Documentação

(Entrega 03)

Projeto de Software

Professor: Fábio Morais

Grupo: 07

Lázaro Queiroz do Nascimento - 123110628
Matheus Galdino de Souza - 123111147
João Lucas Gomes Brandão - 123110397
Igor Raffael Menezes de Melo - 123110549
Vinícius de Oliveira Porto - 123110505

Descrição USs Implementadas

US09 - Solicitação de Compra: Igor Raffael (100%)

Descrição

Foi implementado o endpoint responsável pela solicitação de compra de ativos por parte do cliente. Para isso, foi criado o CRUD de **Compra**, que herda de uma classe abstrata chamada **Transacao**, a qual também será utilizada futuramente em operações de resgate.

A lógica permite que o cliente solicite a compra de um ativo disponível para o seu plano, desde que possua saldo suficiente em sua carteira. A verificação de saldo é feita de forma direta, sem necessidade de iterar sobre os ativos da carteira.
Como parte da implementação, o módulo de **Carteira** foi refatorado:

- Os ativos são armazenados de forma indireta por meio de uma associação entre o ID do ativo e uma entidade auxiliar chamada **AtivoCarteira**, que armazena a quantidade total e o valor acumulado.

- As compras não são armazenadas diretamente na carteira; ao invés disso, qualquer checagem relacionada a elas é feita diretamente via banco de dados.

Além disso, foram realizados testes para garantir o correto funcionamento da solicitação de compra, considerando diferentes cenários de saldo, plano e disponibilidade de ativos.

Além disso, pequenas refatorações foram feitas por todo código, as quais podem ser vistas abaixo nas decisões de design e no log de commits do projeto.

US10 - Acompanhamento de Status da Compra: Vinicius (100%)

Descrição

Foi implementada a rota que permite ao cliente acompanhar o status de todas as compras realizadas, retornando para cada uma delas as informações de data, ativo, quantidade, valor e estado atual. Os estados possíveis da compra são: "Solicitado", "Disponível", "Comprado" e "Em carteira". Para garantir que o fluxo siga a ordem correta dos estados, optou-se por implementar a mudança entre eles utilizando o padrão State.

Além disso, foram implementados testes de integração que asseguram que o cliente consiga acompanhar o status de suas compras em todos os estados possíveis, incluindo cenários com múltiplas compras em diferentes estágios.

US11 - Confirmação de Compra pelo Administrador: Lázaro Queiroz (100%)

Descrição

Foi implementada a rota que permite ao administrador aprovar ou recusar uma solicitação de compra de ativo pelo cliente, o que muda o status da compra aprovada de `SOLICITADO` para `DISPONÍVEL`, caso seja aprovada ou mantém-se no status inicial caso recusada.

Além disso, depois da confirmação de aprovação ou recusa, o usuário que realizou a determinada compra é notificado do desdobramento da ação do administrador.

Ademais, testes de integração foram implementados para garantir a corretude da feature implementada.

US12 - Confirmação de Compra pelo Cliente: João Lucas (100%)

Descrição

Foi implementada a rota que permite confirmar uma compra previamente liberada pela corretora. Dessa forma, o cliente pode finalizar a transação de forma segura, garantindo que o valor seja debitado do seu saldo e que o ativo seja transferido para sua carteira. Ao confirmar, a compra passa pelos estados Disponível → Comprado → Em Carteira, assegurando que a operação seja concluída de ponta a ponta, desde a liberação feita pela corretora até a posse do ativo pelo cliente.

Antes da finalização, o sistema valida se a compra realmente pertence ao cliente, se ainda está no estado “Disponível” e se há saldo suficiente para a operação. Uma vez concluída, a compra é marcada como “Em Carteira”, representando que o ativo já está sob controle do cliente e disponível para acompanhamento e futuras transações. Também foram implementados testes automatizados cobrindo os principais cenários da funcionalidade.

US13 - Visualização da Carteira de Investimentos: Matheus Galdino (100%)

Descrição

Foi implementada a rota que permite o cliente visualizar sua carteira de investimentos, contendo todos seus ativos e respectivos detalhes, como tipo, quantidade, valor de aquisição, valor atual e desempenho (lucro ou prejuízo).

Além disso, também foram criados os testes de integração para garantir o desempenho esperado, ou seja, retornar a carteira, quando ela possui ativos ou está vazia, e retornar exceção, caso o cliente não exista / seja encontrado.

Decisões de Design

1. Estratégias de Modelagem de Ativos

- **Padrão Strategy:** utilizado para a definição dinâmica do comportamento de ativos com base em seu tipo (Ação, CriptoMoeda, TesouroDireto).
- **Conversão das interfaces para classes abstratas:** necessária para possibilitar a persistência com *JPA*, mantendo a extensibilidade e coesão da arquitetura.
- **Enum TipoAtivo exclusivo no DTO:** utilizado para reduzir o acoplamento entre os DTOs e a estrutura interna da aplicação.
- **Enum StatusDisponibilidade em vez de boolean:** melhora a legibilidade código e reduz a possibilidade de erros lógicos.

2. Integração entre Serviços

- **Facade entre AtivoService e ClienteService:** Criação da classe *AtivoClienteService* para intermediar a comunicação entre os serviços de Cliente e Ativo, especialmente para a US05, promovendo uma separação das responsabilidades e organização da lógica de negócio.
- **Repositórios separados para Cliente e Administrador:** apesar de existir apenas um administrador, foi decidida a separação dos repositórios para manter a clareza e a responsabilidade única de cada camada.
- **Autenticação simplificada:** embora tenha sido testada a criação de *um AutenticacaoService*, optou-se por verificar diretamente o *codigoAcesso* nos repositórios, de acordo com a autoridade exigida por cada funcionalidade.

3. Desacoplamento entre DTOs e Domínio

- O enum *TipoAtivo* é utilizado exclusivamente no DTO de atualização (Patch), evitando exposição da estrutura interna do domínio.
- Essa abordagem facilita a validação automática e garante segurança tipada com valores restritos.
- A lógica de ativação e desativação de ativos foi encapsulada dentro da própria entidade, centralizando as regras de negócio e promovendo a coesão.

4. Lógica Centralizada no Service

- Toda a lógica de negócio relativa à atualização de ativos foi centralizada na classe *AtivoServiceImpl*.
- O controller atua como uma camada de delegação, encaminhando as requisições ao serviço.
- Para evitar o uso de instanceof, a estrutura dos ativos foi modificada para incluir um enum *TipoAtivo*, permitindo o uso de métodos como *getTipo()* e *getNomeTipo()* diretamente, aumentando legibilidade e coesão.

5. Herança e Estrutura de Usuários

- A classe abstrata *Usuario* agrupa atributos comuns a *Cliente* e *Administrador* (como id, nome e codigoAcesso).
- A herança é persistida com a estratégia *SINGLE_TABLE* do JPA, utilizando @DiscriminatorColumn para diferenciar os tipos de usuários.
  A classe *Cliente* representa a entidade persistente e está localizada no pacote model.
  Utiliza-se *ClientePostPutRequestDTO* para entrada de dados e *ClienteResponseDTO* para saída, evitando a exposição de informações sensíveis.
- O mapeamento entre DTOs e entidade é realizado com o ModelMapper, configurado na classe *ModelMapperConfig*, com typeMap específico que ignora o id durante atualizações.
- O enum *TipoPlano* define os diferentes planos de forma tipada e segura, garantindo consistência nos dados de domínio.

6. Composição de Funcionalidade entre Serviços

- Foi necessário utilizar serviços e controllers tanto de *Cliente* quanto de *Ativo*.
- Para evitar acoplamento excessivo e garantir organização, foi criado um novo controller que integra os dois serviços, funcionando como um ponto de orquestração das regras de negócio relacionadas à listagem de ativos conforme o plano do cliente.

7. Status de Compra (e futuramente Status de Resgate)

- **Padrão State:** utilizado para implementar o fluxo de estados da compra através da interface EstadoCompraState que define o comportamento comum, e quatro classes concretas que a implementam, representando cada estado possível da compra: “Solicitado”, “Disponível”, “Comprado” e “Em carteira”. Dessa forma, cada classe é responsável por encapsular a lógica de transição para o próximo estado permitido, garantindo que o fluxo siga corretamente a ordem estabelecida pelas regras de negócio.
- **ENUM EstadoCompra:** como o JPA não permite a persistência direta de interfaces no banco de dados, foi necessário criar um ENUM para mapear o estado persistente no banco e a instância concreta da classe correspondente,

além de ser utilizado também no DTO da compra. O ENUM é atualizado por meio do método definido no padrão State, que realiza a transição do estado da compra de acordo com a classe atualmente instanciada, garantindo consistência entre o valor persistido e o comportamento da aplicação. Essa abordagem permite que a lógica de negócio permaneça isolada nas classes concretas que implementam a interface, mantendo o modelo coeso e de fácil evolução.

Diagrama