Documentação

(Entrega 04)

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

US14 - Resgatar ativos: Igor Raffael (100%)

Descrição

Foi-se implementado toda a rota para o CRUD de transações do tipo resgates - ou seja, controller, service, repositories, models, interface, enums e derivados - que permitam ao cliente da API solicitar o resgate de um ativo, dado que atenda às restrições necessárias: possuir quantidade suficiente do ativo e ter aprovação do administrador.

Ademais, foram criados os templates (implementação parcial) das rotas necessários à visualização dos resgates e cálculo dos impostos para permitir a testagem do que foi exposto acima e facilitar a implementação das USs futuras.

Por fim, foram também implementados testes de integração para garantir o funcionamento das partes do sistema envolvidas na operação de solicitar resgate.

US15 - Calcular automaticamente imposto devido sobre o resgate: Vinícius Porto (100%)

Descrição

Foi realizado o cálculo do imposto após o cliente solicitar o resgate de um ativo comprado. O cálculo é feito através do método calcular imposto na classe abstrata TipoDeAtivo. Dessa forma, cada subclasse (Acao, TesouroDireto, Criptomoeda) encapsula sua própria lógica de cálculo, garantindo maior manutenibilidade e extensibilidade do sistema.

Além disso, foram implementados testes para validar o cálculo correto da alíquota de imposto de acordo com o tipo de ativo e a isenção de imposto nos casos em que não houver lucro ou houver prejuízo.

US16 - Acompanhar status de cada resgate iniciado: João Brandão (100%)

Descrição

Foi implementado todo o fluxo de resgates de ativos seguindo o padrão State – ou seja, controller, service, repository, models, enums e classes de estado – que permitem ao cliente da API solicitar um resgate, acompanhar seu status e executar a finalização do resgate.

O sistema garante que o resgate só seja aprovado se o administrador autorizar e só seja executado quando estiver no estado Confirmado. As classes de estado

(Solicitado, Confirmado e EmConta) encapsulam a lógica de transição entre estados, garantindo consistência e rastreabilidade do processo.

O ResgateController expõe endpoints para solicitar, consultar, listar, executar e atualizar o status do resgate, enquanto o ResgateServiceImpl gerencia as regras de negócio, como validação de cliente, cálculo de impostos e aplicação do valor na conta. Além disso, foram implementados testes de integração visando assegurar a qualidade do código produzido e segurança no comportamento desejado.

US17 - Confirmação de Resgate pelo Administrador: Matheus Galdino(100%)

Descrição

Implementação da funcionalidade que permite ao administrador aprovar ou recusar uma solicitação de resgate de um cliente. Quando isso ocorre, o cliente recebe uma notificação de alerta sobre a decisão do administrador. Para tanto, foi criada a rota que, com o id do resgate, o código de acesso do administrador e sua decisão, chama atualizarStatusResgate e realiza a lógica necessária.

Testes de integração com o Controller foram criados, averiguando quando a operação funciona como esperado, tanto para aprovação quanto para rejeição de um resgate. Além de também verificar os casos que lançam exceção, são eles: para código do administrador é inválido, para resgate não encontrado, para cliente ou para ativo do resgate não encontrado e para resgates com estado diferente de solicitado (ou seja, confirmado ou em conta).

US18 - Consultar todo o histórico de resgate e compras pelo cliente: Lázaro Queiroz (100%)

Descrição

Implementação da rota que permite que clientes do sistema consultem seus históricos de transação, permitindo-os também aplicar filtros nessas listagem, de acordo com o período de solicitação de cada uma delas, no tipo de ativo ao qual elas se referem, no tipo de operação (Compra ou Resgate) bem como no status da operação (Solicitada, Disponível, Em Carteira [para Compras], Em Conta [para Resgates], etc.).

Além disso, testes de integração foram implementados para garantir a integridade da funcionalidade e de partes do sistema envolvidas no seu funcionamento.

US19 - Consultar todas as operações realizadas: Vinícius Porto (50%) e Matheus Galdino (50%)

Descrição

Implementação da rota para permitir que o administrador visualize todas as transações realizadas por todos os clientes, com filtro pelo cliente, tipo da transação (compra ou resgate), tipo de ativo e data. Nessa US foi utilizado o padrão Strategy, por meio da interface TransacaoStrategy, que foi implementada pelo service de compras e de resgates, no qual o permite que cada service (compra e resgate) seja responsável por listar todos seus objetos armazenados, permitindo maior manutenibilidade, extensibilidade e evolução futura do sistema.

Também foram implementados os testes que verificam se cada tipo de filtro está funcionando adequadamente e se funciona quando múltiplos filtros são ativados.

US20 - Exportar um extrato completo em CSV das operações realizadas pelo cliente: João Brandão (100%)

Descrição

Foi implementado o recurso de exportação de extrato de operações do cliente em formato CSV – ou seja, controller, service e DTOs foram adaptados para suportar essa funcionalidade. O ClienteController expõe o endpoint /clientes/{idCliente}/extrato, que valida o código de acesso do cliente e retorna um arquivo CSV contendo todas as transações, incluindo compras e resgates, com informações de ativo, quantidade, impostos, valores e datas de solicitação e finalização.

TransacaoService gera o CSV, construindo seu conteúdo a partir das transações do cliente e formatando corretamente datas e valores. O sistema garante que apenas transações pertencentes ao cliente sejam incluídas no extrato, assegurando segurança e integridade dos dados. Foram implementados também testes de integração para verificar o correto funcionamento do endpoint e a geração precisa do CSV.

Essa implementação permite ao cliente obter um histórico completo de suas operações de forma padronizada e facilmente manipulável em planilhas, mantendo rastreabilidade de todas as movimentações financeiras.

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.

8. Consultar Operações

- **Padrão Strategy:** utilizado para que cada service da transação específica (compra ou resgate) implemente a interface TransacaoStrategy e se torna responsável por listar apenas suas próprias operações. Essa abordagem segue o princípio do aberto para extensão e fechado para modificação (OCP), permitindo que novos tipos de transações sejam adicionados futuramente sem impactar as implementações já existentes. Além disso, promove baixo acoplamento, alta coesão e garante que a lógica de cada operação esteja isolada, facilitando testes, manutenção e evolução do sistema.

9. Confirmação de Transação

- O resgate precisa ser confirmado pelo cliente após o administrador autorizar a transação, assim como ocorre com a compra.

Diagrama de Classes