Documento de descrição do Projeto

1. Links importantes:

- Aplicação no Heroku: https://estoquefacil-grupo9.herokuapp.com/swagger-ui.html#/
- Repositório no GitHub: https://github.com/psoft-2020-3/projeto-psoft-grupo-9
- Pull request da entrega: https://github.com/psoft-2020-3/projeto-psoft-grupo-9/pull/33

2. Integrantes e suas atribuições do Projeto:

Quantidade de User Stories obrigatórias:

- Gabriel Brandão: 3
- Matheus Silva: 3
- Higor Araujo: 3
- Gabriel Carvalho: 2
- Matheus Alves: 2

Quais User Stories obrigatórias (escolhido através de sorteio):

- Gabriel Brandão: 1, 3, 7
- Matheus Silva: 6, 8, 11
- Higor Araujo: 4, 5, 13
- Gabriel Carvalho: 9, 12
- Matheus Alves: 2, 10

Atribuições extra-implementação:

- Matheus Silva: Revisar e organizar o documento completo
- Higor Araujo: Descrever as User Stories implementadas
- Gabriel Carvalho: Identificar e listar os Bad Smells detectados no código
- Matheus Alves: Explicar como cada Bad Smell foi resolvido
- Gabriel Brandão: Descrever principais decisões de design tomadas no desenvolvimento do projeto

3. Bad Smells:

- Classe RestApiController:
    - Método listAllUsers:
        - ➢ Bad Smell: Método lista produtos e não usuários.
        - ✓ Solução: Renomeação do método para listAllProdutos.
    - Método criarProduto:
        - ➢ Bad Smells: Método longo e com código duplicado.
        - ✓ Soluções: Utilizar o método doesProdutoExist, criar método para mudar a situação e remover código desnecessário.
    - Método consultarProduto:
        - ➢ Bad Smell: Código duplicado.
        - ✓ Solução: Alterar busca para findById e remover código desnecessário.
    - Método updateProduto:
        - ➢ Bad Smell: Código duplicado.
        - ✓ Solução: Alterar busca para findById e remover código desnecessário.
    - Método deleteUser:
        - ➢ Bad Smells: Método deleta produtos e não usuários. Nome de variável dentro do método também impróprio. Código duplicado.
        - ✓ Soluções: Renomeação do método para deleteProduto e da variável para produto, utilização do método findById e remoção do código desnecessário.

- Classe ProdutoService:
    - ➢ Bad Smells: A classe está populando tabelas direto do service. Usa um atributo para calcular o id manualmente. Método deleteAllUsers deleta produtos. Método saveLote não deveria estar na classe.
    - ✓ Soluções: Criação de um BD para guardar as informações. Remover manipulação de lotes. Alterar nome do método deletaAllUsers para deleteAllProdutos.

- Classe LoteService:
    - ➢ Bad Smells: Guarda os produtos em uma lista. Usa um atributo para calcular id manualmente. Método updateProduto atualiza lote e não produto. Método getIterator retorna null.
    - ✓ Soluções: Guardar produtos no BD. Remover o atributo id calculado manualmente. Alterar nome do método updateProduto para updateLote. Retornar o iterator no método getIterator.

- Classe Produto:

    ➢ Bad Smell: Construtor cria produto vazio
    ✓ Solução: Remover construtor

4. User Stories obrigatórias implementadas:

User Story 1:

Foram injetadas as seguintes dependências no sistema:
- Banco de Dados H2;
- JPA;
- Dev Tools.

As seguintes classes foram criadas:
- FacadeService e FacadeServiceImpl;
- Os métodos que estavam na RestApiController mudaram para seus respectivos novos controllers (Produto e Lote);
- Criação de DTO para Produto;
- Foram criados as classes de persistência de dados (ProdutoRepository e LoteRepository);
- A classe DummyData faz a inserção de produtos e lotes no banco de dados para testar.

User Story 2:

As seguintes classes foram criadas:
- Admin (representação de um administrador no sistema);
- AdminService;
- AdminController;
- AdminDTO ;
- AdminRepository.

Os seguintes métodos foram criados na api:
- criarAdmin: instancia e salva um administrador no banco de dados;
- fazerLoginAdmin: desde que esteja criado, um administrador pode logar no sistema com seu CPF e senha para realizar as operações posteriormente criadas.

A seguinte alteração foi feita:
- A classe DummyData faz a inserção de admins e o login de um deles no banco de dados para testar.

User Story 3:

As seguintes alterações foram feitas:

- Ajustes no arquivo pom.xml;
- O método criarProduto passa a ser responsabilidade do controller de produto ao invés de ser responsabilidade da controller da api.

User Story 4:

A seguinte classe foi criada:

- ProdutoDTODescricaoUsuario: dá a descrição de um produto para um usuário que não precisa estar logado no sistema, essa descrição oculta certos atributos do produto que não convêm ao usuário visualizar, apenas o admin.

O seguinte método foi criado na api:

- consultarProduto: um usuário pode consultar informações sobre determinado produto sem estar logado no sistema.

User Story 5:

As seguintes classes foram criadas:

- OrdenaProdutoPorNome.
- OrdenaProdutoPorCategoria.
- OrdenaProdutoPorPreco.
- OrdenaProdutoPorSituacao.
- OrdenaProdutoPorFabricante.
- OrdenaVendaPorQTDEDeItens.
- OrdenaVendaPorPrecoTotal.

Essas classes implementam a interface Comparator, que define o comportamento de ordenação baseado em algum critério.

Os seguintes métodos foram criados na api:

- listarProdutosOrdenados: o administrador, desde que esteja logado, pode gerar uma lista dos produtos registrados, ordenados por um atributo específico (nome, categoria, preço, situação e fabricante);
- listarVendasOrdenadas: o administrador, desde que esteja logado, pode gerar uma lista das vendas registradas, ordenadas por um atributo específico (quantidade de itens e preço total).

User Story 6:

As seguintes classes foram criadas:

- Categoria (categoria de um produto, o contexto em que ele está inserido, cada categoria pode ter um desconto que pode ser associado ao preço do produto);
- CategoriaService;

- CategoriaController;
- CategoriaRepository.

Os seguintes métodos foram criados na api:
- criarCategoria: instancia e salva uma categoria no banco de dados;
- listarCategorias: lista as categorias existentes no sistema;
- adicionarDesconto: atribui um desconto a determinada categoria, esse desconto é realizado no preço do produto posteriormente.

A seguinte alteração foi feita:
- A classe DummyData faz a inserção de categorias no banco de dados para testar.

User Story 7:

A seguinte classe foi criada:
- Relatorio: contem uma lista de lotes, lista de produtos e lista de vendas, a classe é responsável por fazer o relatório geral do mercado.

O seguinte método foi criado na api:
- relatorioMercado: é responsável por realizar o relatório.

User Story 8:

As seguintes alterações foram feitas:
- Ajuste de algumas condições;
- Correção e adição de novas validações;
- Tornando produto indisponível ao seu lote zerar;
- Visualização de produtos indisponíveis sem preço.

User Story 9:

As seguintes alterações foram feitas:
- Produto passa a ter um atributo booleano produtoVencido que funciona como uma flag para sabermos quando um produto está vencido ou não. Seguido disso, foi criado um método isProdutoVencido, dentro de Produto, que retorna uma variável booleana para que o controller de produto possa fazer determinadas verificações ao listar os produtos vencidos.

O seguinte método foi criado na api:
- listProdutosVencidos: retorna uma lista com os produtos vencidos.

User Story 10:

As seguintes classes foram criadas:
- Venda (representa a venda de um ou mais produtos no sistema);

- VendaController;
- VendaRepository;
- VendaSevice;
- VendaServiceImpl.

O seguinte método foi criado na api:

- registrarVenda: realiza a venda de um ou mais produtos, de tipos e quantidades diferentes, caso os mesmos estejam disponíveis.

User Story 11:

As seguintes classes foram criadas:

- Item (representa um item em determinada venda do sistema, que é basicamente composto por um produto e sua respectiva quantidade);
- ItemRepository.

O seguinte método foi criado na api:

- cancelarVenda: cancela uma venda e torna os produtos vinculados a ela disponíveis novamente.

A seguinte alteração foi feita:

- Ajuste no cálculo de preço da venda com desconto em sua categoria.

User Story 12:

Os seguintes métodos foram criados na api:

- listarProdutosBaixa: lista os produtos que estão com quantidades abaixo de 15 no sistema;
- listarLotesBaixa: lista os lotes que possuem produtos que estão com quantidades abaixo de 15 no sistema.

User Story 13:

A seguinte alteração foi feita:

- Lote passa a ter o método isLoteProximoDoVencimento, que retorna uma booleana de acordo com a diferença entre a data de vencimento do lote e a data atual, se for menor ou igual a 30 dias esse lote está próximo do vencimento.

O seguinte método foi criado na api:

- listarLotesProximosDoVencimento: um administrador pode consultar os produtos que estão com pelo menos 30 dias de proximidade em relação a data de validade.

5. Principais decisões de Design:

Para a estrutura base do projeto, decidimos utilizar o padrão de projeto **facade**. No projeto, as classes do subsistema representam diferentes serviços para as diferentes funcionalidades disponibilizadas pelo mercado. É importante ressaltar também que as classes do subsistema não têm referência à fachada. As classes não têm conhecimento de nenhuma fachada e são projetadas para funcionar de forma independente, mesmo que não exista uma fachada. As classes do subsistema são usadas pela fachada, mas não o contrário.

Para a implementação da fachada, fornecemos uma interface **FacadeService**, na qual possui todos os métodos que são disponibilizados para realização de alguma ação/serviço. Embora o padrão não exija uma interface, foi criada uma, com base no que resume o *princípio da inversão de dependência*. Dessa forma, podemos ter clientes que interagem com os serviços por meio da fachada. É uma forma também de diminuir o acoplamento entre as classes.

Para implementação dos serviços, disponibilizamos a classe **FacadeServiceImpl** que implementa a **FacadeService**. Sendo assim, a classe têm todos os métodos que serão manipulados. A classe de implementação possui todos os controladores (produto, venda, lote, admin, categoria) e esses controladores por sua vez são responsáveis pelas suas próprias funções. Isso possibilita o baixo acoplamento entre as classes, uma vez que uma classe maior (**FacadeServiceImpl**) tem conhecimento de todas as outras e através dela é possível fazer todas as manipulações necessárias. O cliente da nossa aplicação é o **RestApiController**, no qual tem uma **FacadeService**.

Os controladores do sistema fazem apenas o que devem fazer, cada um deles tem uma classe serviço (Ex.: ProdutoController tem ProdutoService que é uma interface que disponibiliza os serviços que ela pode fazer como criarProduto, salvarProduto, deletar e assim por diante. Da mesma forma acontece para os demais controladores).

Aplicando esse padrão, podemos notar que quando é necessário ter conhecimento de mais de uma classe para realizar alguma ação (Ex: Para realizar uma venda eu preciso conhecer o ControllerProduto e o ControllerVenda), através da **FacadeServiceImpl**, podemos fazer isso sem que ocorra um alto acoplamento entre as classes necessárias.