmo de ordenação (Ex: Bolha, Inserção, Seleção, etc.) que você implementou para a lista de produtos?

Não foi implementado um algoritmo de ordenação manual (como Bubble Sort). Foi utilizada a funcionalidade nativa do **Django REST Framework (DRF)**, o `filters.OrderingFilter`.

No frontend (`NovaVenda.jsx`), a requisição à API é feita com o parâmetro `ordering=nome`: `api.get('/produtos/?ordering=nome')`

No backend (`api/views.py`), a `ProdutoViewSet` está configurada para aceitar esta ordenação.

b) Por que você escolheu *esse* algoritmo específico?

Eu escolhi esta abordagem por ser a prática profissional mais eficiente e escalável.

Em vez de puxar todos os produtos da base de dados para a memória do backend (Python) ou do frontend (JavaScript) e *depois* aplicar um algoritmo de ordenação (o que seria muito lento com milhares de produtos), esta abordagem delega a ordenação diretamente à **base de dados** (através do comando `ORDER BY nome` em SQL).

A base de dados é altamente otimizada para operações de ordenação, tornando a resposta da API muito mais rápida e eficiente.

3. Ponto Crítico da Simulação

a) Descreva o momento em que você encontrou a maior dificuldade lógica ou o "bug" mais complicado durante o desenvolvimento.

O bug mais complicado foi o erro de **CORS (Cross-Origin Resource Sharing)**, que apareceu durante a implementação da tela de Login.

b) O que você esperava que o código fizesse e o que ele *realmente* estava fazendo?

- **O que eu esperava:** Que o frontend (React) a correr em `http://localhost:5173` (ou `5174`) enviasse os dados de login (utilizador e senha) via `POST` para o backend (Django) em `http://localhost:8000/api/api-token-auth/`.
- **O que realmente estava a fazer:** O navegador bloqueava a requisição `POST`. O backend (Django) recebia uma requisição `OPTIONS` (chamada "preflight request") e a rejeitava, pois o `localhost:5173` não estava na sua lista de permissões.

c) Qual foi o seu processo de pensamento para depurar e, finalmente, encontrar a solução?

1. O frontend exibia uma mensagem de erro genérica ("Erro ao conectar ao servidor").
2. Para descobrir a causa real, usei as **Ferramentas de Desenvolvedor (F12)** do navegador e abri a aba **"Console"**.
3. O Console mostrou o erro explícito de CORS, indicando que a origem `http://localhost:5173` (ou `5174`) tinha sido bloqueada.
4. A solução foi no **backend (Django)**: abri o `petshop_backend/settings.py` e certifiquei-me de que `corsheaders` estava em `INSTALLED_APPS` e `MIDDLEWARE`.
5. Finalmente, adicionei a origem correta do frontend à lista `CORS_ALLOWED_ORIGINS` (ex: `"http://localhost:5173"`).

4. Detalhe da Validação

Escolha uma das validações obrigatórias (ex: e-mail do cliente).

a) Em qual arquivo e linha (aproximadamente) está essa validação no seu projeto?

A validação do e-mail do cliente foi feita em **dois** locais:

1. **Frontend (React):** No arquivo `frontend/src/pages/Clientes.jsx`, dentro da função `handleFormChange` (aprox. linha 50). A lógica é: `if (!value.includes('@') || !value.includes('.')) { setErroEmail(...) }`.
2. **Backend (Django):** No arquivo `api/models.py`, na definição do `Cliente` (aprox. linha 10), ao usar `models.EmailField(unique=True)`. O Django REST Framework aplica automaticamente a validação de formato de e-mail deste campo.

b) A validação acontece no frontend ou no backend? Por que você escolheu fazer dessa forma?

A validação acontece em **ambos (Frontend e Backend)**, o que é considerado a melhor prática por duas razões distintas:

1. **Validação no Frontend:** É feita para melhorar a **Experiência do Utilizador (UX)**. Ela dá feedback *imediato* ao utilizador (ex: "E-mail inválido") antes mesmo de ele submeter o formulário, evitando uma chamada de API desnecessária.

2. **Validação no Backend:** Esta é a validação de **segurança e integridade** principal. O backend *nunca* deve confiar nos dados vindos do frontend (pois a validação do frontend pode ser facilmente contornada). A validação no backend (no `EmailField` do Django) garante que apenas dados válidos e íntegros sejam salvos na base de dados.

5. Autoavaliação e Próximos Passos

Se você tivesse mais três horas para trabalhar neste projeto, qual seria a primeira funcionalidade ou melhoria de interface que você adicionaria?

Se eu tivesse mais três horas, a primeira funcionalidade que adicionaria seria um **Dashboard** na página inicial (na rota `/`, que atualmente redireciona para `/produtos`).

Este dashboard incluiria:

1. **Cartões de Resumo:** "Total de Vendas Hoje", "Clientes Cadastrados" e "Produtos com Stock Baixo".
2. **Um Gráfico Simples:** Um gráfico de barras mostrando o "Faturamento dos Últimos 7 Dias".
3. **Lista Rápida:** Os "Top 5 Produtos Mais Vendidos".

Descreva por que você acredita que essa seria a melhoria mais importante para o usuário final (o gerente do Pet Shop).

Eu acredito que esta seria a melhoria mais importante porque o objetivo do sistema é substituir planilhas manuais, que não geram apenas erros, mas também *dificultam a visualização dos dados*.

O gerente do Pet Shop não quer apenas *registar* vendas; ele quer *entender* o seu negócio. Um dashboard transforma os dados brutos (vendas, produtos) em **informação acionável**, permitindo ao gerente tomar decisões rápidas (ex: "Preciso de comprar mais do Produto X", "As vendas esta semana estão fracas").