Documentação

Entrega 1

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

US01 - Implementação do CRUD de Ativos
Responsáveis João Lucas (50%), Igor Raffael (50%)

Descrição

Implementação completa do CRUD para ativos da aplicação. Essa funcionalidade inclui a criação e integração de de diversas camadas e componentes relacionados à entidade Ativo, além de preparar para funcionalidades futuras.

Os componentes Implementados foram:

- **Classe Ativo:** classe principal que representa um ativo genérico no sistema, servindo como superclasse para os tipos concretos de ativos.
- **Subclasses Acao, Criptomoeda e TesouroDireto:** representam os diferentes tipos específicos de ativos, cada um possui lógica própria, permitindo diferenciar comportamentos e atributos (caso venham a existir em USs futuras), por meio do padrão Strategy.
- **Enum StatusDisponibilidade:** Criado para substituir o uso de booleanos simples, melhorando a clareza do estado do ativo e evitando ambiguidade na lógica de disponibilidade.
- **Enum TipoAtivo (uso exclusivo nos DTOs):** representa os tipos de ativo como ACAO, CRIPTOMOEDA e TESOURO_DIRETO, sendo utilizado apenas nos DTOs para evitar acoplamento direto com a lógica interna do domínio.
- **Interface TipoDeAtivo (convertida em classe abstrata):** inicialmente uma interface usada para aplicar o padrão Strategy, foi convertida em classe abstrata para permitir persistência via JPA.
- **Padrão Strategy aplicado à tipagem de ativos:** cada tipo de ativo implementa comportamentos específicos definidos em *TipoDeAtivo*, promovendo flexibilidade e modularidade.
- **DTOs criados**
    - **AtivoResponseDTO:** utilizado para formatar as respostas da API, encapsulando dados relevantes ao ativo.
    - **AtivoPostPutRequestDTO:** representa os dados necessários para a criação e atualização de ativos.
- **Camadas Arquiteturais**
    - **AtivoService:** centraliza a lógica de negócio relacionada aos ativos.
    - **AtivoController:** responsável por expor os endpoints do CRUD via API REST.
    - **AtivoRepository:** realiza a persistência de ativos no banco de dados.
    - **TipoAtivoRepository:** permite o acesso estruturado aos tipos de ativos disponíveis.

- **Testes de Integração:** foram desenvolvidos testes automatizados para garantir o correto funcionamento das operações de criação, listagem, edição e exclusão de ativos, validando os fluxos completos da aplicação.

US02 - Ativação e Desativação de Ativos Cadastrados
Responsáveis: Igor Raffael (50%), João Lucas (50%)

Descrição

Implementação da funcionalidade que permite ao administrador alterar o status de disponibilidade de ativos previamente cadastrados no sistema, ativando-os ou desativando-os conforme a necessidade. Essa funcionalidade é fundamental para garantir o controle adequado dos recursos, permitindo que apenas ativos disponíveis sejam utilizados em operações do sistema.

A lógica de negócio associada à alteração de status foi centralizada no serviço de ativos, garantindo coesão e facilitando a manutenção. Adicionalmente, foram desenvolvidos testes automatizados que asseguram o correto funcionamento da funcionalidade, cobrindo cenários de ativação, desativação e casos inválidos.

Os componentes implementados foram:

- **Método ativarOuDesativarAtivo() na classe AtivoController:** responsável por expor a operação via endpoint para o administrador.
- **Método ativarOuDesativar() na interface AtivoService:** define o contrato da lógica de negócio.
- **Implementação do método ativarOuDesativar() na classe AtivoServiceImpl:** contém a lógica para alternar o status de disponibilidade do ativo.
- **Testes de Integração:** cobrindo cenários de ativação e desativação dos ativos.

US03 - Atualização da Cotação dos Ativos do Tipo Ação ou Criptomoeda
Responsável: Matheus Galdino (100%)
Descrição

Implementação da funcionalidade que permite ao administrador atualizar o valor (cotação) de ativos do tipo Ação ou Criptomoeda.

Os componentes implementados foram:

- **Enum TipoAtivo:** criado para representar os diferentes tipos de ativo e facilitar a filtragem de quais tipos podem ter sua cotação atualizada.
- **Exceções personalizadas:** “CotacaoNaoPodeSerAtualizadaException” lançada quando o tipo de ativo não é elegível para atualização, nesse caso, apenas Tesouro Direto, e “VariacaoMininaDeCotacaoNaoAtingidaException” lançada quando a nova cotação não atinge a variação mínima de 1%.
- **Método getTipo() em TipoDeAtivo:** Adicionado à interface TipoDeAtivo e implementado nas subclasses para retornar o tipo como um enum, permitindo a filtragem segura e clara durante a lógica de atualização.

- **Método atualizarCotacao() no AtivoServiceImpl:** responsável por validar o tipo de ativo, calcular a variação e atualizar a cotação caso os critérios sejam atendidos.
- **Testes de Integração:** foram criados testes para verificar (1) o correto funcionamento da lógica de variação mínima, (2) a rejeição de tentativas de atualização de ativos do tipo não permitido e (3) a aplicação correta das exceções.

US04 - Implementação do CRUD de Cliente
Responsável: Vinícius Porto (100%)

Descrição

Implementação de CRUD para clientes com os atributos requisitados. Não é permitido a exibição de código de acesso dos clientes e há verificação do código para operações do cliente (exceto leitura), permitindo que apenas o cliente edite seu próprio cadastro.

Classes implementadas: ClienteController, Cliente, ClientePostPutRequestDTO, ClienteResponseDTO, Usuario, TipoPlano, ClienteRepository, ClienteService, ClienteServiceImpl e ModelMapperConfig.

Na classe ClienteControllerTests foram implementados 44 testes de integração com o objetivo de validar o correto funcionamento dos endpoints da API relacionados à entidade Cliente. Esses testes cobrem todas as operações CRUD, incluindo os verbos HTTP GET, POST, PUT e DELETE, garantindo que as funcionalidades de criação, recuperação, atualização e remoção de clientes funcionem conforme o esperado. Além disso, os testes realizam validações detalhadas sobre os campos dos DTOs utilizados (ClientePostPutRequestDTO e ClienteResponseDTO), assegurando que as mensagens de erro sejam corretamente retornadas em casos de entrada inválida, como campos obrigatórios ausentes ou formatos incorretos.

US05 - Visualização de Ativos por Plano de Assinatura
Responsável: Lázaro Queiroz (100%)

Descrição

Implementação da funcionalidade que permite ao cliente visualizar os ativos disponíveis conforme seu plano de assinatura.

- Clientes do plano Normal podem visualizar apenas ativos do tipo Tesouro Direto.
- Clientes do plano Premium podem visualizar todos os tipos de ativos: Tesouro Direto, Ações e Criptomoedas.

  As tarefas realizadas foram:
- Criação das classes *AtivoClienteController*, *AtivoClienteService* e *AtivoClienteImpl* responsáveis por mediar a interação entre os serviços de Ativo e de Cliente, realizando a filtragem dos ativos com base no plano de

assinatura do cliente. Essa abordagem procura reduzir o acoplamento entre os módulos e manter a separação de responsabilidades.
- Criação da classe *TipoAtivoRepository* para permitir uma interação estruturada com os tipos de ativos, possibilitando uma única referência de tipo reutilizável por todos os ativos existentes.
- Criação de testes de integração para garantir que (1) clientes do plano Normal só tenham acesso aos ativos permitidos e (2) clientes do plano Premium visualizem todos os tipos de ativos.

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.

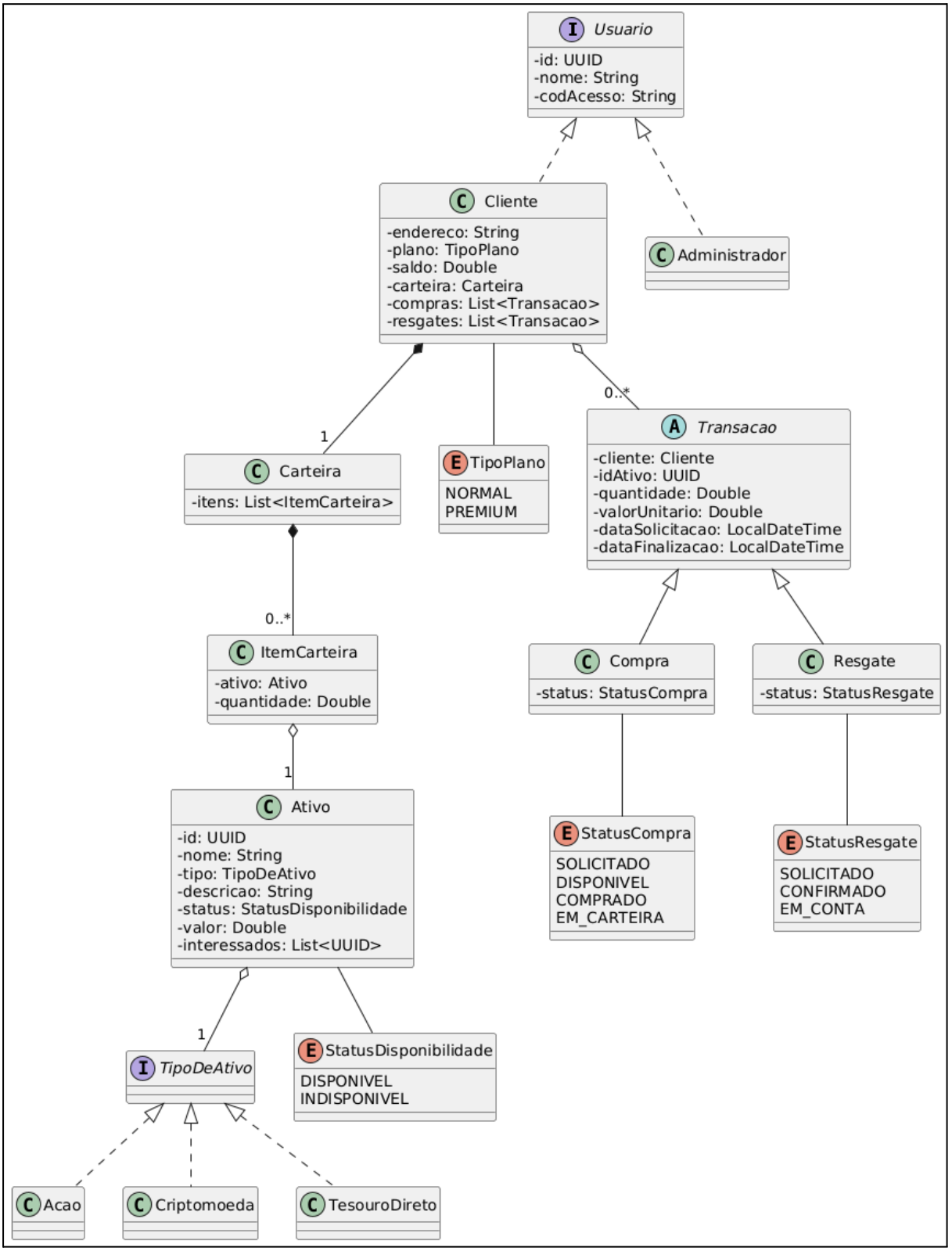
5. Herança e Estrutura de Usuários

- 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.

Diagrama


Usuario
-id: UUID
-nome: String
-codAcesso: String
Cliente
-endereco: String
-plano: TipoPlano
-saldo: Double
-carteira: Carteira
-compras: List<Transacao>
-resgates: List<Transacao>
Administrador
1
0..*
Carteira
-itens: List<ItemCarteira>
TipoPlano
NORMAL
PREMIUM
Transacao
-cliente: Cliente
-idAtivo: UUID
-quantidade: Double
-valorUnitario: Double
-dataSolicitacao: LocalDateTime
-dataFinalizacao: LocalDateTime
0..*
ItemCarteira
-ativo: Ativo
-quantidade: Double
1
Compra
-status: StatusCompra
Resgate
-status: StatusResgate
Ativo
-id: UUID
-nome: String
-tipo: TipoDeAtivo
-descricao: String
-status: StatusDisponibilidade
-valor: Double
-interessados: List<UUID>
StatusCompra
SOLICITADO
DISPONIVEL
COMPRADO
EM_CARTEIRA
StatusResgate
SOLICITADO
CONFIRMADO
EM_CONTA
1
TipoDeAtivo
StatusDisponibilidade
DISPONIVEL
INDISPONIVEL
Acao
Criptomoeda
TesouroDireto